

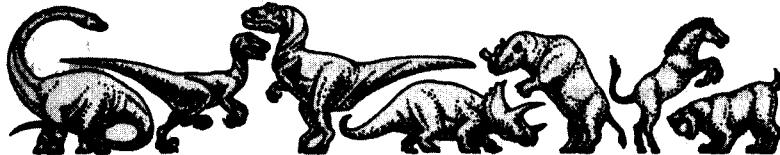
**SILBERSCHATZ
GALVIN
GAGNE**

**OPERATING
SYSTEM
CONCEPTS**

SIXTH EDITION

OPERATING SYSTEM CONCEPTS

Sixth Edition



ABRAHAM SILBERSCHATZ

Bell Laboratories

PETER BAER GALVIN

Corporate Technologies, Inc.

GREG GAGNE

Westminster College



JOHN WILEY & SONS, INC.

New York / Chichester / Weinheim / Brisbane / Singapore / Toronto

ACQUISITIONS EDITOR	Paul Crockett
SENIOR MARKETING MANAGER	Katherine Hepburn
SENIOR PRODUCTION EDITOR	Ken Santor
COVER DESIGNER	Madelyn Lesure
COVER ART	Susan E. Cyr
SENIOR ILLUSTRATION COORDINATOR	Anna Melhorn

This book was set in Palatino by Abraham Silberschatz and printed and bound by Courier-Westford. The cover was printed by Phoenix Color Corporation.

This book is printed on acid-free paper.

The paper in this book was manufactured by a mill whose forest management programs include sustained yield harvesting of its timberlands. Sustained yield harvesting principles ensure that the numbers of trees cut each year does not exceed the amount of new growth.

Copyright © 2002 John Wiley & Sons, Inc. All rights reserved.

No part of this publication may be reproduced, stored in a retrieval system or transmitted in any form or by any means, electronic, mechanical, photocopying, recording, scanning or otherwise, except as permitted under Sections 107 or 108 of the 1976 United States Copyright Act, without either the prior written permission of the Publisher, or authorization through payment of the appropriate per-copy fee to the Copyright Clearance Center, 222 Rosewood Drive, Danvers, MA 01923, (508) 750-8400, fax (508) 750-4470. Requests to the Publisher for permission should be addressed to the Permissions Department, John Wiley & Sons, Inc., 605 Third Avenue, New York, NY 10158-0012, (212) 850-6011, fax (212) 850-6008, E-Mail: PERMREQ@WILEY.COM.

41
TWR
1940/6

ISBN 0-471-41743-2

Printed in the United States of America

10 9 8 7 6 5 4 3 2 1

*To my mother, Wira,
my wife, Haya,
and my children, Lemor, Sivan, and Aaron*

Avi Silberschatz

*To my wife, Carla,
and my children, Gwendolyn and Owen*

Peter Baer Galvin

*To my parents, Marlene and Roland,
my wife, Pat, and my sons, Tom and Jay*

Greg Gagne

PREFACE

Operating systems are an essential part of any computer system. Similarly, a course on operating systems is an essential part of any computer-science education. This field is undergoing change at a breathtakingly rapid rate, as computers are now prevalent in virtually every application, from games for children through the most sophisticated planning tools for governments and multinational firms. Yet the fundamental concepts remain fairly clear, and it is on these that we base this book.

We wrote this book as a text for an introductory course in operating systems at the junior or senior undergraduate level or at the first-year graduate level. It provides a clear description of the *concepts* that underlie operating systems. As prerequisites, we assume that the reader is familiar with basic data structures, computer organization, and a high-level language, such as C. The hardware topics required for an understanding of operating systems are included in Chapter 2. For code examples, we use predominantly C as well as some Java, but the reader can still understand the algorithms without a thorough knowledge of these languages.

The fundamental concepts and algorithms covered in the book are often based on those used in existing commercial operating systems. Our aim is to present these concepts and algorithms in a general setting that is not tied to one particular operating system. We present a large number of examples that pertain to the most popular operating systems, including Sun Microsystems' Solaris 2, Linux; Microsoft MS-DOS, Windows NT, and Windows 2000; DEC VMS and TOPS-20, IBM OS/2, and the Apple Macintosh Operating System.

Concepts are presented using intuitive descriptions. Important theoretical results are covered, but formal proofs are omitted. The bibliographical notes contain pointers to research papers in which results were first presented and proved, as well as references to material for further reading. In place of proofs, figures and examples are used to suggest why we should expect the result in question to be true.

Content of this Book

The text is organized in seven major parts:

- **Overview:** Chapters 1 through 3 explain what operating systems *are*, what they *do*, and how they are *designed* and *constructed*. They explain how the concept of an operating system has developed, what the common features of an operating system are, what an operating system does for the user, and what it does for the computer-system operator. The presentation is motivational, historical, and explanatory in nature. We have avoided a discussion of how things are done internally in these chapters. Therefore, they are suitable for individuals or for students in lower-level classes who want to learn what an operating system is, without getting into the details of the internal algorithms. Chapter 2 covers the hardware topics that are important to an understanding of operating systems. Readers well-versed in hardware topics, including I/O, DMA, and hard-disk operation, may choose to skim or skip this chapter.
- **Process management:** Chapters 4 through 8 describe the process concept and concurrency as the heart of modern operating systems. A *process* is the unit of work in a system. Such a system consists of a collection of *concurrently* executing processes, some of which are operating-system processes (those that execute system code), and the rest of which are user processes (those that execute user code). These chapters cover methods for process scheduling, interprocess communication, process synchronization, and deadlock handling. Also included under this topic is a discussion of threads.
- **Storage management:** Chapters 9 through 12 deal with a process in main memory during execution. To improve both the utilization of CPU and the speed of its response to its users, the computer must keep several processes in memory. There are many different memory-management schemes. These schemes reflect various approaches to memory management, and the effectiveness of the different algorithms depends on the situation. Since main memory is usually too small to accommodate all data and programs, and since it cannot store data permanently, the computer system must provide secondary storage to back up main memory. Most modern computer systems use disks as the primary on-line storage medium for information,

both programs and data. The file system provides the mechanism for on-line storage of and access to both data and programs residing on the disks. These chapters describe the classic internal algorithms and structures of storage management. They provide a firm practical understanding of the algorithms used—the properties, advantages, and disadvantages.

- **I/O systems:** Chapters 13 and 14 describe the devices that attach to a computer and the multiple dimensions in which they vary. In many ways, they are also the slowest major components of the computer. Because devices differ so widely, the operating system needs to provide a wide range of functionality to applications to allow them to control all aspects of the devices. This section discusses system I/O in depth, including I/O system design, interfaces, and internal system structures and functions. Because devices are a performance bottleneck, performance issues are examined. Matters related to secondary and tertiary storage are explained as well.
- **Distributed systems:** Chapters 15 through 17 deal with a collection of processors that do not share memory or a clock—a *distributed system*. Such a system provides the user with access to the various resources that the system maintains. Access to a shared resource allows computation speedup and improved data availability and reliability. Such a system also provides the user with a distributed file system, which is a file-service system whose users, servers, and storage devices are dispersed among the sites of a distributed system. A distributed system must provide various mechanisms for process synchronization and communication, for dealing with the deadlock problem and the variety of failures that are not encountered in a centralized system.
- **Protection and security:** Chapters 18 and 19 explain the processes in an operating system that must be protected from one another's activities. For the purposes of protection and security, we use mechanisms that ensure that only those processes that have gained proper authorization from the operating system can operate on the files, memory segments, CPU, and other resources. Protection is a mechanism for controlling the access of programs, processes, or users to the resources defined by a computer system. This mechanism must provide a means for specification of the controls to be imposed, as well as a means of enforcement. Security protects the information stored in the system (both data and code), as well as the physical resources of the computer system, from unauthorized access, malicious destruction or alteration, and accidental introduction of inconsistency.
- **Case studies:** Chapters 20 through 22, in the book, and Appendices A through C, on the website, integrate the concepts described in this book by describing real operating systems. These systems include Linux, Windows 2000, FreeBSD, Mach, and Nachos. We chose Linux and FreeBSD because

UNIX—at one time—was almost small enough to understand, yet was not a “toy” operating system. Most of its internal algorithms were selected for *simplicity*, rather than for speed or sophistication. Both Linux and FreeBSD are readily available to computer-science departments, so many students have access to these systems. We chose Windows 2000 because it provides an opportunity for us to study a modern operating system that has a design and implementation drastically different from those of UNIX. We also cover the Nachos System, which allows students to get their hands *dirty*—to take apart the code for an operating system, to see how it works at a low level, to build significant pieces of the operating system themselves, and to observe the effects of their work. Chapter 22 briefly describes a few other influential operating systems.

The Sixth Edition

As we wrote this Sixth Edition, we were guided by the many comments and suggestions we received from readers of our previous editions, as well as by our own observations about the rapidly changing fields of operating systems and networking. We rewrote the material in most of the chapters by bringing older material up to date and removing material that was no longer of interest. We rewrote all Pascal code, used in previous editions to demonstrate certain algorithms, into C, and we included a small amount of Java as well.

We made substantive revisions and changes in organization in many of the chapters. Most importantly, we added two new chapters and reorganized the distributed systems coverage. Because networking and distributed systems have become more prevalent in operating systems, we moved some distributed systems material, client–server, in particular, out of distributed systems chapters and integrated it into earlier chapters.

- **Chapter 3, Operating-System Structures**, now includes a section discussing the Java virtual machine (JVM).
- **Chapter 4, Processes**, includes new sections describing sockets and remote procedure calls (RPCs).
- **Chapter 5, Threads**, is a new chapter that covers multithreaded computer systems. Many modern operating systems now provide features for a process to contain multiple threads of control.
- **Chapters 6 through 10** are the old Chapters 5 through 9, respectively.
- **Chapter 11, File-System Interface**, is the old Chapter 10. We have modified the chapter substantially, including the coverage of NFS from the Distributed File System chapter (Chapter 16).

- **Chapter 12 and 13** are the old Chapters 11 and 12, respectively. We have added a new section in Chapter 13, I/O Systems, covering STREAMS.
- **Chapter 14, Mass-Storage Structure**, combines old Chapters 13 and 14.
- **Chapter 15, Distributed System Structures**, combines old Chapters 15 and 16.
- **Chapter 19, Security**, is the old Chapter 20.
- **Chapter 20, The Linux System**, is the old Chapter 22, updated to cover new recent developments.
- **Chapter 21, Windows 2000**, is a new chapter.
- **Chapter 22, Historical Perspective**, is the old Chapter 24.
- **Appendix A** is the old Chapter 21 on UNIX updated to cover FreeBSD.
- **Appendix B** covers the Mach operating system.
- **Appendix C** covers the Nachos system.

The three appendices are provided online.

Teaching Supplements and Web Page

The web page for this book contains the three appendices, the set of slides that accompanies the book, in PDF and Powerpoint format, the three case studies, the most recent errata list, and a link to the authors home page. John Wiley & Sons maintains the web page at

<http://www.wiley.com/college/silberschatz>

To obtain restricted supplements, contact your local John Wiley & Sons sales representative. You can find your representative at the “Find a Rep?” web page:
<http://www.jsw-edcv.wiley.com/college/findarep>

Mailing List

We provide an environment in which users can communicate among themselves and with us. We have created a mailing list consisting of users of our book with the following address: os-book@research.bell-labs.com. If you wish to be on the list, please send a message to avi@bell-labs.com indicating your name, affiliation, and e-mail address.

Suggestions

We have attempted to clean up every error in this new Edition, but—as happens with operating systems—a few obscure bugs may remain. We would appreciate hearing from you about any textual errors or omissions that you identify. If you would like to suggest improvements or to contribute exercises, we would also be glad to hear from you. Please send correspondence to Avi Silberschatz, Vice President, Information Sciences Research Center, MH 2T-310, Bell Laboratories, 600 Mountain Ave., Murray Hill, NJ 07974 (avi@bell-labs.com).

Acknowledgments

This book is derived from the previous editions, the first three of which were coauthored by James Peterson. Others who helped us with previous editions include Hamid Arabnia, Randy Bentson, David Black, Joseph Boykin, Jeff Brumfield, Gael Buckley, P. C. Capon, John Carpenter, Thomas Casavant, Ajay Kumar Datta, Joe Deck, Sudarshan K. Dhall, Thomas Doeppner, Caleb Drake, M. Raşit Eskicioglu, Hans Flack, Robert Fowler, G. Scott Graham, Rebecca Hartman, Wayne Hathaway, Christopher Haynes, Mark Holliday, Richard Kieburtz, Carol Kroll, Thomas LeBlanc, John Leggett, Jerrold Leichter, Ted Leung, Gary Lippman, Carolyn Miller, Michael Molloy, Yoichi Muraoka, Jim M. Ng, Banu Özden, Ed Posnak, Boris Putanec, Charles Qualline, John Quarterman, Jesse St. Laurent, John Stankovic, Adam Stauffer, Steven Stepanek, Hal Stern, Louis Stevens, Pete Thomas, David Umbaugh, Steve Vinoski, Tommy Wagner, John Werth, and J. S. Weston.

We thank the following people who contributed to this edition of the book: Bruce Hillyer reviewed and helped with the rewrite of Chapters 2, 12, 13, and 14. Mike Reiter reviewed and helped with the rewrite of Chapter 18. Parts of Chapter 14 were derived from a paper by Hillyer and Silberschatz [1996]. Parts of Chapter 17 were derived from a paper by Levy and Silberschatz [1990]. Chapter 20 was derived from an unpublished manuscript by Stephen Tweedie. Chapter 21 was derived from an unpublished manuscript by Cliff Martin. Cliff Martin helped with updating the UNIX appendix to cover FreeBSD. Mike Shapiro reviewed the Solaris information and Jim Mauro answered several Solaris-related questions.

We thank the following people who reviewed this edition of the book: Rida Bazzi, Arizona State University; Roy Campbell, University of Illinois-Chicago; Gil Carrick, University of Texas at Arlington; Richard Guy, UCLA; Max Hailperin, Gustavus Adolphus College; Ahmed Kamel, North Dakota State University; Morty Kwestel, New Jersey Institute of Technology; Gustavo Rodriguez-Rivera, Purdue University; Carolyn J. C. Schable, Colorado State University; Thomas P. Skinner, Boston University; Yannis Smaragdakis, Geor-

gia Tech; Larry L. Wear, California State University, Chico; James M. Westall, Clemson University; and Yang Xiang, University of Massachusetts.

Our Acquisitions Editors, Bill Zobrist and Paul Crockett, provided expert guidance as we prepared this Edition. They were both assisted by Susanah Barr, who managed the many details of this project smoothly. Katherine Hepburn was our Marketing Manager. The Senior Production Editor was Ken Santor. The cover illustrator was Susan Cyr while the cover designer was Madelyn Lesure. Barbara Heaney was in charge of overseeing the copy-editing and Katie Habib copyedited the manuscript. The freelance proofreader was Katrina Avery; the freelance indexer was Rosemary Simpson. The Senior Illustration Coordinator was Anna Melhorn. Marilyn Turnamian helped generate figures and update the text, Instructors Manual, and slides.

Finally, we would like to add some personal notes. Avi would like to extend his gratitude to Krystyna Kwiecien, whose devoted care of his mother has given him the peace of mind he needed to focus on the writing of this book; Pete, would like to thank Harry Kasparian, and his other co-workers, who gave him the freedom to work on this project while doing his “real job”; Greg would like to acknowledge two significant achievements by his children during the period he worked on this text: Tom—age 5—learned to read, and Jay—age 2—learned to talk.

Abraham Silberschatz, Murray Hill, NJ, 2001

Peter Baer Galvin, Norton, MA, 2001

Greg Gagne, Salt Lake City, UT, 2001

CONTENTS

PART ONE ■ OVERVIEW

Chapter 1 Introduction

1.1 What Is an Operating System?	3	1.8 Handheld Systems	19
1.2 Mainframe Systems	7	1.9 Feature Migration	20
1.3 Desktop Systems	11	1.10 Computing Environments	21
1.4 Multiprocessor Systems	12	1.11 Summary	23
1.5 Distributed Systems	14	Exercises	24
1.6 Clustered Systems	16	Bibliographical Notes	25
1.7 Real-Time Systems	17		

Chapter 2 Computer-System Structures

2.1 Computer-System Operation	27	2.6 Network Structure	48
2.2 I/O Structure	30	2.7 Summary	51
2.3 Storage Structure	34	Exercises	52
2.4 Storage Hierarchy	38	Bibliographical Notes	54
2.5 Hardware Protection	42		

Chapter 3 Operating-System Structures

3.1 System Components	55	3.7 System Design and Implementation	85
3.2 Operating-System Services	61	3.8 System Generation	88
3.3 System Calls	63	3.9 Summary	89
3.4 System Programs	72	Exercises	90
3.5 System Structure	74	Bibliographical Notes	92
3.6 Virtual Machines	80		

PART TWO ■ PROCESS MANAGEMENT

Chapter 4 Processes

4.1 Process Concept	95	4.6 Communication in Client-Server Systems	117
4.2 Process Scheduling	99	4.7 Summary	126
4.3 Operations on Processes	103	Exercises	127
4.4 Cooperating Processes	107	Bibliographical Notes	128
4.5 Interprocess Communication	109		

Chapter 5 Threads

5.1 Overview	129	5.7 Linux Threads	144
5.2 Multithreading Models	132	5.8 Java Threads	145
5.3 Threading Issues	135	5.9 Summary	147
5.4 Pthreads	139	Exercises	147
5.5 Solaris 2 Threads	141	Bibliographical Notes	148
5.6 Window 2000 Threads	143		

Chapter 6 CPU Scheduling

6.1 Basic Concepts	151	6.6 Algorithm Evaluation	172
6.2 Scheduling Criteria	155	6.7 Process Scheduling Models	177
6.3 Scheduling Algorithms	157	6.8 Summary	184
6.4 Multiple-Processor Scheduling	169	Exercises	185
6.5 Real-Time Scheduling	170	Bibliographical Notes	187

Chapter 7 Process Synchronization

7.1 Background	189	7.7 Monitors	216
7.2 The Critical-Section Problem	191	7.8 OS Synchronization	223
7.3 Synchronization Hardware	197	7.9 Atomic Transactions	225
7.4 Semaphores	201	7.10 Summary	235
7.5 Classic Problems of Synchronization	206	Exercises	236
7.6 Critical Regions	211	Bibliographical Notes	240

Chapter 8 Deadlocks

8.1 System Model	243	8.6 Deadlock Detection	260
8.2 Deadlock Characterization	245	8.7 Recovery from Deadlock	264
8.3 Methods for Handling Deadlocks	248	8.8 Summary	266
8.4 Deadlock Prevention	250	Exercises	266
8.5 Deadlock Avoidance	253	Bibliographical Notes	270

PART THREE ■ STORAGE MANAGEMENT

Chapter 9 Memory Management

9.1 Background	273	9.6 Segmentation with Paging	309
9.2 Swapping	280	9.7 Summary	312
9.3 Contiguous Memory Allocation	283	Exercises	313
9.4 Paging	287	Bibliographical Notes	316
9.5 Segmentation	303		

Chapter 10 Virtual Memory

10.1 Background	317	10.7 Operating-System Examples	353
10.2 Demand Paging	320	10.8 Other Considerations	356
10.3 Process Creation	328	10.9 Summary	363
10.4 Page Replacement	330	Exercises	364
10.5 Allocation of Frames	344	Bibliographical Notes	369
10.6 Thrashing	348		

Chapter 11 File-System Interface

11.1 File Concept	371	11.6 Protection	402
11.2 Access Methods	379	11.7 Summary	406
11.3 Directory Structure	383	Exercises	407
11.4 File-System Mounting	393	Bibliographical Notes	409
11.5 File Sharing	395		

Chapter 12 File-System Implementation

12.1 File-System Structure	411	12.7 Recovery	437
12.2 File-System Implementation	413	12.8 Log-Structured File System	439
12.3 Directory Implementation	420	12.9 NFS	441
12.4 Allocation Methods	421	12.10 Summary	448
12.5 Free-Space Management	430	Exercises	449
12.6 Efficiency and Performance	433	Bibliographical Notes	451

PART FOUR ■ I/O SYSTEMS

Chapter 13 I/O Systems

13.1 Overview	455	13.6 STREAMS	481
13.2 I/O Hardware	456	13.7 Performance	483
13.3 Application I/O Interface	466	13.8 Summary	487
13.4 Kernel I/O Subsystem	472	Exercises	487
13.5 Transforming I/O to Hardware Operations	478	Bibliographical Notes	488

Chapter 14 Mass-Storage Structure

14.1 Disk Structure	491	14.7 Stable-Storage Implementation	514
14.2 Disk Scheduling	492	14.8 Tertiary-Storage Structure	516
14.3 Disk Management	498	14.9 Summary	526
14.4 Swap-Space Management	502	Exercises	528
14.5 RAID Structure	505	Bibliographical Notes	535
14.6 Disk Attachment	512		

PART FIVE ■ DISTRIBUTED SYSTEMS

Chapter 15 Distributed System Structures

15.1 Background	539	15.7 Design Issues	564
15.2 Topology	546	15.8 An Example: Networking	566
15.3 Network Types	548	15.9 Summary	568
15.4 Communication	551	Exercises	569
15.5 Communication Protocols	558	Bibliographical Notes	571
15.6 Robustness	562		

Chapter 16 Distributed File Systems

16.1 Background	573	16.6 An Example: AFS	586
16.2 Naming and Transparency	575	16.7 Summary	591
16.3 Remote File Access	579	Exercises	592
16.4 Stateful Versus Stateless Service	583	Bibliographical Notes	593
16.5 File Replication	585		

Chapter 17 Distributed Coordination

17.1 Event Ordering	595	17.6 Election Algorithms	618
17.2 Mutual Exclusion	598	17.7 Reaching Agreement	620
17.3 Atomicity	601	17.8 Summary	623
17.4 Concurrency Control	605	Exercises	624
17.5 Deadlock Handling	610	Bibliographical Notes	625

PART SIX ■ PROTECTION AND SECURITY

Chapter 18 Protection

18.1 Goals of Protection	629	18.6 Capability-Based Systems	645
18.2 Domain of Protection	630	18.7 Language-Based Protection	648
18.3 Access Matrix	636	18.8 Summary	654
18.4 Implementation of Access Matrix	640	Exercises	655
18.5 Revocation of Access Rights	643	Bibliographical Notes	656

Chapter 19 Security

19.1 The Security Problem	657
19.2 User Authentication	659
19.3 Program Threats	663
19.4 System Threats	666
19.5 Securing Systems and Facilities	671
19.6 Intrusion Detection	674
19.7 Cryptography	680
19.8 Computer-Security Classifications	686
19.9 An Example: Windows NT	687
19.10 Summary	689
Exercises	690
Bibliographical Notes	691

PART SEVEN ■ CASE STUDIES**Chapter 20 The Linux System**

20.1 History	695
20.2 Design Principles	700
20.3 Kernel Modules	703
20.4 Process Management	707
20.5 Scheduling	711
20.6 Memory Management	716
20.7 File Systems	724
20.8 Input and Output	729
20.9 Interprocess Communication	732
20.10 Network Structure	734
20.11 Security	737
20.12 Summary	739
Exercises	740
Bibliographical Notes	741

Chapter 21 Windows 2000

21.1 History	743
21.2 Design Principles	744
21.3 System Components	746
21.4 Environmental Subsystems	763
21.5 File System	766
21.6 Networking	774
21.7 Programmer Interface	780
21.8 Summary	787
Exercises	787
Bibliographical Notes	788

Chapter 22 Historical Perspective

22.1 Early Systems	789
22.2 Atlas	796
22.3 XDS-940	797
22.4 THE	798
22.5 RC 4000	799
22.6 CTSS	800
22.7 MULTICS	800
22.8 OS/360	801
22.9 Mach	803
22.10 Other Systems	804

Appendix A The FreeBSD System (contents online)

A.1 History	A807	A.7 File System	A834
A.2 Design Principles	A813	A.8 I/O System	A842
A.3 Programmer Interface	A815	A.9 Interprocess Communication	A846
A.4 User Interface	A823	A.10 Summary	A852
A.5 Process Management	A827	Exercises	A852
A.6 Memory Management	A831	Bibliographical Notes	A853

Appendix B The Mach System (contents online)

B.1 History	A855	B.7 Programmer Interface	A880
B.2 Design Principles	A857	B.8 Summary	A881
B.3 System Components	A858	Exercises	A882
B.4 Process Management	A862	Bibliographical Notes	A883
B.5 Interprocess Communication	A868	Credits	A885
B.6 Memory Management	A874		

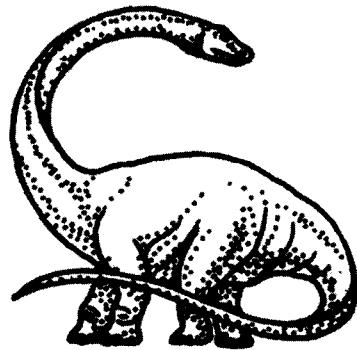
Appendix C The Nachos System (contents online)

C.1 Overview	A888	C.5 Conclusions	A900
C.2 Nachos Software Structure	A890	Bibliographical Notes	A901
C.3 Sample Assignments	A893	Credits	A902
C.4 Obtaining a Copy of Nachos	A898		

Bibliography 807**Credits 837****Index 839**

Part One

OVERVIEW



An *operating system* is a program that acts as an intermediary between the user of a computer and the computer hardware. The purpose of an operating system is to provide an environment in which a user can execute programs in a *convenient* and *efficient* manner.

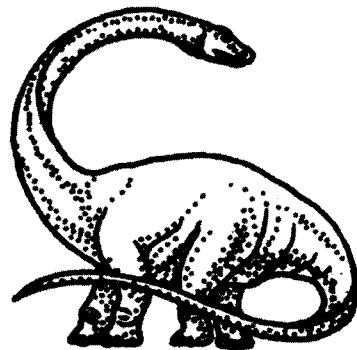
We trace the development of operating systems from the first hands-on systems, through multiprogrammed and time-shared systems, to current handheld and real-time systems. Understanding the evolution of operating systems gives us an appreciation for what an operating system does and how it does it.

The operating system must ensure the correct operation of the computer system. To prevent user programs from interfering with the proper operation of the system, the hardware must provide appropriate mechanisms. We describe the basic computer architecture that makes it possible to write a correct operating system.

The operating system provides certain services to programs and to the users of those programs in order to make their tasks easier. The services differ from one operating system to another, but we identify and explore some common classes of these services.

Chapter 1

INTRODUCTION



An **operating system** is a program that manages the computer hardware. It also provides a basis for application programs and acts as an intermediary between a user of a computer and the computer hardware. An amazing aspect of operating systems is how varied they are in accomplishing these tasks. Mainframe operating systems are designed primarily to optimize utilization of hardware. Personal computer (PC) operating systems support complex games, business applications, and everything in between. Handheld computer operating systems are designed to provide an environment in which a user can easily interface with the computer to execute programs. Thus, some operating systems are designed to be *convenient*, others to be *efficient*, and others some combination of the two.

To understand what operating systems are, we must first understand how they have developed. In this chapter, we trace the development of operating systems from the first hands-on systems through multiprogrammed and time-shared systems to PCs, and handheld computers. We also discuss operating system variations, such as parallel, real-time, and embedded systems. As we move through the various stages, we see how the components of operating systems evolved as natural solutions to problems in early computer systems.

1.1 ■ What Is an Operating System?

An operating system is an important part of almost every computer system. A computer system can be divided roughly into four components: the *hardware*, the *operating system*, the *application programs*, and the *users* (Figure 1.1).

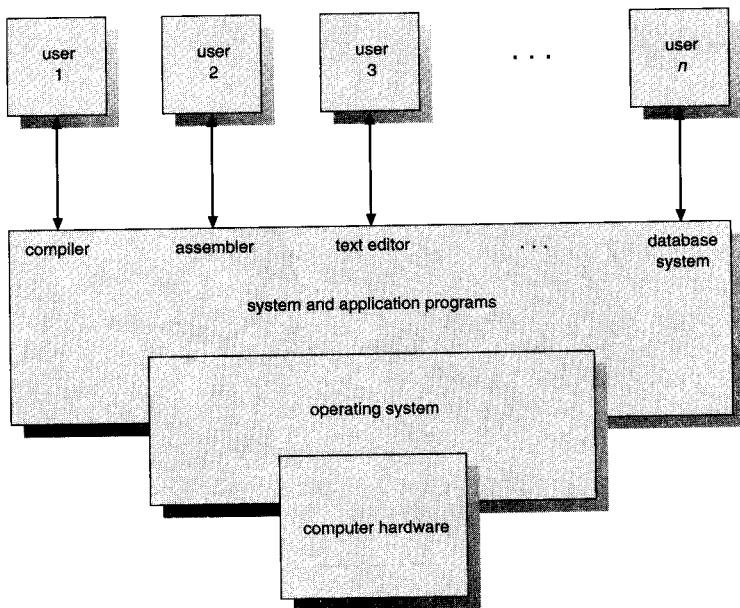


Figure 1.1 Abstract view of the components of a computer system.

The **hardware**—the **central processing unit (CPU)**, the **memory**, and the **input/output (I/O) devices**—provides the basic computing resources. The **application programs**—such as word processors, spreadsheets, compilers, and web browsers—define the ways in which these resources are used to solve the computing problems of the users. The operating system controls and coordinates the use of the hardware among the various application programs for the various users.

The components of a computer system are its hardware, software, and data. The operating system provides the means for the proper use of these resources in the operation of the computer system. An operating system is similar to a *government*. Like a government, it performs no useful function by itself. It simply provides an *environment* within which other programs can do useful work. Operating systems can be explored from two viewpoints: the user and the system.

1.1.1 User View

The user view of the computer varies by the interface being used. Most computer users sit in front of a PC, consisting of a monitor, keyboard, mouse, and system unit. Such a system is designed for one user to monopolize its resources, to maximize the work (or play) that the user is performing. In this case, the operating system is designed mostly for **ease of use**, with

some attention paid to performance, and none paid to resource utilization. Performance is important to the user, but it does not matter if most of the system is sitting idle, waiting for the slow I/O speed of the user.

Some users sit at a terminal connected to a **mainframe** or **minicomputer**. Other users are accessing the same computer through other terminals. These users share resources and may exchange information. The operating system is designed to maximize **resource utilization**—to assure that all available CPU time, memory, and I/O are used efficiently, and that no individual user takes more than her fair share.

Other users sit at **workstations**, connected to networks of other workstations and servers. These users have dedicated resources at their disposal, but they also share resources such as networking and servers—file, compute and print servers. Therefore, their operating system is designed to compromise between individual usability and resource utilization.

Recently, many varieties of handheld computers have come into fashion. These devices are mostly standalone, used singly by individual users. Some are connected to networks, either directly by wire or (more often) through wireless modems. Due to power and interface limitations they perform relatively few remote operations. The operating systems are designed mostly for individual usability, but performance per amount of battery life is important as well.

Some computers have little or no user view. For example, embedded computers in home devices and automobiles may have a numeric keypad, and may turn indicator lights on or off to show status, but mostly they and their operating systems are designed to run without user intervention.

1.1.2 System View

From the computer's point of view, the operating system is the program that is most intimate with the hardware. We can view an operating system as a **resource allocator**. A computer system has many resources—hardware and software—that may be required to solve a problem: CPU time, memory space, file-storage space, I/O devices, and so on. The operating system acts as the manager of these resources. Facing numerous and possibly conflicting requests for resources, the operating system must decide how to allocate them to specific programs and users so that it can operate the computer system efficiently and fairly.

A slightly different view of an operating system emphasizes the need to control the various I/O devices and user programs. An operating system is a **control program**. A **control program** manages the execution of user programs to prevent errors and improper use of the computer. It is especially concerned with the operation and control of I/O devices.

In general, however, we have no completely adequate definition of an operating system. Operating systems exist because they are a reasonable way to solve the problem of creating a usable computing system. The fundamental

goal of computer systems is to execute user programs and to make solving user problems easier. Toward this goal, computer hardware is constructed. Since bare hardware alone is not particularly easy to use, application programs are developed. These programs require certain common operations, such as those controlling the I/O devices. The common functions of controlling and allocating resources are then brought together into one piece of software: the operating system.

In addition, we have no universally accepted definition of what is part of the operating system. A simple viewpoint is that it includes everything a vendor ships when you order “the operating system.” The storage (memory, disks, and tapes) requirements and features included, however, vary greatly across systems. (The storage capacity of a system is measured in gigabytes. (A kilobyte or KB is 1,024 bytes, a megabyte or MB is $1,024^2$ bytes, and a gigabyte or GB is $1,024^3$ bytes, but computer manufacturers often round off these numbers and say that a megabyte is 1 million bytes, and a gigabyte is 1 billion bytes.)) Some systems take up less than 1 megabyte of space and lack even a full-screen editor, whereas others require hundreds of megabytes of space and are entirely based on graphical windowing systems. A more common definition is that the operating system is the one program running at all times on the computer (usually called the *kernel*), with all else being application programs. This last definition is the one that we generally follow. The matter of what constitutes an operating system is becoming important. In 1998, the United States Department of Justice filed suit against Microsoft, in essence claiming that Microsoft included too much functionality in its operating systems and thus prevented competition from application vendors.

1.1.3 System Goals

It is easier to define an operating system by what it *does* than by what it *is*, but even this can be tricky. The primary goal of some operating system is *convenience for the user*. Operating systems exist because they are supposed to make it easier to compute with them than without them. This view is particularly clear when you look at operating systems for small PCs.

The primary goal of other operating systems is *efficient* operation of the computer system. This is the case for large, shared, multiuser systems. These systems are expensive, so it is desirable to make them as efficient as possible. These two goals—convenience and efficiency—are sometimes contradictory. In the past, efficiency was often more important than convenience (Section 1.2.1). Thus, much of operating-system theory concentrates on optimal use of computing resources. Operating systems have also evolved over time. For example, UNIX started with a keyboard and printer as its interface, limiting how convenient it could be for the user. Over time, hardware changed, and UNIX was ported to new hardware with more user-friendly interfaces. Many **graphic**

user interfaces (GUIs) were added, allowing UNIX to be more convenient for users while still concentrating on efficiency.

The design of an operating system is a complex task. Designers face many tradeoffs in the design and implementation, and many people are involved not only in bringing an operating system to fruition, but also constantly revising and updating it. How well any given operating system meets its design goals is open to debate, and is subjective to the different users of the operating system.

To see what operating systems are and what they do, let us consider how they have developed over the past 45 years. By tracing that evolution, we can identify the common elements of operating systems, and see how and why these systems have developed as they have.

Operating systems and computer architecture have influenced each other a great deal. To facilitate the use of the hardware, researchers developed operating systems. Users of the operating systems then proposed changes in hardware design to simplify them. In this short historical review, notice how identification of operating-system problems led to the introduction of new hardware features.

1.2 ■ Mainframe Systems

Mainframe computer systems were the first computers used to tackle many commercial and scientific applications. In this section, we trace the growth of mainframe systems from simple **batch systems**, where the computer runs one—and only one—application, to **time-shared systems**, which allow for user interaction with the computer system.

1.2.1 Batch Systems

Early computers were physically enormous machines run from a console. The common input devices were card readers and tape drives. The common output devices were line printers, tape drives, and card punches. The user did not interact directly with the computer systems. Rather, the user prepared a job—which consisted of the program, the data, and some control information about the nature of the job (control cards)—and submitted it to the computer operator. The job was usually in the form of punch cards. At some later time (after minutes, hours, or days), the output appeared. The output consisted of the result of the program, as well as a dump of the final memory and register contents for debugging.

The operating system in these early computers was fairly simple. Its major task was to transfer control automatically from one job to the next. The operating system was always resident in memory (Figure 1.2).

To speed up processing, operators **batched** together jobs with similar needs and ran them through the computer as a group. Thus, the programmers would

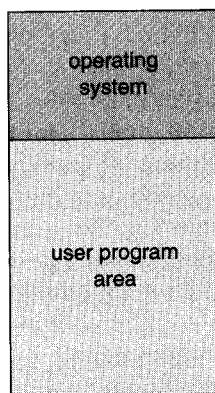


Figure 1.2 Memory layout for a simple batch system.

leave their programs with the operator. The operator would sort programs into batches with similar requirements and, as the computer became available, would run each batch. The output from each job would be sent back to the appropriate programmer.

In this execution environment, the CPU is often idle, because the speeds of the mechanical I/O devices are intrinsically slower than are those of electronic devices. Even a slow CPU works in the microsecond range, with thousands of instructions executed per second. A fast card reader, on the other hand, might read 1200 cards per minute (or 20 cards per second). Thus, the difference in speed between the CPU and its I/O devices may be three orders of magnitude or more. Over time, of course, improvements in technology and the introduction of disks resulted in faster I/O devices. However, CPU speeds increased to an even greater extent, so the problem was not only unresolved, but exacerbated.

The introduction of disk technology allowed the operating system to keep all jobs on a disk, rather than in a serial card reader. With direct access to several jobs, the operating system could perform **job scheduling**, to use resources and perform tasks efficiently. We discuss a few important aspects of job and CPU scheduling here; we discuss them in detail in Chapter 6.

1.2.2 Multiprogrammed Systems

The most important aspect of job scheduling is the ability to multiprogram. A single user cannot, in general, keep either the CPU or the I/O devices busy at all times. **Multiprogramming** increases CPU utilization by organizing jobs so that the CPU always has one to execute.

The idea is as follows: The operating system keeps several jobs in memory simultaneously (Figure 1.3). This set of jobs is a subset of the jobs kept in the job pool—since the number of jobs that can be kept simultaneously in memory is usually much smaller than the number of jobs that can be in the job pool. The

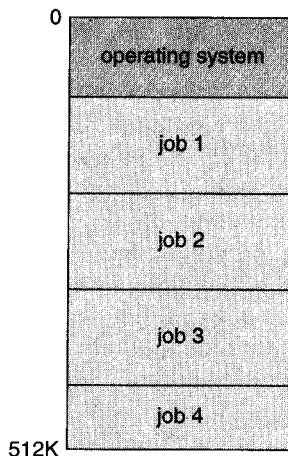


Figure 1.3 Memory layout for a multiprogramming system.

operating system picks and begins to execute one of the jobs in the memory. Eventually, the job may have to wait for some task, such as an I/O operation, to complete. In a non-multiprogrammed system, the CPU would sit idle. In a multiprogramming system, the operating system simply switches to, and executes, another job. When *that* job needs to wait, the CPU is switched to *another* job, and so on. Eventually, the first job finishes waiting and gets the CPU back. As long as at least one job needs to execute, the CPU is never idle.

This idea is common in other life situations. A lawyer does not work for only one client at a time. While one case is waiting to go to trial or have papers typed, the lawyer can work on another case. If she has enough clients, the lawyer will never be idle for lack of work. (Idle lawyers tend to become politicians, so there is a certain social value in keeping lawyers busy.)

Multiprogramming is the first instance where the operating system must make decisions for the users. Multiprogrammed operating systems are therefore fairly sophisticated. All the jobs that enter the system are kept in the job pool. This pool consists of all processes residing on disk awaiting allocation of main memory. If several jobs are ready to be brought into memory, and if there is not enough room for all of them, then the system must choose among them. Making this decision is *job scheduling*, which is discussed in Chapter 6. When the operating system selects a job from the job pool, it loads that job into memory for execution. Having several programs in memory at the same time requires some form of memory management, which is covered in Chapters 9 and 10. In addition, if several jobs are ready to run at the same time, the system must choose among them. Making this decision is *CPU scheduling*, which is discussed in Chapter 6. Finally, multiple jobs running concurrently require that their ability to affect one another be limited in all phases of the operating

system, including process scheduling, disk storage, and memory management. These considerations are discussed throughout the text.

1.2.3 Time-Sharing Systems

Multiprogrammed, batched systems provided an environment where the various system resources (for example, CPU, memory, peripheral devices) were utilized effectively, but it did not provide for user interaction with the computer system. **Time sharing** (or **multitasking**) is a logical extension of multiprogramming. The CPU executes multiple jobs by switching among them, but the switches occur so frequently that the users can interact with each program while it is running.

An **interactive** (or **hands-on**) **computer system** provides direct communication between the user and the system. The user gives instructions to the operating system or to a program directly, using a keyboard or a mouse, and waits for immediate results. Accordingly, the **response time** should be short—typically within 1 second or so.

A time-shared operating system allows many users to share the computer simultaneously. Since each action or command in a time-shared system tends to be short, only a little CPU time is needed for each user. As the system switches rapidly from one user to the next, each user is given the impression that the entire computer system is dedicated to her use, even though it is being shared among many users.

A time-shared operating system uses CPU scheduling and multiprogramming to provide each user with a small portion of a time-shared computer. Each user has at least one separate program in memory. A program loaded into memory and executing is commonly referred to as a **process**. When a process executes, it typically executes for only a short time before it either finishes or needs to perform I/O. I/O may be interactive; that is, output is to a display for the user and input is from a user keyboard, mouse, or other device. Since interactive I/O typically runs at “people speeds,” it may take a long time to complete. Input, for example, may be bounded by the user’s typing speed; seven characters per second is fast for people, but incredibly slow for computers. Rather than let the CPU sit idle when this interactive input takes place, the operating system will rapidly switch the CPU to the program of some other user.

Time-sharing operating systems are even more complex than multiprogrammed operating systems. In both, several jobs must be kept simultaneously in memory, so the system must have memory management and protection (Chapter 9). To obtain a reasonable response time, jobs may have to be swapped in and out of main memory to the disk that now serves as a backing store for main memory. A common method for achieving this goal is **virtual memory**, which is a technique that allows the execution of a job that may not be completely in memory (Chapter 10). The main advantage of the virtual-memory

scheme is that programs can be larger than **physical memory**. Further, it abstracts main memory into a large, uniform array of storage, separating **logical memory** as viewed by the user from physical memory. This arrangement frees programmers from concern over memory-storage limitations.

Time-sharing systems must also provide a file system (Chapters 11 and 12). The file system resides on a collection of disks; hence, disk management must be provided (Chapter 14). Also, time-sharing systems provide a mechanism for concurrent execution, which requires sophisticated CPU-scheduling schemes (Chapter 6). To ensure orderly execution, the system must provide mechanisms for job synchronization and communication (Chapter 7), and it may ensure that jobs do not get stuck in a deadlock, forever waiting for one another (Chapter 8).

The idea of time sharing was demonstrated as early as 1960, but since time-shared systems are difficult and expensive to build, they did not become common until the early 1970s. Although some batch processing is still done, most systems today are time sharing. Accordingly, multiprogramming and time sharing are the central themes of modern operating systems, and they are the central themes of this book.

1.3 ■ Desktop Systems

Personal computers PCs appeared in the 1970s. During their first decade, the CPUs in PCs lacked the features needed to protect an operating system from user programs. PC operating systems therefore were neither multiuser nor multitasking. However, the goals of these operating systems have changed with time; instead of maximizing CPU and peripheral utilization, the systems opt for maximizing user convenience and responsiveness. These systems include PCs running Microsoft Windows and the Apple Macintosh. The MS-DOS operating system from Microsoft has been superseded by multiple flavors of Microsoft Windows, and IBM has upgraded MS-DOS to the OS/2 multitasking system. The Apple Macintosh operating system has been ported to more advanced hardware, and now includes new features, such as virtual memory and multitasking. With the release of MacOS X, the core of the operating system is now based on Mach and FreeBSD UNIX for scalability, performance, and features, but it retains the same rich GUI. Linux, a UNIX-like operating system available for PCs, has also become popular recently.

Operating systems for these computers have benefited in several ways from the development of operating systems for mainframes. Microcomputers were immediately able to adopt some of the technology developed for larger operating systems. On the other hand, the hardware costs for microcomputers are sufficiently low that individuals have sole use of the computer, and CPU utilization is no longer a prime concern. Thus, some of the design decisions made in operating systems for mainframes may not be appropriate for smaller systems.

Other design decisions still apply. For example, file protection was, at first, not necessary on a personal machine. However, these computers are now often tied into other computers over local-area networks or other Internet connections. When other computers and other users can access the files on a PC, file protection again becomes a necessary feature of the operating system. The lack of such protection has made it easy for malicious programs to destroy data on systems such as MS-DOS and the Macintosh operating system. These programs may be self-replicating, and may spread rapidly via **worm** or **virus** mechanisms and disrupt entire companies or even worldwide networks. Advanced time-sharing features such as protected memory and file permissions are not enough, on their own, to safeguard a system from attack. Recent security breaches have shown that time and again. These topics are discussed in Chapters 18 and 19.

1.4 ■ Multiprocessor Systems

Most systems to date are single-processor systems; that is, they have only one main CPU. However, **multiprocessor systems** (also known as **parallel systems** or **tightly coupled systems**) are growing in importance. Such systems have more than one processor in close communication, sharing the computer bus, the clock, and sometimes memory and peripheral devices.

Multiprocessor systems have three main advantages.

1. **Increased throughput.** By increasing the number of processors, we hope to get more work done in less time. The speed-up ratio with N processors is not N ; rather, it is less than N . When multiple processors cooperate on a task, a certain amount of overhead is incurred in keeping all the parts working correctly. This overhead, plus contention for shared resources, lowers the expected gain from additional processors. Similarly, a group of N programmers working closely together does not result in N times the amount of work being accomplished.
2. **Economy of scale.** Multiprocessor systems can save more money than multiple single-processor systems, because they can share peripherals, mass storage, and power supplies. If several programs operate on the same set of data, it is cheaper to store those data on one disk and to have all the processors share them, than to have many computers with local disks and many copies of the data.
3. **Increased reliability.** If functions can be distributed properly among several processors, then the failure of one processor will not halt the system, only slow it down. If we have ten processors and one fails, then each of the remaining nine processors must pick up a share of the work of the failed processor. Thus, the entire system runs only 10 percent slower, rather than failing altogether. This ability to continue providing service proportional

to the level of surviving hardware is called **graceful degradation**. Systems designed for graceful degradation are also called **fault tolerant**.

Continued operation in the presence of failures requires a mechanism to allow the failure to be detected, diagnosed, and, if possible, corrected. The Tandem system uses both hardware and software duplication to ensure continued operation despite faults. The system consists of two identical processors, each with its own local memory. The processors are connected by a bus. One processor is the primary and the other is the backup. Two copies are kept of each process: one on the primary processor and the other on the backup. At fixed checkpoints in the execution of the system, the state information of each job—including a copy of the memory image—is copied from the primary machine to the backup. If a failure is detected, the backup copy is activated and is restarted from the most recent checkpoint. This solution is expensive, since it involves considerable hardware duplication.

The most common multiple-processor systems now use **symmetric multiprocessing (SMP)**, in which each processor runs an identical copy of the operating system, and these copies communicate with one another as needed. Some systems use **asymmetric multiprocessing**, in which each processor is assigned a specific task. A master processor controls the system; the other processors either look to the master for instruction or have predefined tasks. This scheme defines a master–slave relationship. The master processor schedules and allocates work to the slave processors.

SMP means that all processors are peers; no master–slave relationship exists between processors. Each processor concurrently runs a copy of the operating system. Figure 1.4 illustrates a typical SMP architecture. An example of the SMP system is Encore’s version of UNIX for the Multimax computer. This computer can be configured such that it employs dozens of processors, all running copies of UNIX. The benefit of this model is that many processes can run simultaneously— N processes can run if there are N CPUs—without causing a significant deterioration of performance. However, we must carefully control I/O to ensure that the data reach the appropriate processor. Also, since the CPUs are separate, one may be sitting idle while another is overloaded, resulting in inefficiencies. These inefficiencies can be avoided if the processors share certain data structures. A multiprocessor system of this form will allow

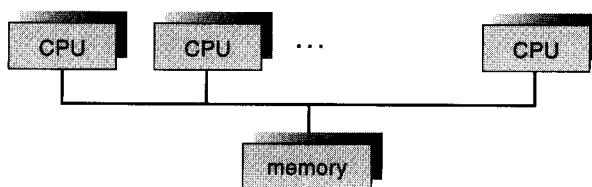


Figure 1.4 Symmetric multiprocessing architecture.

processes and resources—such as memory—to be shared dynamically among the various processors, and can lower the variance among the processors. Such a system must be written carefully, as we shall see in Chapter 7. Virtually all modern operating systems—including Windows NT, Solaris, Digital UNIX, OS/2, and Linux—now provide support for SMP.

The difference between symmetric and asymmetric multiprocessing may be the result of either hardware or software. Special hardware can differentiate the multiple processors, or the software can be written to allow only one master and multiple slaves. For instance, Sun's operating system SunOS Version 4 provides asymmetric multiprocessing, whereas Version 5 (Solaris 2) is symmetric on the same hardware.

As microprocessors become less expensive and more powerful, additional operating-system functions are off-loaded to slave processors (or **back-ends**). For example, it is fairly easy to add a microprocessor with its own memory to manage a disk system. The microprocessor could receive a sequence of requests from the main CPU and implement its own disk queue and scheduling algorithm. This arrangement relieves the main CPU of the overhead of disk scheduling. PCs contain a microprocessor in the keyboard to convert the keystrokes into codes to be sent to the CPU. In fact, this use of microprocessors has become so common that it is no longer considered multiprocessing.

1.5 ■ Distributed Systems

A **network**, in the simplest terms, is a communication path between two or more systems. Distributed systems depend on networking for their functionality. By being able to communicate, distributed systems are able to share computational tasks, and provide a rich set of features to users.

Networks vary by the protocols used, the distances between nodes, and the transport media. TCP/IP is the most common network protocol, although ATM and other protocols are in widespread use. Likewise, operating-system support of protocols varies. Most operating systems support TCP/IP, including the Windows and UNIX operating systems. Some systems support proprietary protocols to suit their needs. To an operating system, a network protocol simply needs an interface device—a network adapter, for example—with a device driver to manage it, and software to package data in the communications protocol to send it and to unpackage it to receive it. These concepts are discussed throughout the book.

Networks are typecast based on the distances between their nodes. A **local-area network (LAN)**, exists within a room, a floor, or a building. A **wide-area network (WAN)**, usually exists between buildings, cities, or countries. A global company may have a WAN to connect its offices, worldwide. These networks could run one protocol or several protocols. The continuing advent of new technologies brings about new forms of networks. For exam-

ple, a **metropolitan-area network (MAN)**, could link buildings within a city. BlueTooth devices communicate over a short distance of several feet, in essence creating a **small-area network**.

The media to carry networks are equally varied. They include copper wires, fiber strands, and wireless transmissions between satellites, microwave dishes, and radios. When computing devices are connected to cellular phones, they create a network. Even very short-range infrared communication can be used for networking. At a rudimentary level, whenever computers communicate they use or create a network. These networks also vary by their performance and reliability.

1.5.1 Client-Server Systems

As PCs have become faster, more powerful, and cheaper, designers have shifted away from the centralized system architecture. Terminals connected to centralized systems are now being supplanted by PCs. Correspondingly, user-interface functionality that used to be handled directly by the centralized systems is increasingly being handled by the PCs. As a result, centralized systems today act as **server systems** to satisfy requests generated by **client systems**. The general structure of a client–server system is depicted in Figure 1.5.

Server systems can be broadly categorized as compute servers and file servers.

- **Compute-server systems** provide an interface to which clients can send requests to perform an action, in response to which they execute the action and send back results to the client.
- **File-server systems** provide a file-system interface where clients can create, update, read, and delete files.

1.5.2 Peer-to-Peer Systems

The growth of computer networks—especially the Internet and World Wide Web (WWW)—has had a profound influence on the recent development of operating systems. When PCs were introduced in the 1970s, they were designed

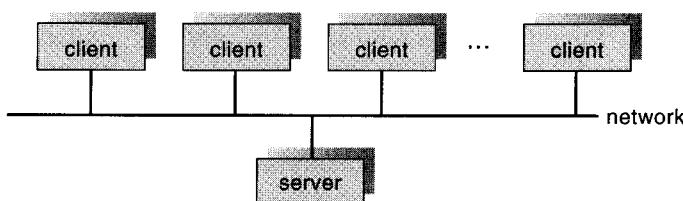


Figure 1.5 General structure of a client-server system.

for “personal” use and were generally considered standalone computers. With the beginning of widespread public use of the Internet in the 1980s for electronic mail, ftp, and gopher, many PCs became connected to computer networks. With the introduction of the Web in the mid-1990s, network connectivity became an essential component of a computer system.

Virtually all modern PCs and workstations are capable of running a web browser for accessing **hypertext** documents on the Web. Operating systems (such as Windows, OS/2, MacOS, and UNIX) now also include the system software (such as TCP/IP and PPP) that enables a computer to access the Internet via a local-area network or telephone connection. Several include the web browser itself, as well as electronic mail, remote login, and file-transfer clients and servers.

In contrast to the tightly coupled systems discussed in Section 1.4, the computer networks used in these applications consist of a collection of processors that do not share memory or a clock. Instead, each processor has its own local memory. The processors communicate with one another through various communication lines, such as high-speed buses or telephone lines. These systems are usually referred to as **loosely coupled systems** (or **distributed systems**).

Some operating systems have taken the concept of networks and distributed systems further than the notion of providing network connectivity. A **network operating system** is an operating system that provides features such as file sharing across the network, and that includes a communication scheme that allows different processes on different computers to exchange messages. A computer running a network operating system acts autonomously from all other computers on the network, although it is aware of the network and is able to communicate with other networked computers. A distributed operating system is a less autonomous environment: The different operating systems communicate closely enough to provide the illusion that only a single operating system controls the network. We cover computer networks and distributed systems in Chapters 15 through 17.

1.6 ■ Clustered Systems

Like parallel systems, **clustered systems** gather together multiple CPUs to accomplish computational work. Clustered systems differ from parallel systems, however, in that they are composed of two or more individual systems coupled together. The definition of the term *clustered* is not concrete; many commercial packages wrestle with what a clustered system is, and why one form is better than another. The generally accepted definition is that clustered computers share storage and are closely linked via LAN networking.

Clustering is usually performed to provide **high availability**. A layer of cluster software runs on the cluster nodes. Each node can monitor one or more of the others (over the LAN). If the monitored machine fails, the monitoring

machine can take ownership of its storage, and restart the application(s) that were running on the failed machine. The failed machine can remain down, but the users and clients of the application would only see a brief interruption of service.

In **asymmetric clustering**, one machine is in **hot standby mode** while the other is running the applications. The hot standby host (machine) does nothing but monitor the active server. If that server fails, the hot standby host becomes the active server. In **symmetric mode**, two or more hosts are running applications, and they are monitoring each other. This mode is obviously more efficient, as it uses all of the available hardware. It does require that more than one application be available to run.

Other forms of clusters include parallel clusters and clustering over a WAN. Parallel clusters allow multiple hosts to access the same data on the shared storage. Because most operating systems lack support for this simultaneous data access by multiple hosts, parallel clusters are usually accomplished by special versions of software and special releases of applications. For example, Oracle Parallel Server is a version of Oracle's database that has been designed to run on parallel clusters. Each machine runs Oracle, and a layer of software tracks access to the shared disk. Each machine has full access to all data in the database.

In spite of improvements in distributed computing, most systems do not offer general-purpose distributed file systems. Therefore, most clusters do not allow shared access to data on the disk. For this, distributed file systems must provide access control and locking to the files to ensure no conflicting operations occur. This type of service is commonly known as a **distributed lock manager (DLM)**. Work is ongoing for general-purpose distributed file systems, with vendors like Sun Microsystems announcing roadmaps for delivery of a DLM within the operating system.

Cluster technology is rapidly changing. Cluster directions include global clusters, in which the machines could be anywhere in the world (or anywhere a WAN reaches). Such projects are still the subject of research and development.

Clustered system use and features should expand greatly as **storage-area networks (SANs)**, as described in Section 14.6.3, become prevalent. SANs allow easy attachment of multiple hosts to multiple storage units. Current clusters are usually limited to two or four hosts due to the complexity of connecting the hosts to shared storage.

1.7 ■ Real-Time Systems

Another form of a special-purpose operating system is the **real-time system**. A real-time system is used when rigid time requirements have been placed on the operation of a processor or the flow of data; thus, it is often used as a control device in a dedicated application. Sensors bring data to the computer. The com-

puter must analyze the data and possibly adjust controls to modify the sensor inputs. Systems that control scientific experiments, medical imaging systems, industrial control systems, and certain display systems are real-time systems. Some automobile-engine fuel-injection systems, home-appliance controllers, and weapon systems are also real-time systems.

A real-time system has well-defined, fixed time constraints. Processing *must* be done within the defined constraints, or the system will fail. For instance, it would not do for a robot arm to be instructed to halt *after* it had smashed into the car it was building. A real-time system functions correctly only if it returns the correct result within its time constraints. Contrast this requirement to a time-sharing system, where it is desirable (but not mandatory) to respond quickly, or to a batch system, which may have no time constraints at all.

Real-time systems come in two flavors: hard and soft. A **hard real-time system** guarantees that critical tasks be completed on time. This goal requires that all delays in the system be bounded, from the retrieval of stored data to the time that it takes the operating system to finish any request made of it. Such time constraints dictate the facilities that are available in hard real-time systems. Secondary storage of any sort is usually limited or missing, with data instead being stored in short-term memory or in read-only memory (ROM). ROM is located on nonvolatile storage devices that retain their contents even in the case of electric outage; most other types of memory are volatile. Most advanced operating-system features are absent too, since they tend to separate the user from the hardware, and that separation results in uncertainty about the amount of time an operation will take. For instance, virtual memory (Chapter 10) is almost never found on real-time systems. Therefore, hard real-time systems conflict with the operation of time-sharing systems, and the two cannot be mixed. Since none of the existing general-purpose operating systems support hard real-time functionality, we do not concern ourselves with this type of system in this text.

A less restrictive type of real-time system is a **soft real-time system**, where a critical real-time task gets priority over other tasks, and retains that priority until it completes. As in hard real-time systems, the operating-system kernel delays need to be bounded: A real-time task cannot be kept waiting indefinitely for the kernel to run it. Soft real time is an achievable goal that can be mixed with other types of systems. Soft real-time systems, however, have more limited utility than hard real-time systems. Given their lack of deadline support, they are risky to use for industrial control and robotics. They are useful, however in several areas, including multimedia, virtual reality, and advanced scientific projects—such as undersea exploration and planetary rovers. These systems need advanced operating-system features that cannot be supported by hard real-time systems. Because of the expanded uses for soft real-time functionality, it is finding its way into most current operating systems, including major versions of UNIX.

In Chapter 6, we consider the scheduling facility needed to implement soft real-time functionality in an operating system. In Chapter 10, we describe the design of memory management for real-time computing. Finally, in Chapter 21, we describe the real-time components of the Windows 2000 operating system.

1.8 ■ Handheld Systems

Handheld systems include **personal digital assistants (PDAs)**, such as *Palm-Pilots* or cellular telephones with connectivity to a network such as the Internet. Developers of handheld systems and applications face many challenges, most of which are due to the limited size of such devices. For example, a PDA is typically about 5 inches in height and 3 inches in width, and it weighs less than one-half pound. Due to this limited size, most handheld devices have a small amount of memory, include slow processors, and feature small display screens. We will take a look now at each of these limitations.

Many handheld devices have between 512 KB and 8 MB of memory. (Contrast this with a typical PC or workstation, which may have several hundred megabytes of memory!) As a result, the operating system and applications must manage memory efficiently. This includes returning all allocated memory back to the memory manager once the memory is no longer being used. In Chapter 10 we will explore virtual memory, which allows developers to write programs that behave as if the system has more memory than may be physically available. Currently, many handheld devices do not use virtual memory techniques, thus forcing program developers to work within the confines of limited physical memory.

A second issue of concern to developers of handheld devices is the speed of the processor used in the device. Processors for most handheld devices often run at a fraction of the speed of a processor in a PC. Faster processors require more power. To include a faster processor in a handheld device would require a larger battery that would have to be replaced (or recharged) more frequently. To minimize the size of most handheld devices, smaller, slower processors which consume less power are typically used. Therefore, the operating system and applications must be designed not to tax the processor.

The last issue confronting program designers for handheld devices is the small display screens typically available. Whereas a monitor for a home computer may measure up to 21 inches, the display for a handheld device is often no more than 3 inches square. Familiar tasks, such as reading e-mail or browsing web pages, must be condensed onto smaller displays. One approach for displaying the content in web pages is **web clipping**, where only a small subset of a web page is delivered and displayed on the handheld device.

Some handheld devices may use wireless technology, such as BlueTooth (Section 1.5), allowing remote access to e-mail and web browsing. Cellular telephones with connectivity to the Internet fall into this category. However,

many PDAs currently do not provide wireless access. To download data to these devices, typically one first downloads the data to a PC or workstation, and then downloads the data to the PDA. Some PDAs allow data to be directly copied from one device to another using an infrared link. Generally, the limitations in the functionality of PDAs are balanced by their convenience and portability. Their use continues to expand as network connections become more available and other options, such as cameras and MP3 players, expand their utility.

1.9 ■ Feature Migration

Overall, an examination of operating systems for mainframes and microcomputers shows that features once available only on mainframes have been adopted by microcomputers. The same concepts are appropriate for the various classes of computers: mainframes, minicomputers, microcomputers, and handhelds. Many of the concepts depicted in Figure 1.6 will be covered later in this book. However, to start understanding modern operating systems, you need to realize the theme of feature migration and to recognize the long history of many operating-system features.

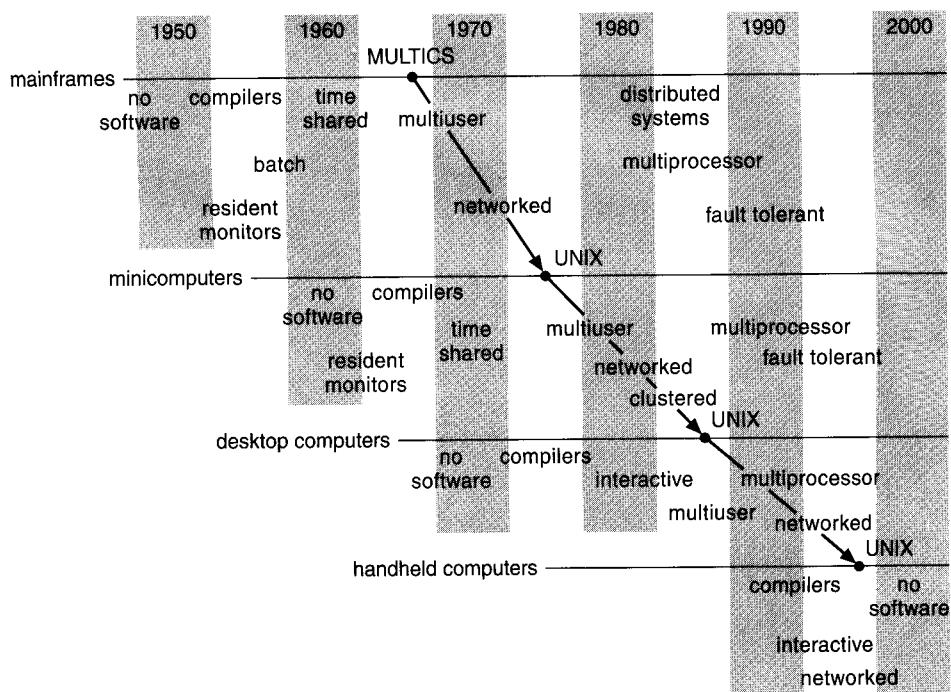


Figure 1.6 Migration of operating-system concepts and features.

A good example of this movement occurred with the MULTIplexed Information and Computing Services (MULTICS) operating system. MULTICS was developed from 1965 to 1970 at the Massachusetts Institute of Technology (MIT) as a computing **utility**. It ran on a large, complex mainframe computer (the GE 645). Many of the ideas that were developed for MULTICS were subsequently used at Bell Laboratories (one of the original partners in the development of MULTICS) in the design of UNIX. The UNIX operating system was designed circa 1970 for a PDP-11 minicomputer. Around 1980, the features of UNIX became the basis for UNIX-like operating systems on microcomputer systems, and they are being included in more recent operating systems such as Microsoft Windows NT, IBM OS/2, and the Macintosh operating system. Thus, the features developed for a large mainframe system have moved to microcomputers over time.

At the same time as features of large operating systems were being scaled down to fit PCs, more powerful, faster, and more sophisticated hardware systems were being developed. The **personal workstation** is a large PC—for example, the Sun SPARCstation, the HP/Apollo, the IBM RS/6000, and the Intel Pentium class system running Windows NT or a UNIX derivative. Many universities and businesses have large numbers of workstations tied together with local-area networks. As PCs gain more sophisticated hardware and software, the line dividing the two categories—mainframes and microcomputers—is blurring.

1.10 ■ Computing Environments

Now that we have traced the development of operating systems from the first hands-on systems through multiprogrammed and time-shared systems to PCs and handheld computers, we can give a brief overview of how such systems are used in a variety of computing environment settings.

1.10.1 Traditional Computing

As computing matures, the lines among many of the traditional computing environments are blurring. Consider the “typical office environment.” Just a few years ago, this environment consisted of PCs connected to a network, with servers providing file and print service. Remote access was awkward, and portability was achieved by laptop computers carrying some of the user’s workspace. Terminals attached to mainframes were prevalent at many companies as well, with even fewer remote access and portability options.

The current trend is toward more ways to access these environments. Web technologies are stretching the boundaries of traditional computing. Companies implement **portals** which provide web accessibility to their internal servers. **Network computers** are essentially terminals that understand web-

based computing. Handheld computers can synchronize with PCs to allow very portable use of company information. They can also connect to **wireless networks** to use the company's web portal (as well as the myriad other web resources).

At home, most users had a single computer with a slow modem connection to the office, the Internet, or both. Network connection speeds once attainable only at great cost are now available at low cost, allowing more access to more data at a company or from the Web. Those fast data connections are allowing home computers to serve up web pages and to contain their own networks with printers, client PCs, and servers. Some homes even have **firewalls** to protect these home environments from security breaches. Those firewalls cost thousands of dollars a few years ago, and did not even exist a decade ago.

1.10.2 Web-Based Computing

The Web has become ubiquitous, leading to more access by a wider variety of devices than was dreamt about a few years ago. PCs are still the most prevalent access devices, with workstations (high-end graphics-oriented PCs), handheld PDAs, and even cell phones also providing access.

Web computing has increased the emphasis on networking. Devices that were not previously networked now have wired or wireless access. Devices that were networked now have faster network connectivity, either by improved networking technology, optimized network implementation code, or both.

The implementation of web-based computing has given rise to new categories of devices, such as **load balancers** which distribute network connections among a pool of similar servers. Operating systems like Windows 95, which acted as web clients, have evolved into Windows ME and Windows 2000, which can act as web servers as well as clients. Generally, the Web has increased the complexity of devices as their users require them to be web-enabled.

1.10.3 Embedded Computing

Embedded computers are the most prevalent form of computers in existence. They run embedded real-time operating systems. These devices are found everywhere, from car engines and manufacturing robots to VCRs and microwave ovens. They tend to have very specific tasks. The systems they run on are usually primitive, lacking advanced features, such as virtual memory, and even disks. Thus, the operating systems provide limited features. They usually have little or no user interface, preferring to spend their time monitoring and managing hardware devices, such as automobile engines and robotic arms.

As an example, consider the aforementioned firewalls and load balancers. Some are general-purpose computers, running standard operating systems—such as UNIX—with special-purpose applications loaded to implement the

functionality. Others are hardware devices with a special-purpose operating system embedded within, providing just the functionality desired.

The use of embedded systems continues to expand. The power of those devices, both as standalone units and as members of networks and the Web, is sure to increase as well. Entire houses can be computerized, so that a central computer—either a general-purpose computer or an embedded system—can control heating and lighting, alarm systems, and even coffee makers. Web access can let a home-owner tell the house to heat up before he arrives home. Someday, the refrigerator may call the grocery store when it notices the milk is gone.

1.11 ■ Summary

Operating systems have been developed over the past 45 years for two main purposes. First, the operating system attempts to schedule computational activities to ensure good performance of the computing system. Second, it provides a convenient environment for the development and execution of programs. Initially, computer systems were used from the front console. Software such as assemblers, loaders, linkers, and compilers improved the convenience of programming the system, but also required substantial set-up time. To reduce the set-up time, facilities hired operators and batched similar jobs.

Batch systems allowed automatic job sequencing by a resident operating system and greatly improved the overall utilization of the computer. The computer no longer had to wait for human operation. CPU utilization was still low, however, because of the slow speed of the I/O devices relative to that of the CPU. Off-line operation of slow devices provided a means to use multiple reader-to-tape and tape-to-printer systems for one CPU.

To improve the overall performance of the computer system, developers introduced the concept of multiprogramming, so that several jobs could be kept in memory at one time. The CPU is switched back and forth among them to increase CPU utilization and to decrease the total time needed to execute the jobs.

Multiprogramming also allows time sharing. Time-shared operating systems allow many users (from one to several hundred) to use a computer system interactively at the same time.

PCs are microcomputers; they are considerably smaller and less expensive than mainframe systems. Operating systems for these computers have benefited from the development of operating systems for mainframes in several ways. However, since an individual has sole use of the computer, CPU utilization is no longer a prime concern. Hence, some of the design decisions made for mainframe operating systems may not be appropriate for these smaller systems. Other design decisions, such as those for security, are appropriate for both small and large systems, as PCs can now be connected to other computers and users through networks and the Web.

Parallel systems have more than one CPU in close communication; the CPUs share the computer bus, and sometimes share memory and peripheral devices. Such systems can provide increased throughput and enhanced reliability. Distributed systems allow sharing of resources on geographically dispersed hosts. Clustered systems allow multiple machines to perform computations on data contained on shared storage, and let computing continue in the case of failure of some subset of cluster members.

A hard real-time system is often used as a control device in a dedicated application. A hard real-time operating system has well-defined, fixed time constraints. Processing *must* be done within the defined constraints, or the system will fail. Soft real-time systems have less stringent timing constraints, and do not support deadline scheduling.

Recently, the influence of the Internet and the World Wide Web has encouraged the development of modern operating systems that include web browsers and networking and communication software as integral features.

We have shown the logical progression of operating-system development, driven by inclusion of features in the CPU hardware needed for advanced functionality. This trend can be seen today in the evolution of PCs, with inexpensive hardware being improved sufficiently to allow, in turn, improved characteristics.

■ Exercises

- 1.1 What are the three main purposes of an operating system?
- 1.2 List the four steps needed to run a program on a completely dedicated machine.
- 1.3 What is the main advantage of multiprogramming?
- 1.4 What are the main differences between operating systems for mainframe computers and PCs?
- 1.5 In a multiprogramming and time-sharing environment, several users share the system simultaneously. This situation can result in various security problems.
 - a. What are two such problems?
 - b. Can we ensure the same degree of security in a time-shared machine as we have in a dedicated machine? Explain your answer.
- 1.6 Define the essential properties of the following types of operating systems:
 - a. Batch
 - b. Interactive

- c. Time sharing
 - d. Real time
 - e. Network
 - f. Parallel
 - g. Distributed
 - h. Clustered
 - i. Handheld
- 1.7 We have stressed the need for an operating system to make efficient use of the computing hardware. When is it appropriate for the operating system to forsake this principle and to “waste” resources? Why is such a system not really wasteful?
- 1.8 Under what circumstances would a user be better off using a time-sharing system, rather than a PC or single-user workstation?
- 1.9 Describe the differences between symmetric and asymmetric multiprocessing. What are three advantages and one disadvantage of multiprocessor systems?
- 1.10 What is the main difficulty that a programmer must overcome in writing an operating system for a real-time environment?
- 1.11 Consider the various definitions of *operating system*. Consider whether the operating system should include applications such as web browsers and mail programs. Argue both pro and con positions, and support your answers.
- 1.12 What are the tradeoffs inherent in handheld computers?
- 1.13 Consider a computing cluster consisting of two nodes running a database. Describe two ways in which the cluster software can manage access to the data on the disk. Discuss the benefits and detriments of each.

Bibliographical Notes

Time-sharing systems were proposed first by Strachey [1959]. The earliest time-sharing systems were the Compatible Time-Sharing System (CTSS) developed at MIT (Corbato et al. [1962]) and the SDC Q-32 system, built by the System Development Corporation (Schwartz et al. [1964], Schwartz and Weissman [1967]). Other early, but more sophisticated, systems include the MULTIplexed Information and Computing Services (MULTICS) system developed at MIT (Corbato and Vyssotsky [1965]), the XDS-940 system developed at the University of

California at Berkeley (Lichtenberger and Pirtle [1965]), and the IBM TSS/360 system (Lett and Konigsford [1968]).

MS-DOS and PCs are described by Norton [1986] and by Norton and Wilton [1988]. An overview of the Apple Macintosh hardware and software is presented in Apple [1987]. The OS/2 operating system is covered in Microsoft [1989]. More OS/2 information can be found in Letwin [1988] and in Deitel and Kogan [1992]. Solomon and Russinovich [2000] discuss the structure of Microsoft Windows 2000 operating system.

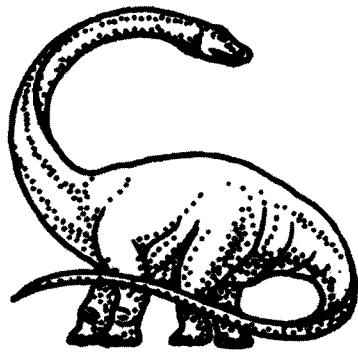
A good coverage of cluster computing is presented by Buyya [1999]. Recent advances in cluster computing is offered by Ahmed [2000].

Discussions concerning handheld devices are offered by Murray [1998] and Rhodes and McKeehan [1999].

Many general textbooks cover operating systems, including Milenkovic [1987], Finkel [1988], Deitel [1990], Stallings [2000b], Nutt [1999] and Tanenbaum [2001].

Chapter 2

COMPUTER-SYSTEM STRUCTURES



We need to have a general knowledge of the structure of a computer system before we can explore the details of system operation. In this chapter, we look at several disparate parts of this structure to round out our background knowledge. This chapter is mostly concerned with computer-system architecture, so you can skim or skip it if you already understand the concepts. The first topics covered here include system startup, I/O, and storage.

The operating system must also ensure the correct operation of the computer system. To ensure that user programs will not interfere with the proper operation of the system, the hardware must provide appropriate mechanisms to ensure correct behavior. Later in this chapter, we describe the basic computer architecture that makes it possible to write a functional operating system. We conclude with a network architecture overview.

2.1 ■ Computer-System Operation

A modern, general-purpose computer system consists of a CPU and a number of device controllers that are connected through a common bus that provides access to shared memory (Figure 2.1). Each device controller is in charge of a specific type of device (for example, disk drives, audio devices, and video displays). The CPU and the device controllers can execute concurrently, competing for memory cycles. To ensure orderly access to the shared memory, a memory controller is provided whose function is to synchronize access to the memory.

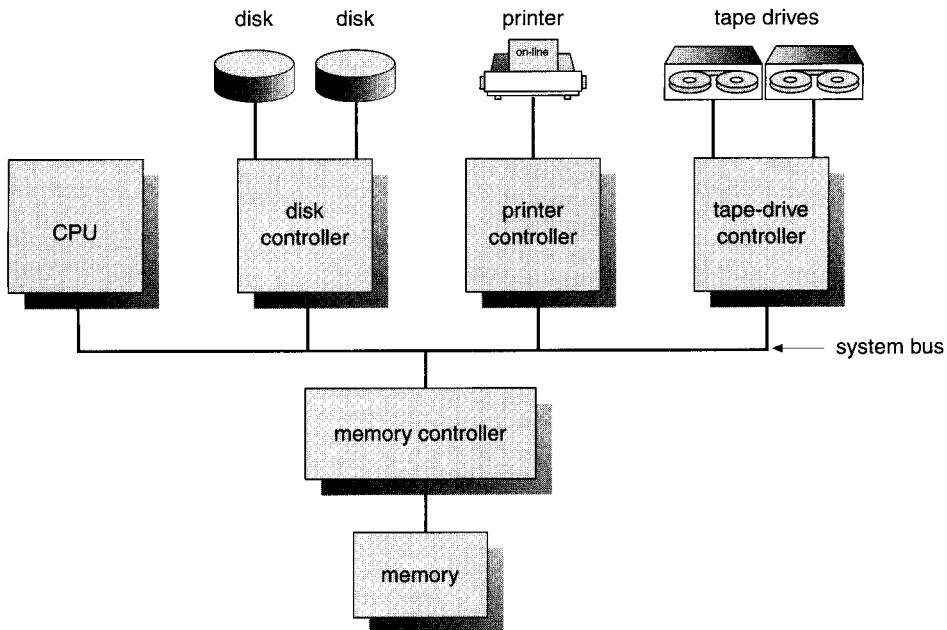


Figure 2.1 A modern computer system.

For a computer to start running—for instance, when it is powered up or rebooted—it needs to have an initial program to run. This initial program, or **bootstrap program**, tends to be simple. Typically, it is stored in read-only memory (ROM) such as firmware or EEPROM within the computer hardware. It initializes all aspects of the system, from CPU registers to device controllers to memory contents. The bootstrap program must know how to load the operating system and to start executing that system. To accomplish this goal, the bootstrap program must locate and load into memory the operating-system kernel. The operating system then starts executing the first process, such as “init,” and waits for some event to occur.

The occurrence of an event is usually signaled by an **interrupt** from either the hardware or the software. Hardware may trigger an interrupt at any time by sending a signal to the CPU, usually by way of the system bus. Software may trigger an interrupt by executing a special operation called a **system call** (also called a **monitor call**).

Modern operating systems are **interrupt driven**. If there are no processes to execute, no I/O devices to service, and no users to whom to respond, an operating system will sit quietly, waiting for something to happen. Events are almost always signaled by the occurrence of an interrupt or a trap. A **trap** (or an **exception**) is a software-generated interrupt caused either by an error (for example, division by zero or invalid memory access) or by a specific request

from a user program that an operating-system service be performed. The interrupt-driven nature of an operating system defines that system's general structure. For each type of interrupt, separate segments of code in the operating system determine what action should be taken. An interrupt service routine is provided that is responsible for dealing with the interrupt.

When the CPU is interrupted, it stops what it is doing and immediately transfers execution to a fixed location. The fixed location usually contains the starting address where the service routine for the interrupt is located. The interrupt service routine executes; on completion, the CPU resumes the interrupted computation. A time line of this operation is shown in Figure 2.2.

Interrupts are an important part of a computer architecture. Each computer design has its own interrupt mechanism, but several functions are common. The interrupt must transfer control to the appropriate interrupt service routine. The straightforward method for handling this transfer would be to invoke a generic routine to examine the interrupt information; the routine, in turn, would call the interrupt-specific handler. However, interrupts must be handled quickly, and, given that only a predefined number of interrupts is possible, a table of pointers to interrupt routines can be used instead. The interrupt routine is then called indirectly through the table, with no intermediate routine needed. Generally, the table of pointers is stored in low memory (the first 100 or so locations). These locations hold the addresses of the interrupt service routines for the various devices. This array, or **interrupt vector**, of addresses is then indexed by a unique device number, given with the interrupt request, to provide the address of the interrupt service routine for the interrupting device. Operating systems as different as MS-DOS and UNIX dispatch interrupts in this manner.

The interrupt architecture must also save the address of the interrupted instruction. Many old designs simply stored the interrupt address in a fixed location or in a location indexed by the device number. More recent architec-

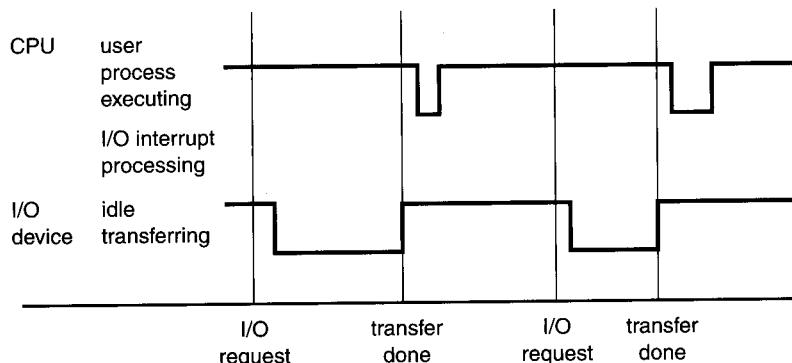


Figure 2.2 Interrupt time line for a single process doing output.

tures store the return address on the system stack. If the interrupt routine needs to modify the processor state—for instance, by modifying register values—it must explicitly save the current state and then restore that state before returning. After the interrupt is serviced, the saved return address is loaded into the program counter, and the interrupted computation resumes as though the interrupt had not occurred.

A system call is invoked in a variety of ways, depending on the functionality provided by the underlying processor. In all forms, it is the method used by a process to request action by the operating system. A system call usually takes the form of a trap to a specific location in the interrupt vector. This trap can be executed by a generic trap instruction, although some systems (such as the MIPS R2000 family) have a specific `syscall` instruction.

2.2 ■ I/O Structure

As we discussed in Section 2.1, a general-purpose computer system consists of a CPU and multiple device controllers that are connected through a common bus. Each device controller is in charge of a specific type of device. Depending on the controller, there may be more than one attached device. For instance, the **small computer-systems interface (SCSI)** controller can have seven or more devices attached to it. A device controller maintains some local buffer storage and a set of special-purpose registers. The device controller is responsible for moving the data between the peripheral devices that it controls and its local buffer storage. The size of the local buffer within a device controller varies from one controller to another, depending on the particular device being controlled. For example, the size of the buffer of a disk controller is the same as or a multiple of the size of the smallest addressable portion of a disk, called a **sector**, which is usually 512 bytes.

2.2.1 I/O Interrupts

To start an I/O operation, the CPU loads the appropriate registers within the device controller. The device controller, in turn, examines the contents of these registers to determine what action to take. For example, if it finds a read request, the controller will start the transfer of data from the device to its local buffer. Once the transfer of data is complete, the device controller informs the CPU that it has finished its operation. It accomplishes this communication by triggering an interrupt.

This situation will occur, in general, as the result of a user process requesting I/O. Once the I/O is started, two courses of action are possible. In the simplest case, the I/O is started; then, at I/O completion, control is returned to the user process. This case is known as **synchronous** I/O. The other possibility, called **asynchronous** I/O, returns control to the user program without

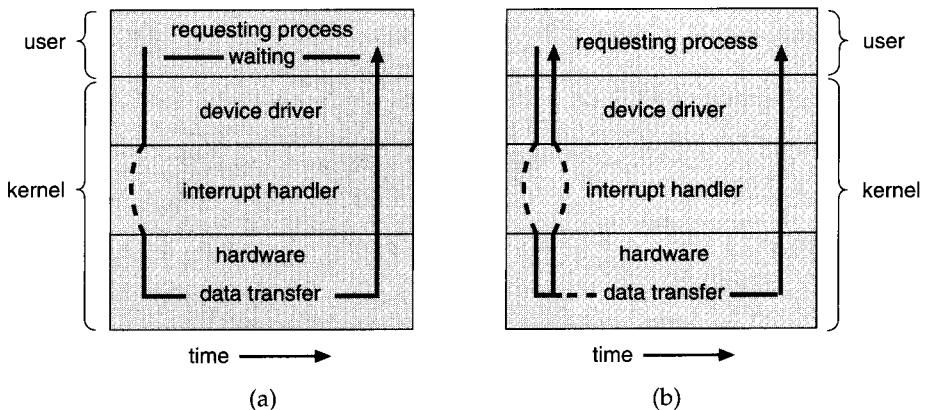


Figure 2.3 Two I/O methods: (a) synchronous, and (b) asynchronous.

waiting for the I/O to complete. The I/O then can continue while other system operations occur (Figure 2.3).

Waiting for I/O completion may be accomplished in one of two ways. Some computers have a special `wait` instruction that idles the CPU until the next interrupt. Machines that do not have such an instruction may have a wait loop:

```
Loop: jmp Loop
```

This tight loop simply continues until an interrupt occurs, transferring control to another part of the operating system. Such a loop might also need to poll any I/O devices that do not support the interrupt structure; instead, these devices simply set a flag in one of their registers and expect the operating system to notice that flag.

If the CPU always waits for I/O completion, at most one I/O request is outstanding at a time. Thus, whenever an I/O interrupt occurs, the operating system knows exactly which device is interrupting. On the other hand, this approach excludes concurrent I/O operations to several devices, and also excludes the possibility of overlapping useful computation with I/O.

A better alternative is to start the I/O and then to continue processing other operating-system or user program code. A system call is then needed to allow the user program to wait for I/O completion, if desired. If no user programs are ready to run, and the operating system has no other work to do, we still require the `wait` instruction or idle loop, as before. We also need to be able to keep track of many I/O requests at the same time. For this purpose, the operating system uses a table containing an entry for each I/O device: the **device-status table** (Figure 2.4). Each table entry indicates the device's type, address, and state (not functioning, idle, or busy). If the device is busy with a request, the type of request and other parameters will be stored in the table entry for that

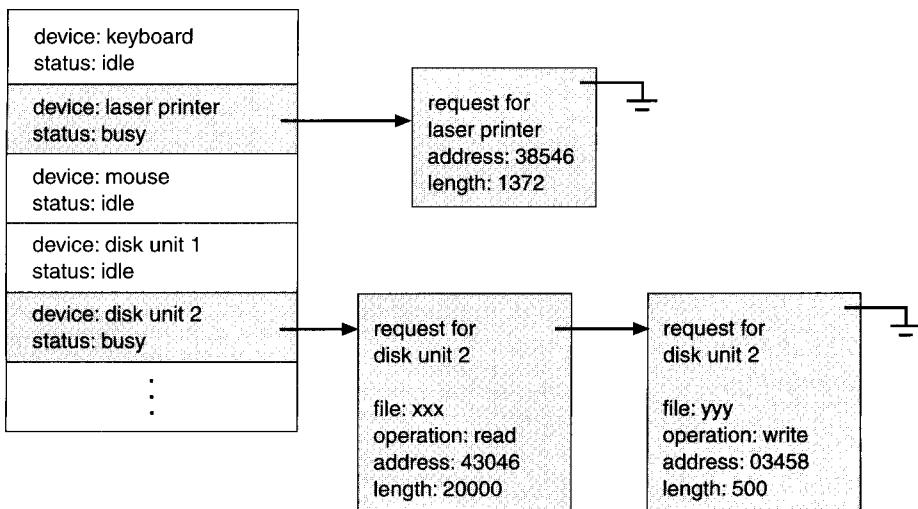


Figure 2.4 Device-status table.

device. Since it is possible for other processes to issue requests to the same device, the operating system will also maintain a wait queue—a list of waiting requests—for each I/O device.

An I/O device interrupts when it needs service. When an interrupt occurs, the operating system first determines which I/O device caused the interrupt. It then indexes into the I/O device table to determine the status of that device, and modifies the table entry to reflect the occurrence of the interrupt. For most devices, an interrupt signals completion of an I/O request. If there are additional requests waiting in the queue for this device, the operating system starts processing the next request.

Finally, control is returned from the I/O interrupt. If a process was waiting for this request to complete (as recorded in the device-status table), we can now return control to it. Otherwise, we return to whatever we were doing before the I/O interrupt: to the execution of the user program (the program started an I/O operation and that operation has now finished, but the program has not yet waited for the operation to complete) or to the wait loop (the program started two or more I/O operations and is waiting for a particular one to finish, but this interrupt was from one of the other operations). In a time-sharing system, the operating system could switch to another ready-to-run process.

The schemes used by specific input devices may vary from this one. Many interactive systems allow users to type ahead—to enter data before the data are requested—on the keyboard. In this case, interrupts may occur, signaling the arrival of characters from the terminal, while the device-status block indicates that no program has requested input from this device. If typeahead is to be allowed, then a buffer must be provided to store the typeahead characters until

some program wants them. In general, we may need a buffer for each input device.

The main advantage of asynchronous I/O is increased system efficiency. While I/O is taking place, the system CPU can be used for processing or starting I/Os to other devices. Because I/O can be slow compared to processor speed, the system makes efficient use of its facilities. In Section 2.2.2, we describe another mechanism for improving system performance.

2.2.2 DMA Structure

In a simple terminal-input driver, when a line is to be read from the terminal, the first character typed is sent to the computer. When that character is received, the asynchronous-communication (or serial-port) device to which the terminal line is connected interrupts the CPU. When the interrupt request from the terminal arrives, the CPU is about to execute some instruction. (If the CPU is in the middle of executing an instruction, the interrupt is normally held pending the completion of instruction execution.) The address of this interrupted instruction is saved, and control is transferred to the interrupt service routine for the appropriate device.

The interrupt service routine saves the contents of any CPU registers that it will need to use. It checks for any error conditions that might have resulted from the most recent input operation. It then takes the character from the device, and stores that character in a buffer. The interrupt routine must also adjust pointer and counter variables, to be sure that the next input character will be stored at the next location in the buffer. The interrupt routine next sets a flag in memory indicating to the other parts of the operating system that new input has been received. The other parts are responsible for processing the data in the buffer, and for transferring the characters to the program that is requesting input (see Section 2.5). Then, the interrupt service routine restores the contents of any saved registers, and transfers control back to the interrupted instruction.

If characters are being typed to a 9600-baud terminal, the terminal can accept and transfer one character approximately every 1 millisecond, or 1000 microseconds. A well-written interrupt service routine to input characters into a buffer may require 2 microseconds per character, leaving 998 microseconds out of every 1000 for CPU computation (and for servicing of other interrupts). Given this disparity, asynchronous I/O is usually assigned a low interrupt priority, allowing other, more important interrupts to be processed first, or even to preempt the current interrupt for another. A high-speed device, however—such as a tape, disk, or communications network—may be able to transmit information at close to memory speeds; if the CPU needs two microseconds to respond to each interrupt and interrupts arrive every four microseconds, for example, that does not leave much time for process execution.

To solve this problem, **direct memory access (DMA)** is used for high-speed I/O devices. After setting up buffers, pointers, and counters for the I/O device,

the device controller transfers an entire block of data directly to or from its own buffer storage to memory, with no intervention by the CPU. Only one interrupt is generated per block, rather than the one interrupt per byte (or word) generated for low-speed devices.

The basic operation of the CPU is the same. A user program, or the operating system itself, may request data transfer. The operating system finds a buffer (an empty buffer for input, or a full buffer for output) from a pool of buffers for the transfer. (A buffer is typically 128 to 4,096 bytes, depending on the device type.) Next, a portion of the operating system called a **device driver** sets the DMA controller registers to use appropriate source and destination addresses, and transfer length. The DMA controller is then instructed to start the I/O operation. While the DMA controller is performing the data transfer, the CPU is free to perform other tasks. Since the memory generally can transfer only one word at a time, the DMA controller “steals” memory cycles from the CPU. This cycle stealing can slow down the CPU execution while a DMA transfer is in progress. The DMA controller interrupts the CPU when the transfer has been completed.

2.3 ■ Storage Structure

Computer programs must be in main memory (also called **random-access memory** or **RAM**) to be executed. Main memory is the only large storage area (millions to billions of bytes) that the processor can access directly. It is implemented in a semiconductor technology called **dynamic random-access memory (DRAM)**, which forms an array of memory words. Each word has its own address. Interaction is achieved through a sequence of **load** or **store** instructions to specific memory addresses. The **load** instruction moves a word from main memory to an internal register within the CPU, whereas the **store** instruction moves the content of a register to main memory. Aside from explicit loads and stores, the CPU automatically loads instructions from main memory for execution.

A typical instruction-execution cycle, as executed on a system with a **von Neumann** architecture, will first fetch an instruction from memory and will store that instruction in the **instruction register**. The instruction is then decoded and may cause operands to be fetched from memory and stored in some internal register. After the instruction on the operands has been executed, the result may be stored back in memory. Notice that the memory unit sees only a stream of memory addresses; it does not know how they are generated (by the instruction counter, indexing, indirection, literal addresses, and so on) or what they are for (instructions or data). Accordingly, we can ignore *how* a memory address is generated by a program. We are interested only in the sequence of memory addresses generated by the running program.

Ideally, we want the programs and data to reside in main memory permanently. This arrangement is not possible for the following two reasons:

1. Main memory is usually too small to store all needed programs and data permanently.
2. Main memory is a *volatile* storage device that loses its contents when power is turned off or otherwise lost.

Thus, most computer systems provide **secondary storage** as an extension of main memory. The main requirement for secondary storage is that it be able to hold large quantities of data permanently.

The most common secondary-storage device is a **magnetic disk**, which provides storage of both programs and data. Most programs (web browsers, compilers, word processors, spreadsheets, and so on) are stored on a disk until they are loaded into memory. Many programs then use the disk as both a source and a destination of the information for their processing. Hence, the proper management of disk storage is of central importance to a computer system, as we discuss in Chapter 14.

In a larger sense, however, the storage structure that we have described—consisting of registers, main memory, and magnetic disks—is only one of many possible storage systems. There are also cache memory, CD-ROM, magnetic tapes, and so on. Each storage system provides the basic functions of storing a datum, and of holding that datum until it is retrieved at a later time. The main differences among the various storage systems lie in speed, cost, size, and volatility. In Sections 2.3.1 through 2.3.3, we describe main memory, magnetic disks, and magnetic tapes, because they illustrate the general properties of all important, commercially-available storage devices. In Chapter 14, we discuss the specific properties of many particular devices, such as floppy disks, hard disks, CD-ROMs, and DVDs.

2.3.1 Main Memory

Main memory and the registers built into the processor itself are the only storage that the CPU can access directly. There are machine instructions that take memory addresses as arguments, but none that take disk addresses. Therefore, any instructions in execution, and any data being used by the instructions, must be in one of these direct-access storage devices. If the data are not in memory, they must be moved there before the CPU can operate on them.

In the case of I/O, as mentioned in Section 2.1, each I/O controller includes registers to hold commands and the data being transferred. Usually, special I/O instructions allow data transfers between these registers and system memory. To allow more convenient access to I/O devices, many computer architectures provide **memory-mapped I/O**. In this case, ranges of memory addresses are

set aside, and are mapped to the device registers. Reads and writes to these memory addresses cause the data to be transferred to and from the device registers. This method is appropriate for devices that have fast response times, such as video controllers. In the IBM PC, each location on the screen is mapped to a memory location. Displaying text on the screen is almost as easy as writing the text into the appropriate memory-mapped locations.

Memory-mapped I/O is also convenient for other devices, such as the serial and parallel ports used to connect modems and printers to a computer. The CPU transfers data through these kinds of devices by reading and writing a few device registers, called an **I/O port**. To send out a long string of bytes through a memory-mapped serial port, the CPU writes one data byte to the data register, then sets a bit in the control register to signal that the byte is available. The device takes the data byte, and then clears the bit in the control register to signal that it is ready for the next byte. Then, the CPU can transfer the next byte. If the CPU uses polling to watch the control bit, constantly looping to see whether the device is ready, this method of operation is called **programmed I/O (PIO)**. If the CPU does not poll the control bit, but instead receives an interrupt when the device is ready for the next byte, the data transfer is said to be **interrupt driven**.

Registers that are built into the CPU are generally accessible within one cycle of the CPU clock. Most CPUs can decode instructions and perform simple operations on register contents at the rate of one or more operations per clock tick. The same cannot be said of main memory, which is accessed via a transaction on the memory bus. Memory access may take many cycles of the CPU clock to complete, in which case the processor normally needs to **stall**, since it does not have the data required to complete the instruction that it is executing. This situation is intolerable because of the frequency of memory accesses. The remedy is to add fast memory between the CPU and main memory. A memory buffer used to accommodate a speed differential, called a **cache**, is described in Section 2.4.1.

2.3.2 Magnetic Disks

Magnetic disks provide the bulk of secondary storage for modern computer systems. Conceptually, disks are relatively simple (Figure 2.5). Each disk **platter** has a flat circular shape, like a CD. Common platter diameters range from 1.8 to 5.25 inches. The two surfaces of a platter are covered with a magnetic material. We store information by recording it magnetically on the platters.

A read–write head “flies” just above each surface of every platter. The heads are attached to a **disk arm**, which moves all the heads as a unit. The surface of a platter is logically divided into circular **tracks**, which are subdivided into **sectors**. The set of tracks that are at one arm position forms a **cylinder**. There may be thousands of concentric cylinders in a disk drive, and each track may contain hundreds of sectors. The storage capacity of common disk drives is measured in gigabytes.

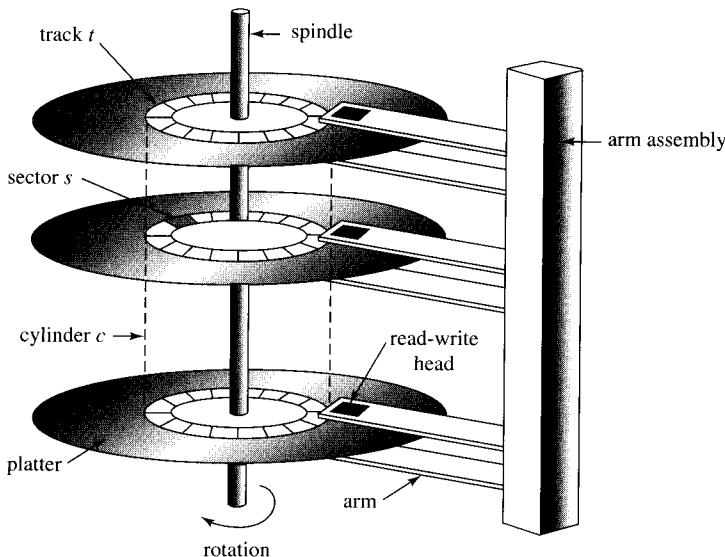


Figure 2.5 Moving-head disk mechanism.

When the disk is in use, a drive motor spins it at high speed. Most drives rotate 60 to 200 times per second. Disk speed has two parts. The **transfer rate** is the rate at which data flow between the drive and the computer. The **positioning time**, sometimes called the **random-access time**, consists of the time to move the disk arm to the desired cylinder, called the **seek time**, and the time for the desired sector to rotate to the disk head, called the **rotational latency**. Typical disks can transfer several megabytes of data per second, and they have seek times and rotational latencies of several milliseconds.

Because the disk head flies on an extremely thin (measured in microns) cushion of air, there is a danger of the head making contact with the disk surface. Although the disk platters are coated with a thin protective layer, sometimes the head will damage the magnetic surface. This accident is called a **head crash**. A head crash normally cannot be repaired; the entire disk must be replaced.

A disk can be **removable**, allowing different disks to be mounted as needed. Removable magnetic disks generally consist of one platter, held in a plastic case to prevent damage while not in the disk drive. **Floppy disks** are inexpensive removable magnetic disks that have a soft plastic case containing a flexible platter. The head of a floppy-disk drive generally sits directly on the disk surface, so the drive is designed to rotate more slowly than a hard-disk drive to reduce the wear on the disk surface. The storage capacity of a floppy disk is typically only 1 MB or so. Removable disks are available that work much like normal hard disks and have capacities measured in gigabytes.

A disk drive is attached to a computer by a set of wires called an **I/O bus**. Several kinds of buses are available, including **enhanced integrated drive electronics (EIDE)**, **advanced technology attachment (ATA)**, and **SCSI** buses. The data transfers on a bus are carried out by special electronic processors called **controllers**. The **host controller** is the controller at the computer end of the bus. A **disk controller** is built into each disk drive. To perform a disk I/O operation, the computer places a command into the host controller, typically using memory-mapped I/O ports, as described in Section 2.3.1. The host controller then sends the command via messages to the disk controller, and the disk controller operates the disk-drive hardware to carry out the command. Disk controllers usually have a built-in cache. Data transfer at the disk drive happens between the cache and the disk surface, and data transfer to the host, at fast electronic speeds, occurs between the cache and the host controller.

2.3.3 Magnetic Tapes

Magnetic tape was used as an early secondary-storage medium. Although it is relatively permanent and can hold large quantities of data, its access time is slow in comparison to that of main memory. In addition, random access to magnetic tape is about a thousand times slower than random access to magnetic disk, so tapes are not very useful for secondary storage. Tapes are used mainly for backup, for storage of infrequently used information, and as a medium for transferring information from one system to another.

A tape is kept in a spool, and is wound or rewound past a read–write head. Moving to the correct spot on a tape can take minutes, but once positioned, tape drives can write data at speeds comparable to disk drives. Tape capacities vary greatly, depending on the particular kind of tape drive. Some tapes hold 2 to 3 times more data than does a large disk drive. Tapes and their drivers are usually categorized by width, including 4, 8, and 19 millimeters, 1/4 and 1/2 inch.

2.4 ■ Storage Hierarchy

The wide variety of storage systems in a computer system can be organized in a hierarchy (Figure 2.6) according to speed and cost. The higher levels are expensive, but they are fast. As we move down the hierarchy, the cost per bit generally decreases, whereas the access time generally increases. This tradeoff is reasonable; if a given storage system were both faster and less expensive than another—other properties being the same—then there would be no reason to use the slower, more expensive memory. In fact, many early storage devices, including paper tape and core memories, are relegated to museums now that magnetic tape and **semiconductor memory** have become faster and cheaper. The top three levels of memory in Figure 2.6 may be constructed using semiconductor memory.

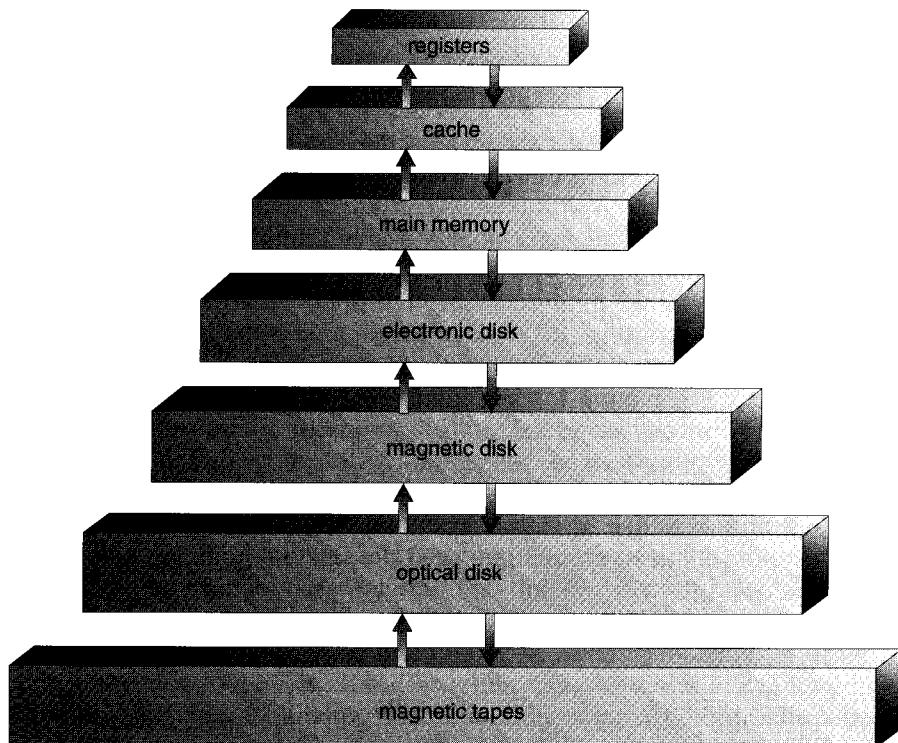


Figure 2.6 Storage-device hierarchy.

In addition to having differing speed and cost, the various storage systems are either volatile or nonvolatile. **Volatile storage** loses its contents when the power to the device is removed. In the absence of expensive battery and generator backup systems, data must be written to **nonvolatile storage** for safekeeping. In the hierarchy shown in Figure 2.6, the storage systems above the electronic disk are volatile, whereas those below are nonvolatile. An **electronic disk** can be designed to be either volatile or nonvolatile. During normal operation, the electronic disk stores data in a large DRAM array, which is volatile. But many electronic-disk devices contain a hidden magnetic hard disk and a battery for backup power. If external power is interrupted, the electronic-disk controller copies the data from RAM to the magnetic disk. When external power is restored, the controller copies the data back into the RAM.

The design of a complete memory system must balance all these factors: It uses only as much expensive memory as necessary, while providing as much inexpensive, nonvolatile memory as possible. Caches can be installed to improve performance where a large access-time or transfer-rate disparity exists between two components.

2.4.1 Caching

Caching is an important principle of computer systems. Information is normally kept in some storage system (such as main memory). As it is used, it is copied into a faster storage system—the cache—on a temporary basis. When we need a particular piece of information, we first check whether it is in the cache. If it is, we use the information directly from the cache; if it is not, we use the information from the main storage system, putting a copy in the cache under the assumption that we will need it again soon.

In addition, internal programmable registers, such as index registers, provide a high-speed cache for main memory. The programmer (or compiler) implements the register-allocation and register-replacement algorithms to decide which information to keep in registers and which to keep in main memory. There are also caches that are implemented totally in hardware. For instance, most systems have an instruction cache to hold the next instructions expected to be executed. Without this cache, the CPU would have to wait several cycles while an instruction is fetched from main memory. For similar reasons, most systems have one or more high-speed data caches in the memory hierarchy. We are not concerned with these hardware-only caches in this text, since they are outside of the control of the operating system.

Because caches have limited size, **cache management** is an important design problem. Careful selection of the cache size and of a replacement policy can result in 80 to 99 percent of all accesses being in the cache, greatly increasing performance. Various replacement algorithms for software-controlled caches are discussed in Chapter 10.

Main memory can be viewed as a fast cache for secondary storage, since data in secondary storage must be copied into main memory for use, and data must be in main memory before being moved to secondary storage for safekeeping. The file-system data, which resides permanently on secondary storage, may appear on several levels in the storage hierarchy. At the highest level, the operating system may maintain a cache of file-system data in main memory. Also, electronic RAM disks (also known as **solid-state disks**) may be used for high-speed storage that is accessed through the file-system interface. The bulk of secondary storage is on magnetic disks. The magnetic-disk storage, in turn, is often backed up onto magnetic tapes or removable disks to protect against data loss in case of a hard-disk failure. Some systems automatically archive old file data from secondary storage to tertiary storage, such as tape jukeboxes, to lower the storage cost (see Chapter 14).

The movement of information between levels of a storage hierarchy may be either explicit or implicit, depending on the hardware design and the controlling operating-system software. For instance, data transfer from cache to CPU and registers is usually a hardware function, with no operating-system intervention. On the other hand, transfer of data from disk to memory is usually controlled by the operating system.

2.4.2 Coherency and Consistency

In a hierarchical storage structure, the same data may appear in different levels of the storage system. For example, suppose that an integer A is located in file B that is to be incremented by 1, and file B resides on magnetic disk. The increment operation proceeds by first issuing an I/O operation to copy the disk block on which A resides to main memory. This operation is followed by copying A to the cache and to an internal register. Thus, the copy of A appears in several places: on the magnetic disk, in main memory, in the cache, and in an internal register (see Figure 2.7). Once the increment takes place in the internal register, the value of A differs in the various storage systems. The value of A becomes the same only after the new value of A is written from the internal register back to the magnetic disk.

In a computing environment where only one process executes at a time, this arrangement poses no difficulties, since an access to the integer A will always be to the copy at the highest level of the hierarchy. However, in a multitasking environment, where the CPU is switched back and forth among various processes, extreme care must be taken to ensure that, if several processes wish to access A, then each of these processes will obtain the most recently updated value of A.

The situation becomes more complicated in a multiprocessor environment where, in addition to maintaining internal registers, each of the CPUs also contains a local cache. In such an environment, a copy of A may exist simultaneously in several caches. Since the various CPUs can all execute concurrently, we must make sure that an update to the value of A in one cache is immediately reflected in all other caches where A resides. This situation is called **cache coherency**, and it is usually a hardware problem (handled below the operating-system level).

In a distributed environment, the situation becomes even more complex. In such an environment, several copies (or replicas) of the same file can be kept on different computers that are distributed in space. Since the various replicas may be accessed and updated concurrently, we must ensure that, when a replica is updated in one place, all other replicas are brought up-to-date as soon as possible. There are various ways to achieve this guarantee, as we discuss in Chapter 16.

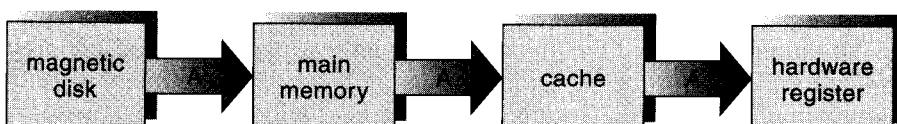


Figure 2.7 Migration of integer A from disk to register.

2.5 ■ Hardware Protection

Early computer systems were single-user programmer-operated systems. When the programmers operated the computer from the console, they had complete control over the system. As operating systems developed, however, this control was given to the operating system. Early operating systems were called **resident monitors**, and starting with the resident monitor, the operating system began to perform many of the functions, especially I/O, for which the programmer had previously been responsible.

In addition, to improve system utilization, the operating system began to *share* system resources among several programs simultaneously. With spooling, one program might have been executing while I/O occurred for other processes; the disk simultaneously held data for many processes. Multiprogramming put several programs in memory at the same time.

This sharing both improved utilization and increased problems. When the system was run without sharing, an error in a program could cause problems for only the one program that was running. With sharing, many processes could be adversely affected by a bug in one program.

For example, consider the simple batch operating system (Section 1.2.1), which provides nothing more than automatic job sequencing. If a program gets stuck in a loop reading input cards, the program will read through all its data and, unless something stops it, will continue reading the cards of the next job, and the next, and so on. This loop could prevent the correct operation of many jobs.

Even more subtle errors can occur in a multiprogramming system, where one erroneous program might modify the program or data of another program, or even the resident monitor itself. MS-DOS and the Macintosh OS both allow this kind of error.

Without protection against these sorts of errors, either the computer must execute only one process at a time, or all output must be suspect. A properly designed operating system must ensure that an incorrect (or malicious) program cannot cause other programs to execute incorrectly.

Many programming errors are detected by the hardware. These errors are normally handled by the operating system. If a user program fails in some way—such as by making an attempt either to execute an illegal instruction, or to access memory that is not in the user's address space—then the hardware will trap to the operating system. The trap transfers control through the interrupt vector to the operating system, just like an interrupt. Whenever a program error occurs, the operating system must abnormally terminate the program. This situation is handled by the same code as is a user-requested abnormal termination. An appropriate error message is given, and the memory of the program may be dumped. The memory dump is usually written to a file so that the user or programmer can examine it, and perhaps can correct and restart the program.

2.5.1 Dual-Mode Operation

To ensure proper operation, we must protect the operating system and all other programs and their data from any malfunctioning program. Protection is needed for any shared resource. The approach taken by many operating systems provides hardware support that allows us to differentiate among various modes of execution.

At the very least, we need two separate **modes** of operation: **user mode** and **monitor mode** (also called **supervisor mode**, **system mode**, or **privileged mode**). A bit, called the **mode bit**, is added to the hardware of the computer to indicate the current mode: monitor (0) or user (1). With the mode bit, we are able to distinguish between a task that is executed on behalf of the operating system, and one that is executed on behalf of the user. As we shall see, this architectural enhancement is useful for many other aspects of system operation.

At system boot time, the hardware starts in monitor mode. The operating system is then loaded, and starts user processes in user mode. Whenever a trap or interrupt occurs, the hardware switches from user mode to monitor mode (that is, changes the state of the mode bit to 0). Thus, whenever the operating system gains control of the computer, it is in monitor mode. The system always switches to user mode (by setting the mode bit to 1) before passing control to a user program.

The dual mode of operation provides us with the means for protecting the operating system from errant users, and errant users from one another. We accomplish this protection by designating some of the machine instructions that may cause harm as **privileged instructions**. The hardware allows privileged instructions to be executed only in monitor mode. If an attempt is made to execute a privileged instruction in user mode, the hardware does not execute the instruction, but rather treats the instruction as illegal and traps it to the operating system.

The concept of privileged instructions also provides us with the means for the user to interact with the operating system by asking the system to perform some designated tasks that only the operating system should do. Each such request is invoked by the user executing a privileged instruction. Such a request is known as a **system call** (also called a **monitor call** or an **operating-system function call**)—as described in Section 2.1.

When a system call is executed, it is treated by the hardware as a software interrupt. Control passes through the interrupt vector to a service routine in the operating system, and the mode bit is set to monitor mode. The system-call service routine is a part of the operating system. The monitor examines the interrupting instruction to determine what system call has occurred; a parameter indicates what type of service the user program is requesting. Additional information needed for the request may be passed in registers, on the stack, or in memory (with pointers to the memory locations passed in registers). The monitor verifies that the parameters are correct and legal, executes the request, and returns control to the instruction following the system call.

The lack of a hardware-supported dual mode can cause serious shortcomings in an operating system. For instance, MS-DOS was written for the Intel 8088 architecture, which has no mode bit, and therefore, no dual mode. A user program running awry can wipe out the operating system by writing over it with data, and multiple programs are able to write to a device at the same time, with possibly disastrous results. More recent and advanced versions of the Intel CPU, such as the Pentium, do provide dual-mode operation. As a result, more recent operating systems, such as Microsoft Windows 2000 and IBM OS/2, take advantage of this feature and provide greater protection for the operating system.

2.5.2 I/O Protection

A user program may disrupt the normal operation of the system by issuing illegal I/O instructions, by accessing memory locations within the operating system itself, or by refusing to relinquish the CPU. We can use various mechanisms to ensure that such disruptions cannot take place in the system.

To prevent users from performing illegal I/O, we define all I/O instructions to be privileged instructions. Thus, users cannot issue I/O instructions directly; they must do it through the operating system. For I/O protection to be complete, we must be sure that a user program can never gain control of the computer in monitor mode. If it could, I/O protection could be compromised.

Consider a computer executing in user mode. It will switch to monitor mode whenever an interrupt or trap occurs, jumping to the address determined from the interrupt vector. If a user program, as part of its execution, stores a new address in the interrupt vector, this new address could overwrite the previous address with an address in the user program. Then, when a corresponding trap or interrupt occurred, the hardware would switch to monitor mode, and would transfer control through the (modified) interrupt vector to the user program! The user program could gain control of the computer in monitor mode. In fact, user programs could gain control of the computer in monitor mode in many other ways. In addition, new bugs are discovered every day that can be exploited to bypass system protections. Those topics are discussed in Chapters 18 and 19. Thus, to do I/O, a user program executes a system call to request that the operating system perform I/O on its behalf (Figure 2.8). The operating system, executing in monitor mode, checks that the request is valid, and (if the request is valid) does the I/O requested. The operating system then returns to the user.

2.5.3 Memory Protection

To ensure correct operation, we must protect the interrupt vector from modification by a user program. In addition, we must also protect the interrupt-service routines in the operating system from modification. Even if the user did not

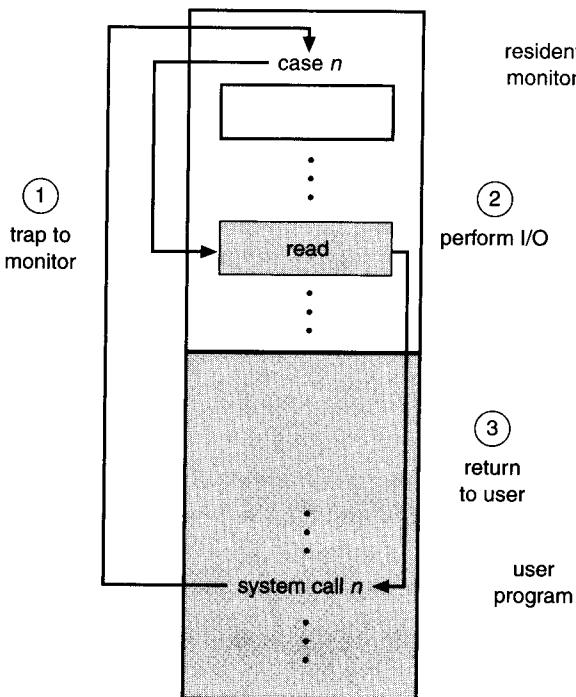


Figure 2.8 Use of a system call to perform I/O.

gain unauthorized control of the computer, modifying the interrupt service routines would probably disrupt the proper operation of the computer system and of its spooling and buffering.

We see then that we must provide memory protection at least for the interrupt vector and the interrupt-service routines of the operating system. In general, we want to protect the operating system from access by user programs, and, in addition, to protect user programs from one another. This protection must be provided by the hardware. It can be implemented in several ways, as we describe in Chapter 9. Here, we outline one such possible implementation.

To separate each program's memory space, we need the ability to determine the range of legal addresses that the program may access, and to protect the memory outside that space. We can provide this protection by using two registers, usually a base and a limit, as illustrated in Figure 2.9. The **base register** holds the smallest legal physical memory address; the **limit register** contains the size of the range. For example, if the base register holds 300040 and limit register is 120900, then the program can legally access all addresses from 300040 through 420940 inclusive.

This protection is accomplished by the CPU hardware comparing *every* address generated in user mode with the registers. Any attempt by a program

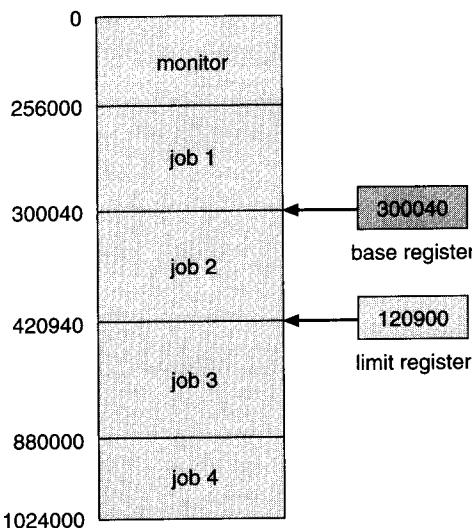


Figure 2.9 A base and a limit register define a logical address space.

executing in user mode to access monitor memory or other users' memory results in a trap to the monitor, which treats the attempt as a fatal error (Figure 2.10). This scheme prevents the user program from (accidentally or deliberately) modifying the code or data structures of either the operating system or other users.

The base and limit registers can be loaded by only the operating system, which uses a special privileged instruction. Since privileged instructions can be executed in only monitor mode, and since only the operating system executes in monitor mode, only the operating system can load the base and limit registers. This scheme allows the monitor to change the value of the registers, but prevents user programs from changing the registers' contents.

The operating system, executing in monitor mode, is given unrestricted access to both monitor and users' memory. This provision allows the operating system to load users' programs into users' memory, to dump out those programs in case of errors, to access and modify parameters of system calls, and so on.

2.5.4 CPU Protection

In addition to protecting I/O and memory, we must ensure that the operating system maintains control. We must prevent a user program from getting stuck in an infinite loop or not calling system services, and never returning control to the operating system. To accomplish this goal, we can use a **timer**. A timer can be set to interrupt the computer after a specified period. The period may be

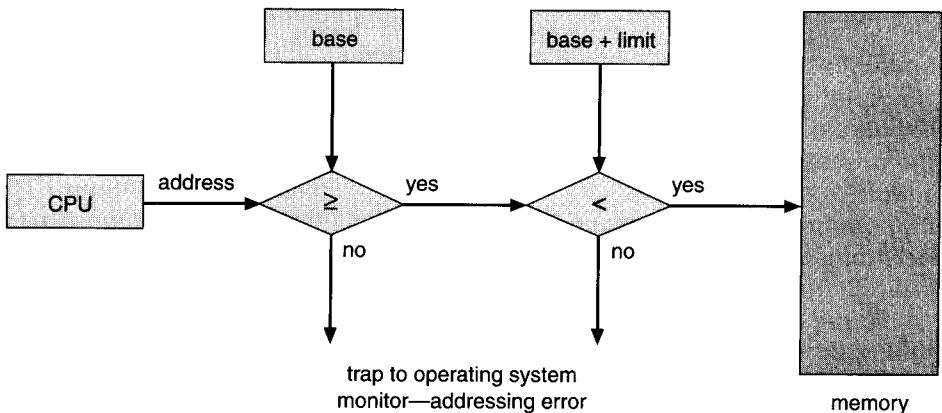


Figure 2.10 Hardware address protection with base and limit registers.

fixed (for example, 1/60 second) or variable (for example, from 1 millisecond to 1 second). A **variable timer** is generally implemented by a fixed-rate clock and a counter. The operating system sets the counter. Every time the clock ticks, the counter is decremented. When the counter reaches 0, an interrupt occurs. For instance, a 10-bit counter with a 1-millisecond clock allows interrupts at intervals from 1 millisecond to 1,024 milliseconds, in steps of 1 millisecond.

Before turning over control to the user, the operating system ensures that the timer is set to interrupt. If the timer interrupts, control transfers automatically to the operating system, which may treat the interrupt as a fatal error or may give the program more time. Clearly, instructions that modify the operation of the timer are privileged.

Thus, we can use the timer to prevent a user program from running too long. A simple technique is to initialize a counter with the amount of time that a program is allowed to run. A program with a 7-minute time limit, for example, would have its counter initialized to 420. Every second, the timer interrupts and the counter is decremented by 1. As long as the counter is positive, control is returned to the user program. When the counter becomes negative, the operating system terminates the program for exceeding the assigned time limit.

A more common use of a timer is to implement time sharing. In the most straightforward case, the timer could be set to interrupt every N milliseconds, where N is the **time slice** that each user is allowed to execute before the next user gets control of the CPU. The operating system is invoked at the end of each time slice to perform various housekeeping tasks, such as adding the value N to the record that specifies (for accounting purposes) the amount of time the user program has executed thus far. The operating system also saves registers, internal variables, and buffers, and changes several other parameters to prepare for the next program to run. This procedure is known as a **context switch**; it is explored in Chapter 4. Following a context switch, the next program continues

with its execution from the point at which it left off (when its previous time slice ran out).

Another use of the timer is to compute the current time. A timer interrupt signals the passage of some period, allowing the operating system to compute the current time in reference to some initial time. If we have interrupts every 1 second, and we have had 1427 interrupts since we were told that it was 1:00 P.M., then we can compute that the current time is 1:23:47 P.M. Some computers determine the current time in this manner, but the calculations must be done carefully for the time to be kept accurately, since the interrupt-processing time (and other times when interrupts are disabled) tends to cause the software clock to slow down. Most computers have a separate hardware time-of-day clock that is independent of the operating system.

2.6 ■ Network Structure

There are basically two types of networks: **local-area networks (LAN)** and **wide-area networks (WAN)**. The main difference between the two is the way in which they are geographically distributed. Local-area networks are composed of processors that are distributed over small geographical areas (such as a single building or a number of adjacent buildings). Wide-area networks, on the other hand, are composed of a number of autonomous processors that are distributed over a large geographical area (such as the United States). These differences imply major variations in the speed and reliability of the communications network, and they are reflected in the distributed operating-system design.

2.6.1 Local-Area Networks

Local-area networks emerged in the early 1970s, as a substitute for large main-frame computer systems. For many enterprises, it is more economical to have a number of small computers, each with its own self-contained applications, rather than a single large system. Because each small computer is likely to need a full complement of peripheral devices (such as disks and printers), and because some form of data sharing is likely to occur in a single enterprise, it was a natural step to connect these small systems into a network.

LANs are usually designed to cover a small geographical area (such as a single building, or a few adjacent buildings) and are generally used in an office environment. All the sites in such systems are close to one another, so the communication links tend to have a higher speed and lower error rate than do their counterparts in wide-area networks. High-quality (expensive) cables are needed to attain this higher speed and reliability. It is also possible to use the cable exclusively for data network traffic. Over longer distances, the cost of using high-quality cable is enormous, and the exclusive use of the cable tends to be prohibitive.

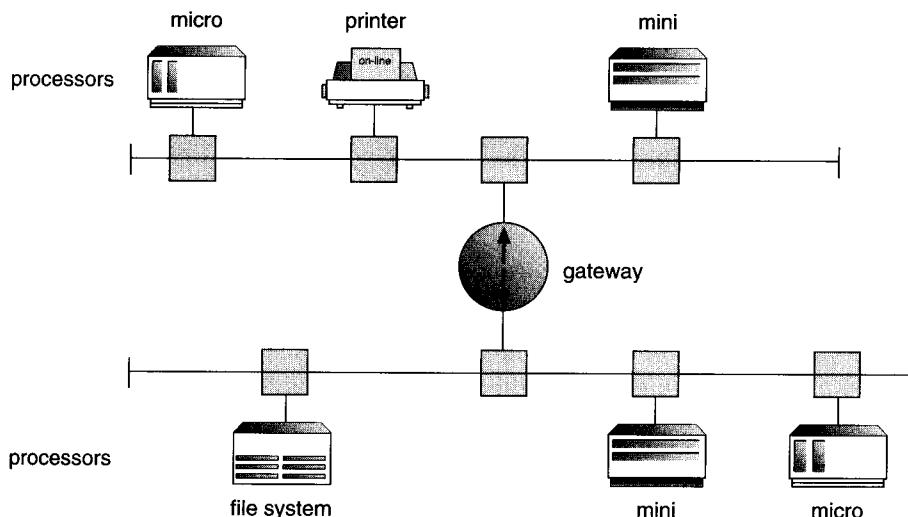


Figure 2.11 Local-area network.

The most common links in a local-area network are twisted pair and fiber optic cabling. The most common configurations are multiaccess bus, ring, and star networks. Communication speeds range from 1 megabit per second, for networks such as AppleTalk, infrared, and the new Bluetooth local radio network, to 1 gigabit per second for gigabit-Ethernet. Ten megabits per second is most common, and is the speed of **10BaseT Ethernet**. **100BaseT Ethernet** requires a higher-quality cable but runs at 100 megabits per second, and is becoming common. Also growing is the use of optical-fiber-based FDDI networking. The FDDI network is token-based and runs at over 100 megabits per second.

A typical LAN may consist of a number of different computers (from mainframes to laptops or PDAs), various shared peripheral devices (such as laser printers or magnetic-tape drives), and one or more gateways (specialized processors) that provide access to other networks (Figure 2.11). An Ethernet scheme is commonly used to construct LANs. An Ethernet network has no central controller, because it is a multiaccess bus, so new hosts can be added easily to the network.

2.6.2 Wide-Area Networks

Wide-area networks emerged in the late 1960s, mainly as an academic research project to provide efficient communication among sites, allowing hardware and software to be shared conveniently and economically by a wide community of users. The first WAN to be designed and developed was the *Arpanet*. Begun in 1968, the Arpanet has grown from a four-site experimental network to a

worldwide network of networks, the Internet, comprising millions of computer systems.

Because the sites in a WAN are physically distributed over a large geographical area, the communication links are, by default, relatively slow and unreliable. Typical links are telephone lines, leased (dedicated data) lines, microwave links, and satellite channels. These communication links are controlled by special **communication processors** (Figure 2.12), which are responsible for defining the interface through which the sites communicate over the network, as well as for transferring information among the various sites.

For example, the Internet WAN provides the ability for hosts at geographically separated sites to communicate with one another. The host computers typically differ from one another in type, speed, word length, operating system, and so on. Hosts are generally on LANs, which are, in turn, connected to the Internet via regional networks. The regional networks, such as NSFnet in the northeast United States, are interlinked with **routers** (Section 15.4.2) to form the worldwide network. Connections between networks frequently use a telephone-system service called T1, which provides a transfer rate of 1.544

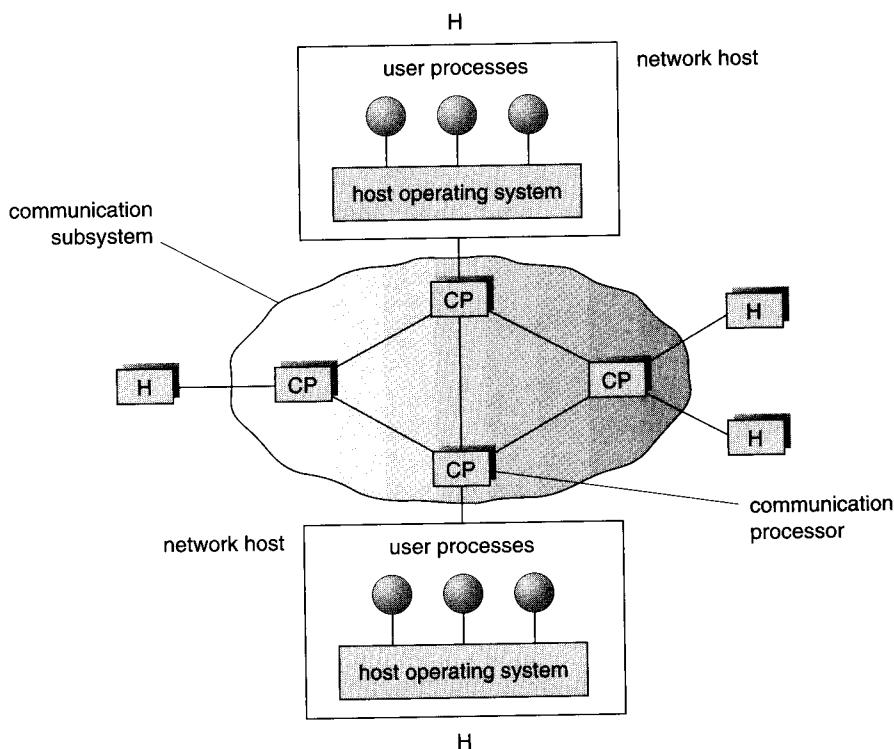


Figure 2.12 Communication processors in a wide-area network.

megabits per second over a leased line. For sites requiring faster Internet access, T1s are collected into multiple-T1 units that work in parallel to provide more throughput. For instance, a T3 is composed of 28 T1 connections and has a transfer rate of 45 megabits per second. The routers control the path each message takes through the net. This routing may be either dynamic, to increase communications efficiency, or static, to reduce security risks or to allow communications charges to be computed.

Other WANs in operation use standard telephone lines as their primary means of communication. **Modems** are devices that accept digital data from the computer side and convert it to the analog signals that the telephone system uses. A modem at the destination site converts the analog signal back to digital and the destination receives the data. The UNIX news network, UUCP, allows systems to communicate with each other at predetermined times, via modems, to exchange messages. The messages are then routed to other nearby systems and in this way either are propagated to all hosts on the network (public messages) or are transferred to their destination (private messages). WANs are generally slower than LANs; their transmission rates range from 1,200 bits per second to over 1 megabit per second. UUCP has been superseded by PPP, the Point-to-Point Protocol. PPP functions over modem connections, allowing home computers to be fully connected to the Internet.

2.7 ■ Summary

Multiprogramming and time-sharing systems improve performance by overlapping CPU and I/O operations on a single machine. Such an overlap requires that data transfer between the CPU and an I/O device be handled either by polled or interrupt-driven access to an I/O port, or by a DMA data transfer.

For a computer to do its job of executing programs, the programs must be in main memory. Main memory is the only large storage area that the processor can access directly. It is an array of words or bytes, ranging in size from hundreds of thousands to hundreds of millions. Each word has its own address. The main memory is a volatile storage device that loses its contents when power is turned off or lost. Most computer systems provide secondary storage as an extension of main memory. The main requirement of secondary storage is to be able to hold large quantities of data permanently. The most common secondary-storage device is a magnetic disk, which provides storage of both programs and data. A magnetic disk is a nonvolatile storage device that also provides random access. Magnetic tapes are used mainly for backup, for storage of infrequently used information, and as a medium for transferring information from one system to another.

The wide variety of storage systems in a computer system can be organized in a hierarchy according to their speed and cost. The higher levels are expen-

sive, but they are fast. As we move down the hierarchy, the cost per bit generally decreases, whereas the access time generally increases.

The operating system must ensure correct operation of the computer system. To prevent user programs from interfering with the proper operation of the system, the hardware has two modes: user mode and monitor mode. Various instructions (such as I/O instructions and halt instructions) are privileged, and can be executed in only monitor mode. The memory in which the operating system resides must also be protected from modification by the user. A timer prevents infinite loops. These facilities (dual mode, privileged instructions, memory protection, timer interrupt) are basic building blocks used by operating systems to achieve correct operation. Chapter 3 continues this discussion with details of the facilities that operating systems provide.

LANs and WANs are the two basic types of networks. Usually connected by expensive twisted-pair or fiber-optic cabling, LANs allow processors distributed over a small geographical area to communicate. Connected by telephone lines, leased lines, microwave links, or satellite channels, WANs allow processors distributed over a larger geographical area to communicate. LANs typically transmit more than 100 megabits per second, whereas slower WANs transmit from 1,200 bits per second to more than 1 megabit per second.

■ Exercises

2.1 Prefetching is a method of overlapping the I/O of a job with that job's own computation. The idea is simple. After a read operation completes and the job is about to start operating on the data, the input device is instructed to begin the next read immediately. The CPU and input device are then both busy. With luck, by the time that the job is ready for the next data item, the input device will have finished reading that data item. The CPU can then begin processing the newly read data, while the input device starts to read the following data. A similar idea can be used for output. In this case, the job creates data that are put into a buffer until an output device can accept them.

Compare the prefetching scheme with **spooling**, where the CPU overlaps the input of one job with the computation and output of other jobs.

- 2.2** How does the distinction between monitor mode and user mode function as a rudimentary form of protection (security) system?
- 2.3** What are the differences between a trap and an interrupt? What is the use of each function?
- 2.4** For what types of operations is DMA useful? Explain your answer.
- 2.5** Which of the following instructions should be privileged?

- a. Set value of timer.
 - b. Read the clock.
 - c. Clear memory.
 - d. Turn off interrupts.
 - e. Switch from user to monitor mode.
- 2.6 Some computer systems do not provide a privileged mode of operation in hardware. Is it possible to construct a secure operating system for these computers? Give arguments both that it is and that it is not possible.
- 2.7 Some early computers protected the operating system by placing it in a memory partition that could not be modified by either the user job or the operating system itself. Describe two difficulties that you think could arise with such a scheme.
- 2.8 Protecting the operating system is crucial to ensuring that the computer system operates correctly. Provision of this protection is the reason for dual-mode operation, memory protection, and the timer. To allow maximum flexibility, however, you should also place minimal constraints on the user.
- The following is a list of instructions that are normally protected. What is the *minimal* set of instructions that must be protected?
- a. Change to user mode.
 - b. Change to monitor mode.
 - c. Read from monitor memory.
 - d. Write into monitor memory.
 - e. Fetch an instruction from monitor memory.
 - f. Turn on timer interrupt.
 - g. Turn off timer interrupt.
- 2.9 Give two reasons why caches are useful. What problems do they solve? What problems do they cause? If a cache can be made as large as the device for which it is caching (for instance, a cache as large as a disk), why not make it that large and eliminate the device?
- 2.10 Writing an operating system that can operate without interference from malicious or undebugged user programs requires hardware assistance. Name three hardware aids for writing an operating system, and describe how they could be used together to protect the operating system.

- 2.11 Some CPUs provide for more than two modes of operation. What are two possible uses of these multiple modes?
- 2.12 What are the main differences between a WAN and a LAN?
- 2.13 What network configuration would best suit the following environments?
- A dormitory floor
 - A university campus
 - A state
 - A nation

Bibliographical Notes

Hennessy and Patterson [1996] provide coverage of I/O systems and buses, and of system architecture in general. Tanenbaum [1990] describes the architecture of microcomputers, starting at a detailed hardware level.

Discussions concerning magnetic-disk technology are presented by Freedman [1983] and by Harker et al. [1981]. Optical disks are covered by Kenville [1982], Fujitani [1984], O'Leary and Kitts [1985], Gait [1988], and Olsen and Kenley [1989]. Discussions of floppy disks are offered by Pechura and Schoeffler [1983] and by Sarisky [1983].

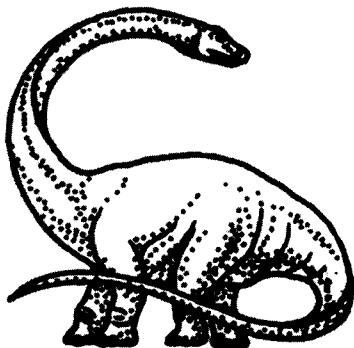
Cache memories, including associative memory, are described and analyzed by Smith [1982]. That paper also includes an extensive bibliography on the subject.

General discussions concerning mass-storage technology are offered by Chi [1982] and by Hoagland [1985].

Tanenbaum [1996] and Halsall [1992] provide general overviews of computer networks. Fortier [1989] presents a detailed discussion of networking hardware and software.

Chapter 3

OPERATING- SYSTEM STRUCTURES



An operating system provides the environment within which programs are executed. Internally, operating systems vary greatly in their makeup, being organized along many different lines. The design of a new operating system is a major task. The goals of the system must be well defined before the design begins. The type of system desired is the basis for choices among various algorithms and strategies.

An operating system may be viewed from several vantage points. One is by examining the services that it provides. Another is by looking at the interface that it makes available to users and programmers. A third is by disassembling the system into its components and their interconnections. In this chapter, we explore all three aspects of operating systems, showing the viewpoints of users, programmers, and operating-system designers. We consider what services an operating system provides, how they are provided, and what the various methodologies are for designing such systems.

3.1 ■ System Components

We can create a system as large and complex as an operating system only by partitioning it into smaller pieces. Each piece should be a well-delineated portion of the system, with carefully defined inputs, outputs, and functions. Obviously, not all systems have the same structure. However, many modern systems share the goal of supporting the system components outlined in Sections 3.1.1 through 3.1.8.

3.1.1 Process Management

A program does nothing unless its instructions are executed by a CPU. A **process** can be thought of as a program in execution, but its definition will broaden as we explore it further. A time-shared user program such as a compiler is a process. A word-processing program being run by an individual user on a PC is a process. A system task, such as sending output to a printer, is also a process. For now, you can consider a process to be a job or a time-shared program, but later you will learn that the concept is more general. As we shall see in Chapter 4, we can provide system calls that allow processes to create subprocesses to execute concurrently.

A process needs certain resources—including CPU time, memory, files, and I/O devices—to accomplish its task. These resources are either given to the process when it is created, or allocated to it while it is running. In addition to the various physical and logical resources that a process obtains when it is created, various initialization data (or input) may be passed along. For example, consider a process whose function is to display the status of a file on the screen of a terminal. The process will be given as an input the name of the file, and will execute the appropriate instructions and system calls to obtain and display on the terminal the desired information. When the process terminates, the operating system will reclaim any reusable resources.

We emphasize that a program by itself is not a process; a program is a *passive* entity, such as the contents of a file stored on disk, whereas a process is an *active* entity, with a **program counter** specifying the next instruction to execute. The execution of a process must be sequential. The CPU executes one instruction of the process after another, until the process completes. Further, at any time, at most one instruction is executed on behalf of the process. Thus, although two processes may be associated with the same program, they are nevertheless considered two separate execution sequences. It is common to have a program that spawns many processes as it runs.

A process is the unit of work in a system. Such a system consists of a collection of processes, some of which are operating-system processes (those that execute system code) and the rest of which are user processes (those that execute user code). All these processes can potentially execute concurrently, by multiplexing the CPU among them.

The operating system is responsible for the following activities in connection with process management:

- Creating and deleting both user and system processes
- Suspending and resuming processes
- Providing mechanisms for process synchronization
- Providing mechanisms for process communication
- Providing mechanisms for deadlock handling

We discuss process-management techniques in Chapters 4 through 7.

3.1.2 Main-Memory Management

As we discussed in Chapter 1, the main memory is central to the operation of a modern computer system. Main memory is a large array of words or bytes, ranging in size from hundreds of thousands to billions. Each word or byte has its own address. Main memory is a repository of quickly accessible data shared by the CPU and I/O devices. The central processor reads instructions from main memory during the instruction-fetch cycle, and it both reads and writes data from main memory during the data-fetch cycle. The I/O operations implemented via DMA also read and write data in main memory. The main memory is generally the only large storage device that the CPU is able to address and access directly. For example, for the CPU to process data from disk, those data must first be transferred to main memory by CPU-generated I/O calls. Equivalently, instructions must be in memory for the CPU to execute them.

For a program to be executed, it must be mapped to absolute addresses and loaded into memory. As the program executes, it accesses program instructions and data from memory by generating these absolute addresses. Eventually, the program terminates, its memory space is declared available, and the next program can be loaded and executed.

To improve both the utilization of the CPU and the speed of the computer's response to its users, we must keep several programs in memory. Many different memory-management schemes are available, and the effectiveness of the different algorithms depends on the particular situation. Selection of a memory-management scheme for a specific system depends on many factors—especially on the *hardware* design of the system. Each algorithm requires its own hardware support.

The operating system is responsible for the following activities in connection with memory management:

- Keeping track of which parts of memory are currently being used and by whom
- Deciding which processes are to be loaded into memory when memory space becomes available
- Allocating and deallocating memory space as needed

Memory-management techniques will be discussed in Chapters 9 and 10.

3.1.3 File Management

File management is one of the most visible components of an operating system. Computers can store information on several different types of physical media. Magnetic tape, magnetic disk, and optical disk are the most common media. Each of these media has its own characteristics and physical organization. Each

medium is controlled by a device, such as a disk drive or tape drive, that also has unique characteristics. These properties include access speed, capacity, data-transfer rate, and access method (sequential or random).

For convenient use of the computer system, the operating system provides a uniform logical view of information storage. The operating system abstracts from the physical properties of its storage devices to define a logical storage unit, the file. The operating system maps files onto physical media, and accesses these files via the storage devices.

A **file** is a collection of related information defined by its creator. Commonly, files represent programs (both source and object forms) and data. Data files may be numeric, alphabetic, or alphanumeric. Files may be free-form (for example, text files), or may be formatted rigidly (for example, fixed fields). A file consists of a sequence of bits, bytes, lines, or records whose meanings are defined by their creators. The concept of a file is an extremely general one.

The operating system implements the abstract concept of a file by managing mass storage media, such as disks and tapes, and the devices that control them. Also, files are normally organized into directories to ease their use. Finally, when multiple users have access to files, we may want to control by whom and in what ways (for example, read, write, append) files may be accessed.

The operating system is responsible for the following activities in connection with file management:

- Creating and deleting files
- Creating and deleting directories
- Supporting primitives for manipulating files and directories
- Mapping files onto secondary storage
- Backing up files on stable (nonvolatile) storage media

File-management techniques will be discussed in Chapters 11 and 12.

3.1.4 I/O-System Management

One of the purposes of an operating system is to hide the peculiarities of specific hardware devices from the user. For example, in UNIX, the peculiarities of I/O devices are hidden from the bulk of the operating system itself by the **I/O subsystem**. The I/O subsystem consists of

- A memory-management component that includes buffering, caching, and spooling
- A general device-driver interface
- Drivers for specific hardware devices

Only the device driver knows the peculiarities of the specific device to which it is assigned.

We discussed in Chapter 2 how interrupt handlers and device drivers are used in the construction of efficient I/O subsystems. In Chapter 13, we will discuss how the I/O subsystem interfaces to the other system components, manages devices, transfers data, and detects I/O completion.

3.1.5 Secondary-Storage Management

The main purpose of a computer system is to execute programs. These programs, with the data they access, must be in main memory, or **primary storage**, during execution. Because main memory is too small to accommodate all data and programs, and because the data that it holds are lost when power is lost, the computer system must provide **secondary storage** to back up main memory. Most modern computer systems use disks as the principal on-line storage medium, for both programs and data. Most programs—including compilers, assemblers, sort routines, editors, and formatters—are stored on a disk until loaded into memory, and then use the disk as both the source and destination of their processing. Hence, the proper management of disk storage is of central importance to a computer system.

The operating system is responsible for the following activities in connection with disk management:

- Free-space management
- Storage allocation
- Disk scheduling

Because secondary storage is used frequently, it must be used efficiently. The entire speed of operation of a computer may hinge on the speeds of the disk subsystem and of the algorithms that manipulate that subsystem. Techniques for secondary-storage management will be discussed in Chapter 14.

3.1.6 Networking

A **distributed system** is a collection of processors that do not share memory, peripheral devices, or a clock. Instead, each processor has its own local memory and clock, and the processors communicate with one another through various communication lines, such as high-speed buses or networks. The processors in a distributed system vary in size and function. They may include small microprocessors, workstations, minicomputers, and large, general-purpose computer systems.

The processors in the system are connected through a **communication network**, which can be configured in a number of different ways. The network may be fully or partially connected. The communication-network design must

consider message routing and connection strategies, and the problems of contention and security.

A distributed system collects physically separate, possibly heterogeneous, systems into a single coherent system, providing the user with access to the various resources that the system maintains. Access to a shared resource allows computation speedup, increased functionality, increased data availability, and enhanced reliability. Operating systems usually generalize network access as a form of file access, with the details of networking being contained in the network interface's device driver. The protocols that create a distributed system can have a great effect on that system's utility and popularity. The innovation of the World Wide Web was to create a new access method for information sharing. It improved on the existing **file-transfer protocol (FTP)** and **network file-system (NFS)** protocol by removing the need for a user to log in before she is allowed to use a remote resource. It defined a new protocol, **hypertext transfer protocol (http)**, for use in communication between a web server and a web browser. A web browser then just needs to send a request for information to a remote machine's web server, and the information (text, graphics, links to other information) is returned. This increase in convenience fostered huge growth in the use of http and of the Web in general.

We discuss networks and distributed systems in Chapters 15 through 17.

3.1.7 Protection System

If a computer system has multiple users and allows the concurrent execution of multiple processes, then the various processes must be protected from one another's activities. For that purpose, mechanisms ensure that the files, memory segments, CPU, and other resources can be operated on by only those processes that have gained proper authorization from the operating system.

For example, memory-addressing hardware ensures that a process can execute only within its own address space. The timer ensures that no process can gain control of the CPU without eventually relinquishing control. Device-control registers are not accessible to users, so that the integrity of the various peripheral devices is protected.

Protection is any mechanism for controlling the access of programs, processes, or users to the resources defined by a computer system. This mechanism must provide means for specification of the controls to be imposed and means for enforcement.

Protection can improve reliability by detecting latent errors at the interfaces between component subsystems. Early detection of interface errors can often prevent contamination of a healthy subsystem by another subsystem that is malfunctioning. An unprotected resource cannot defend against use (or misuse) by an unauthorized or incompetent user. A protection-oriented system provides a means to distinguish between authorized and unauthorized usage, as we discuss in Chapter 18.

3.1.8 Command-Interpreter System

One of the most important systems programs for an operating system is the **command interpreter**, which is the interface between the user and the operating system. Some operating systems include the command interpreter in the kernel. Other operating systems, such as MS-DOS and UNIX, treat the command interpreter as a special program that is running when a job is initiated, or when a user first logs on (on time-sharing systems).

Many commands are given to the operating system by **control statements**. When a new job is started in a batch system, or when a user logs on to a time-shared system, a program that reads and interprets control statements is executed automatically. This program is sometimes called the **control-card interpreter** or the **command-line interpreter**, and is often known as the **shell**. Its function is simple: To get the next command statement and execute it.

Operating systems are frequently differentiated in the area of the shell, with a user-friendly command interpreter making the system more agreeable to some users. One style of user-friendly interface is the mouse-based window-and-menu system used in the Macintosh and in Microsoft Windows. The mouse is moved to position the mouse pointer on images, or **icons**, on the screen that represent programs, files, and system functions. Depending on the mouse pointer's location, clicking a button on the mouse can invoke a program, select a file or directory—known as a **folder**—or pull down a menu that contains commands. More powerful, complex, and difficult-to-learn shells are appreciated by other users. In some of these shells, commands are typed on a keyboard and displayed on a screen or printing terminal, with the enter (or return) key signaling that a command is complete and is ready to be executed. The MS-DOS and UNIX shells operate in this way.

The command statements themselves deal with process creation and management, I/O handling, secondary-storage management, main-memory management, file-system access, protection, and networking.

3.2 ■ Operating-System Services

An operating system provides an environment for the execution of programs. It provides certain services to programs and to the users of those programs. The specific services provided, of course, differ from one operating system to another, but we can identify common classes. These operating-system services are provided for the convenience of the programmer, to make the programming task easier.

- **Program execution:** The system must be able to load a program into memory and to run that program. The program must be able to end its execution, either normally or abnormally (indicating error).

- **I/O operations:** A running program may require I/O. This I/O may involve a file or an I/O device. For specific devices, special functions may be desired (such as to rewind a tape drive, or to blank a CRT screen). For efficiency and protection, users usually cannot control I/O devices directly. Therefore, the operating system must provide a means to do I/O.
- **File-system manipulation:** The file system is of particular interest. Obviously, programs need to read and write files. Programs also need to create and delete files by name.
- **Communications:** In many circumstances, one process needs to exchange information with another process. Such communication can occur in two major ways. The first takes place between processes that are executing on the same computer; the second takes place between processes that are executing on different computer systems that are tied together by a computer network. Communications may be implemented via **shared memory**, or by the technique of **message passing**, in which packets of information are moved between processes by the operating system.
- **Error detection:** The operating system constantly needs to be aware of possible errors. Errors may occur in the CPU and memory hardware (such as a memory error or a power failure), in I/O devices (such as a parity error on tape, a connection failure on a network, or lack of paper in the printer), and in the user program (such as an arithmetic overflow, an attempt to access an illegal memory location, or a too-great use of CPU time). For each type of error, the operating system should take the appropriate action to ensure correct and consistent computing.

In addition, another set of operating-system functions exists not for helping the user, but for ensuring the efficient operation of the system itself. Systems with multiple users can gain efficiency by sharing the computer resources among the users.

- **Resource allocation:** When multiple users are logged on the system or multiple jobs are running at the same time, resources must be allocated to each of them. Many different types of resources are managed by the operating system. Some (such as CPU cycles, main memory, and file storage) may have special allocation code, whereas others (such as I/O devices) may have much more general request and release code. For instance, in determining how best to use the CPU, operating systems have CPU-scheduling routines that take into account the speed of the CPU, the jobs that must be executed, the number of registers available, and other factors. There might also be routines to allocate a tape drive for use by a job. One such routine locates an unused tape drive and marks an internal table to record the drive's new user. Another routine is used to clear that table.

These routines may also allocate plotters, modems, and other peripheral devices.

- **Accounting:** We want to keep track of which users use how many and which kinds of computer resources. This record keeping may be used for accounting (so that users can be billed) or simply for accumulating usage statistics. Usage statistics may be a valuable tool for researchers who wish to reconfigure the system to improve computing services.
- **Protection:** The owners of information stored in a multiuser computer system may want to control use of that information. When several disjointed processes execute concurrently, it should not be possible for one process to interfere with the others, or with the operating system itself. Protection involves ensuring that all access to system resources is controlled. **Security** of the system from outsiders is also important. Such security starts with each user having to authenticate himself to the system, usually by means of a password, to be allowed access to the resources. It extends to defending external I/O devices, including modems and network adapters, from invalid access attempts, and to recording all such connections for detection of break-ins. If a system is to be protected and secure, precautions must be instituted throughout it. A chain is only as strong as its weakest link.

3.3 ■ System Calls

System calls provide the interface between a process and the operating system. These calls are generally available as assembly-language instructions, and they are usually listed in the various manuals used by the assembly-language programmers.

Certain systems allow system calls to be made directly from a higher-level language program, in which case the calls normally resemble predefined function or subroutine calls. They may generate a call to a special run-time routine that makes the system call, or the system call may be generated directly in-line.

Several languages—such as C, C++, and Perl—have been defined to replace assembly language for systems programming. These languages allow system calls to be made directly. For example, UNIX system calls may be invoked directly from a C or C++ program. System calls for modern Microsoft Windows platforms are part of the **Win32 application programmer interface (API)**, which is available for use by all the compilers written for Microsoft Windows.

As an example of how system calls are used, consider writing a simple program to read data from one file and to copy them to another file. The first input that the program will need is the names of the two files: the input file and the output file. These names can be specified in many ways, depending on the operating-system design. One approach is for the program to ask the

user for the names of the two files. In an interactive system, this approach will require a sequence of system calls, first to write a prompting message on the screen, and then to read from the keyboard the characters that define the two files. On mouse-based window-and-menu systems, a menu of file names is usually displayed in a window. The user can then use the mouse to select the source name, and a similar window can be opened for the destination name to be specified.

Once the two file names are obtained, the program must open the input file and create the output file. Each of these operations requires another system call and may encounter possible error conditions. When the program tries to open the input file, it may find that no file of that name exists or that the file is protected against access. In these cases, the program should print a message on the console (another sequence of system calls), and then terminate abnormally (another system call). If the input file exists, then we must create a new output file. We may find an output file with the same name. This situation may cause the program to abort (a system call), or we may delete the existing file (another system call) and create a new one (another system call). In an interactive system, another option is to ask the user (a sequence of system calls to output the prompting message and to read the response from the keyboard) whether to replace the existing file or to abort the program.

Now that both files are set up, we enter a loop that reads from the input file (a system call) and writes to the output file (another system call). Each `read` and `write` must return status information regarding various possible error conditions. On input, the program may find that the end of the file has been reached, or that a hardware failure occurred in the `read` (such as a parity error). On output, various errors may occur, depending on the output device (such as no more disk space, physical end of tape, printer out of paper).

Finally, after the entire file is copied, the program may close both files (another system call), write a message to the console (more system calls), and finally terminate normally (the final system call). As we can see, even simple programs may make heavy use of the operating system.

Most users never see this level of detail, however. The run-time support system (the set of functions built into libraries included with a compiler) for most programming languages provides a much simpler interface. For example, the `count` statement in C++ is probably compiled into a call to a run-time support routine that issues the necessary system calls, checks for errors, and finally returns to the user program. Thus, most of the details of the operating-system interface are hidden from the programmer by the compiler and by the run-time support package.

System calls occur in different ways, depending on the computer in use. Often, more information is required than simply the identity of the desired system call. The exact type and amount of information vary according to the particular operating system and call. For example, to get input, we may need to specify the file or device to use as the source, and the address and length of

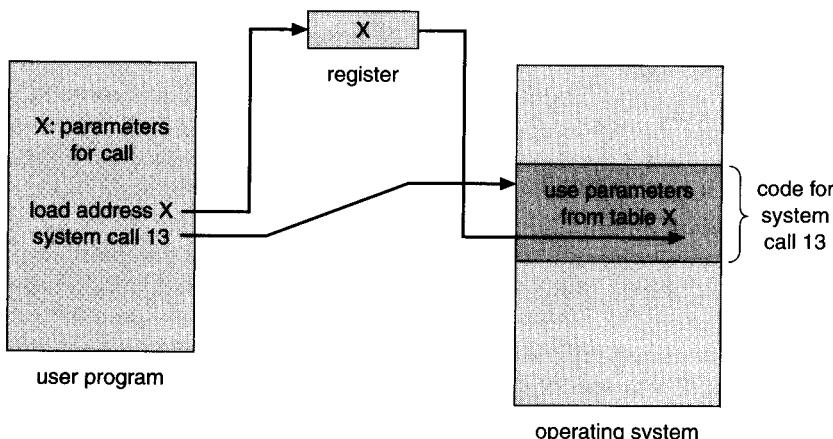


Figure 3.1 Passing of parameters as a table.

the memory buffer into which the input should be read. Of course, the device or file and length may be implicit in the call.

Three general methods are used to pass parameters to the operating system. The simplest approach is to pass the parameters in *registers*. In some cases, however, there may be more parameters than registers. In these cases, the parameters are generally stored in a *block* or table in memory, and the address of the block is passed as a parameter in a register (Figure 3.1). This is the approach taken by Linux. Parameters can also be placed, or *pushed*, onto the *stack* by the program, and *popped off* the stack by the operating system. Some operating systems prefer the block or stack methods, because those approaches do not limit the number or length of parameters being passed.

System calls can be grouped roughly into five major categories: **process control**, **file management**, **device management**, **information maintenance**, and **communications**. In Sections 3.3.1 through 3.3.5, we discuss briefly the types of system calls that may be provided by an operating system. Most of these system calls support, or are supported by, concepts and functions that are discussed in later chapters. Figure 3.2 summarizes the types of system calls normally provided by an operating system.

3.3.1 Process Control

A running program needs to be able to halt its execution either normally (*end*) or abnormally (*abort*). If a system call is made to terminate the currently running program abnormally, or if the program runs into a problem and causes an error trap, a dump of memory is sometimes taken and an error message generated. The dump is written to disk and may be examined by a **debugger** to determine the cause of the problem. Under either normal or abnormal

- Process control
 - end, abort
 - load, execute
 - create process, terminate process
 - get process attributes, set process attributes
 - wait for time
 - wait event, signal event
 - allocate and free memory
- File management
 - create file, delete file
 - open, close
 - read, write, reposition
 - get file attributes, set file attributes
- Device management
 - request device, release device
 - read, write, reposition
 - get device attributes, set device attributes
 - logically attach or detach devices
- Information maintenance
 - get time or date, set time or date
 - get system data, set system data
 - get process, file, or device attributes
 - set process, file, or device attributes
- Communications
 - create, delete communication connection
 - send, receive messages
 - transfer status information
 - attach or detach remote devices

Figure 3.2 Types of system calls.

circumstances, the operating system must transfer control to the command interpreter. The command interpreter then reads the next command. In an interactive system, the command interpreter simply continues with the next command; it is assumed that the user will issue an appropriate command to respond to any error. In a batch system, the command interpreter usually terminates the entire job and continues with the next job. Some systems allow control cards to indicate special recovery actions in case an error occurs. If the program discovers an error in its input and wants to terminate abnormally, it may also want to define an error level. More severe errors can be indicated by a higher-level error parameter. It is then possible to combine normal and abnormal termination by defining a normal termination as error at level 0. The command interpreter or a following program can use this error level to determine the next action automatically.

A process or job executing one program may want to load and execute another program. This feature allows the command interpreter to execute a program as directed by, for example, a user command, the click of a mouse, or a batch command. An interesting question is where to return control when the loaded program terminates. This question is related to the problem of whether the existing program is lost, saved, or allowed to continue execution concurrently with the new program.

If control returns to the existing program when the new program terminates, we must save the memory image of the existing program; thus, we have effectively created a mechanism for one program to call another program. If both programs continue concurrently, we have created a new job or process to be multiprogrammed. Often, system calls exist specifically for this purpose (*create process* or *submit job*).

If we create a new job or process, or perhaps even a set of jobs or processes, we should be able to control its execution. This control requires the ability to determine and reset the attributes of a job or process, including the job's priority, its maximum allowable execution time, and so on (*get process attributes* and *set process attributes*). We may also want to terminate a job or process that we created (*terminate process*) if we find that it is incorrect or is no longer needed.

Having created new jobs or processes, we may need to wait for them to finish their execution. We may want to wait for a certain amount of time (*wait time*); more likely, we may want to wait for a specific event to occur (*wait event*). The jobs or processes should then signal when that event has occurred (*signal event*). System calls of this type, dealing with the coordination of concurrent processes, are discussed in great detail in Chapter 7.

Another set of system calls is helpful in debugging a program. Many systems provide system calls to dump memory. This provision is useful for debugging. A program trace lists each instruction as it is executed; it is provided by fewer systems. Even microprocessors provide a CPU mode known as *single step*, in which a trap is executed by the CPU after every instruction.

The trap is usually caught by a debugger, which is a system program designed to aid the programmer in finding and correcting bugs.

Many operating systems provide a time profile of a program. It indicates the amount of time that the program executes at a particular location or set of locations. A time profile requires either a tracing facility or regular timer interrupts. At every occurrence of the timer interrupt, the value of the program counter is recorded. With sufficiently frequent timer interrupts, a statistical picture of the time spent on various parts of the program can be obtained.

Process and job control have so many facets and variations that we shall use examples to clarify these concepts. The MS-DOS operating system is an example of a single-tasking system, which has a command interpreter that is invoked when the computer is started (Figure 3.3(a)). Because MS-DOS is single-tasking, it uses a simple method to run a program and does not create a new process. It loads the program into memory, writing over most of itself to give the program as much memory as possible (Figure 3.3(b)). It then sets the instruction pointer to the first instruction of the program. The program then runs and either an error causes a trap, or the program executes a system call to terminate. In either case, the error code is saved in the system memory for later use. Following this action, the small portion of the command interpreter that was not overwritten resumes execution. Its first task is to reload the rest of the command interpreter from disk. Once this task is accomplished, the command interpreter makes the previous error code available to the user or to the next program.

Although the MS-DOS operating system does not have general multitasking capabilities, it does provide a method for limited concurrent execution. A TSR program is a program that “hooks an interrupt” and then exits with the

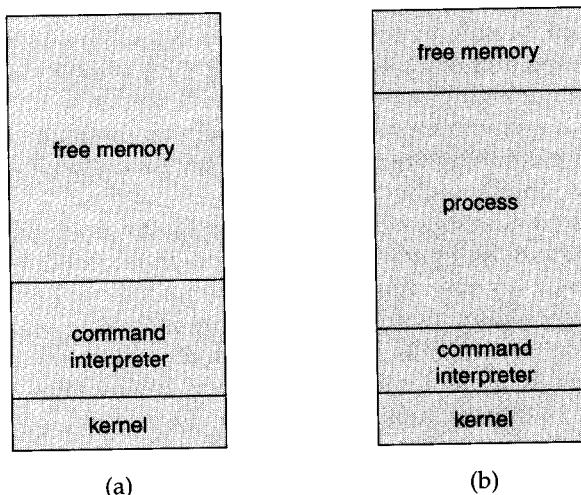


Figure 3.3 MS-DOS execution. (a) At system startup. (b) Running a program.

terminate and stay resident system call. For instance, it can hook the clock interrupt by placing the address of one of its subroutines into the list of interrupt routines to be called when the system timer is triggered. This way, the TSR routine will be executed several times per second, at each clock tick. The **terminate and stay resident** system call causes MS-DOS to reserve the space occupied by the TSR, so it will not be overwritten when the command interpreter is reloaded.

FreeBSD is an example of a multitasking system. When a user logs on to the system, the shell (or command interpreter) of the user's choice is run. This shell is similar to the MS-DOS shell in that it accepts commands and executes programs that the user requests. However, since FreeBSD is a multitasking system, the command interpreter may continue running while another program is executed (Figure 3.4). To start a new process, the shell executes a **fork** system call. Then, the selected program is loaded into memory via an **exec** system call, and the program is then executed. Depending on the way the command is issued, the shell then either waits for the process to finish, or runs the process "in the background." In the latter case, the shell immediately requests another command. When a process is running in the background, it cannot receive input directly from the keyboard, because the shell is using this resource. I/O is therefore done through files, or through a window-and-menu interface. Meanwhile, the user is free to ask the shell to run other programs, to monitor the progress of the running process, to change that program's priority, and so on. When the process is done, it executes an **exit** system call to terminate, returning to the invoking process a status code of 0, or a nonzero error code. This status (or error) code is then available to the shell or other programs. Processes are discussed in Chapter 4.

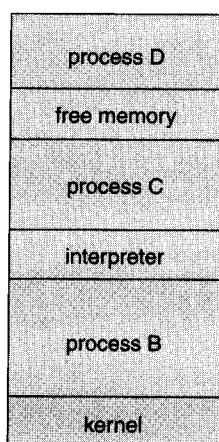


Figure 3.4 UNIX running multiple programs.

3.3.2 File Management

We will discuss the file system in more detail in Chapters 11 and 12. We can identify several common system calls dealing with files, however.

We first need to be able to `create` and `delete` files. Either system call requires the name of the file and perhaps some of the file's attributes. Once the file is created, we need to `open` it and to use it. We may also `read`, `write`, or `reposition` (rewind or skip to the end of the file, for example). Finally, we need to `close` the file, indicating that we are no longer using it.

We may need these same sets of operations for directories if we have a directory structure for organizing files in the file system. In addition, for either files or directories, we need to be able to determine the values of various attributes, and perhaps to reset them if necessary. File attributes include the file name, a file type, protection codes, accounting information, and so on. At least two system calls, `get file attribute` and `set file attribute`, are required for this function. Some operating systems provide many more calls.

3.3.3 Device Management

A program, as it is running, may need additional resources to proceed. Additional resources may be more memory, tape drives, access to files, and so on. If the resources are available, they can be granted, and control can be returned to the user program; otherwise, the program will have to wait until sufficient resources are available.

Files can be thought of as abstract or virtual devices. Thus, many of the system calls for files are also needed for devices. If the system has multiple users, however, we must first `request` the device, to ensure exclusive use of it. After we are finished with the device, we must `release` it. These functions are similar to the `open` and `close` system calls for files.

Once the device has been requested (and allocated to us), we can `read`, `write`, and (possibly) `reposition` the device, just as we can with ordinary files. In fact, the similarity between I/O devices and files is so great that many operating systems, including UNIX and MS-DOS, merge the two into a combined **file-device structure**. In this case, I/O devices are identified by special file names.

3.3.4 Information Maintenance

Many system calls exist simply for the purpose of transferring information between the user program and the operating system. For example, most systems have a system call to return the current `time` and `date`. Other system calls may return information about the system, such as the number of current users, the version number of the operating system, the amount of free memory or disk space, and so on.

In addition, the operating system keeps information about all its processes, and there are system calls to access this information. Generally, there are also calls to reset the process information (get process attributes and set process attributes). In Section 4.1.3, we discuss what information is normally kept.

3.3.5 Communication

There are two common models of communication. In the **message-passing model**, information is exchanged through an interprocess-communication facility provided by the operating system. Before communication can take place, a connection must be opened. The name of the other communicator must be known, be it another process on the same CPU, or a process on another computer connected by a communications network. Each computer in a network has a **host name**, such as an IP name, by which it is commonly known. Similarly, each process has a **process name**, which is translated into an equivalent identifier by which the operating system can refer to it. The `get hostid` and `get processid` system calls do this translation. These identifiers are then passed to the general-purpose `open` and `close` calls provided by the file system, or to specific `open connection` and `close connection` system calls, depending on the system's model of communications. The recipient process usually must give its permission for communication to take place with an `accept connection` call. Most processes that will be receiving connections are special-purpose **daemons**—systems programs provided for that purpose. They execute a `wait for connection` call and are awakened when a connection is made. The source of the communication, known as the *client*, and the receiving daemon, known as a *server*, then exchange messages by `read message` and `write message` system calls. The `close connection` call terminates the communication.

In the **shared-memory model**, processes use `map memory` system calls to gain access to regions of memory owned by other processes. Recall that, normally, the operating system tries to prevent one process from accessing another process' memory. Shared memory requires that several processes agree to remove this restriction. They may then exchange information by reading and writing data in these shared areas. The form of the data and the location are determined by these processes and are not under the operating system's control. The processes are also responsible for ensuring that they are not writing to the same location simultaneously. Such mechanisms are discussed in Chapter 7. We will also look at a variation of the process model—**threads**—that shares memory by default. Threads will be covered in Chapter 5.

Both of these methods are common in operating systems, and some systems even implement both. Message passing is useful when smaller numbers of data need to be exchanged, because no conflicts need to be avoided. It is also easier to implement than is shared memory for intercomputer communication. Shared

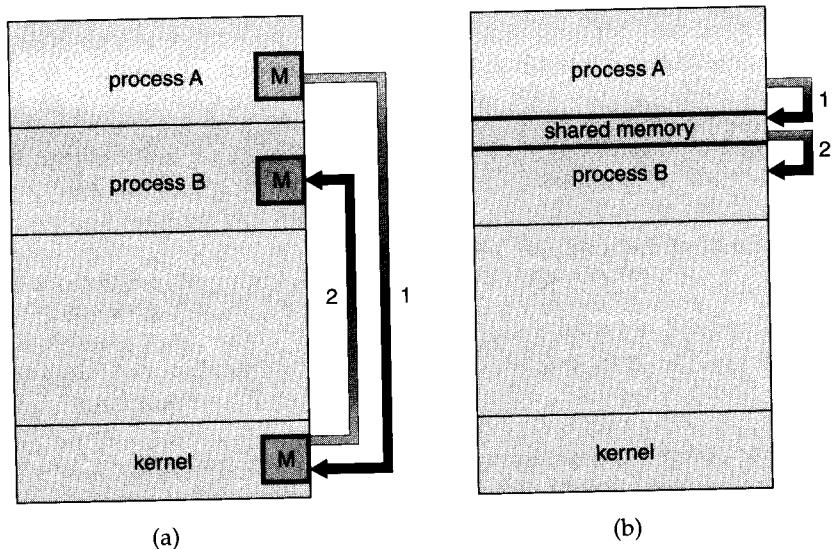


Figure 3.5 Communications models. (a) Msg passing. (b) Shared memory.

memory allows maximum speed and convenience of communication, as it can be done at memory speeds when within a computer. Problems exist, however, in the areas of protection and synchronization. The two communications models are contrasted in Figure 3.5.

3.4 ■ System Programs

Another aspect of a modern system is the collection of system programs. Recall Figure 1.1, which depicted the logical computer hierarchy. At the lowest level is hardware. Next is the operating system, then the system programs, and finally the application programs. System programs provide a convenient environment for program development and execution. Some of them are simply user interfaces to system calls; others are considerably more complex. They can be divided into these categories:

- **File management:** These programs create, delete, copy, rename, print, dump, list, and generally manipulate files and directories.
- **Status information:** Some programs simply ask the system for the date, time, amount of available memory or disk space, number of users, or similar status information. That information is then formatted, and is printed to the terminal or other output device or file.

- **File modification:** Several text editors may be available to create and modify the content of files stored on disk or tape.
- **Programming-language support:** Compilers, assemblers, and interpreters for common programming languages (such as C, C++, Java, Visual Basic, and PERL) are often provided to the user with the operating system. Some of these programs are now priced and provided separately.
- **Program loading and execution:** Once a program is assembled or compiled, it must be loaded into memory to be executed. The system may provide absolute loaders, relocatable loaders, linkage editors, and overlay loaders. Debugging systems for either higher-level languages or machine language are needed also.
- **Communications:** These programs provide the mechanism for creating virtual connections among processes, users, and different computer systems. They allow users to send messages to one another's screens, to browse web pages, to send electronic-mail messages, to log in remotely, or to transfer files from one machine to another.

Most operating systems are supplied with programs that solve common problems, or perform common operations. Such programs include web browsers, word processors and text formatters, spreadsheets, database systems, compiler compilers, plotting and statistical-analysis packages, and games. These programs are known as **system utilities** or **application programs**.

Perhaps the most important system program for an operating system is the command interpreter, the main function of which is to get and execute the next user-specified command.

Many of the commands given at this level manipulate files: create, delete, list, print, copy, execute, and so on. These commands can be implemented in two general ways. In one approach, the command interpreter itself contains the code to execute the command. For example, a command to delete a file may cause the command interpreter to jump to a section of its code that sets up the parameters and makes the appropriate system call. In this case, the number of commands that can be given determines the size of the command interpreter, since each command requires its own implementing code.

An alternative approach—used by UNIX, among other operating systems—implements most commands by system programs. In this case, the command interpreter does not understand the command in any way; it merely uses the command to identify a file to be loaded into memory and executed. Thus, the UNIX command to delete a file

`rm G`

would search for a file called `rm`, load the file into memory, and execute it with the parameter `G`. The function associated with the `rm` command would

be defined completely by the code in the file `rm`. In this way, programmers can add new commands to the system easily by creating new files with the proper names. The command-interpreter program, which can be small, does not have to be changed for new commands to be added.

This approach to the design of a command interpreter has problems. First, because the code to execute a command is a separate system program, the operating system must provide a mechanism for passing parameters from the command interpreter to the system program. This task can often be clumsy, because the command interpreter and the system program may not be in memory at the same time, and the parameter list can be extensive. Also, it is slower to load a program and to execute it than simply to jump to another section of code within the current program.

Another problem is that the interpretation of the parameters is left up to the programmer of the system program. Thus, parameters may be provided inconsistently across programs that appear similar to the user, but were written at different times by different programmers.

The view of the operating system seen by most users is thus defined by the system programs, rather than by the actual system calls. Think of using a PC. When your computer is running the Microsoft Windows operating system, you might see a command-line MS-DOS shell, or the graphical window-and-menu interface. Both use the same set of system calls, but the system calls look different and act in different ways. Consequently, your view may be substantially removed from the actual system structure. The design of a useful and friendly user interface is therefore not a direct function of the operating system. In this book, we shall concentrate on the fundamental problems of providing adequate service to user programs. From the point of view of the operating system, we do not distinguish between user programs and system programs.

3.5 ■ System Structure

A system as large and complex as a modern operating system must be engineered carefully if it is to function properly and be modified easily. A common approach is to partition the task into small components, rather than have one monolithic system. Each of these modules should be a well-defined portion of the system, with carefully defined inputs, outputs, and function. We have already discussed briefly the common components of operating systems (Section 3.1). In this section, we discuss the way that these components are interconnected and melded into a kernel.

3.5.1 Simple Structure

Many commercial systems do not have a well-defined structure. Frequently, such operating systems started as small, simple, and limited systems, and then

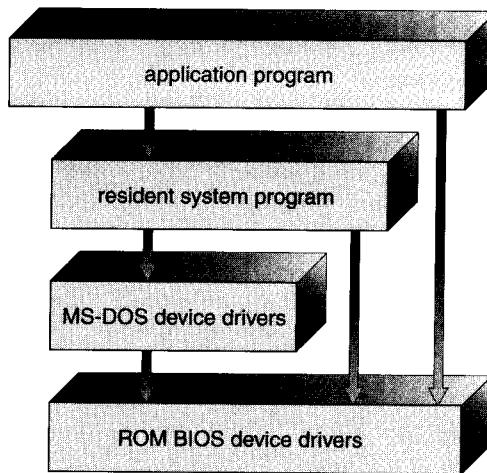


Figure 3.6 MS-DOS layer structure.

grew beyond their original scope. MS-DOS is an example of such a system. It was originally designed and implemented by a few people who had no idea that it would become so popular. It was written to provide the most functionality in the least space (because of the limited hardware on which it ran), so it was not divided into modules carefully. Figure 3.6 shows its structure.

UNIX is another system that was initially limited by hardware functionality. It consists of two separable parts: the kernel and the system programs. The kernel is further separated into a series of interfaces and device drivers, which were added and expanded over the years as UNIX evolved. We can view the traditional UNIX operating system as being layered as shown in Figure 3.7. Everything below the system-call interface and above the physical hardware is the kernel. The kernel provides the file system, CPU scheduling, memory management, and other operating-system functions through system calls. Taken in sum, that is an enormous amount of functionality to be combined into one level. This makes UNIX difficult to enhance, as changes in one section could adversely affect other areas.

System calls define the API to UNIX; the set of system programs commonly available defines the user interface. The programmer and user interfaces define the context that the kernel must support.

New versions of UNIX are designed to use more advanced hardware. Given proper hardware support, operating systems may be broken into pieces that are smaller and more appropriate than are those allowed by the original MS-DOS or UNIX systems. The operating system can then retain much greater control over the computer and over the applications that make use of that computer. Implementers have more freedom to make changes to the inner workings of the system and in the creation of modular operating systems. Under the top-

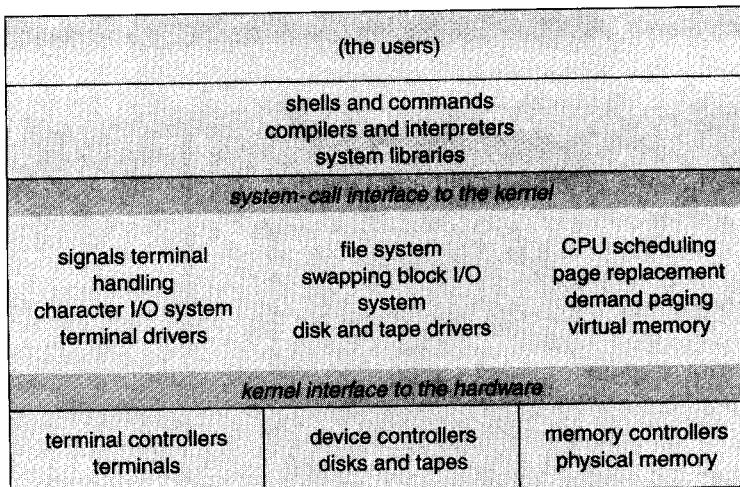


Figure 3.7 UNIX system structure.

down approach, the overall functionality and features are determined and are separated into components. This separation allows programmers to hide information; they are therefore free to implement the low-level routines as they see fit, provided that the external interface of the routine stays unchanged and that the routine itself performs the advertised task.

3.5.2 Layered Approach

The modularization of a system can be done in many ways. One method is the **layered approach**, in which the operating system is broken up into a number of layers (or levels), each built on top of lower layers. The bottom layer (layer 0) is the hardware; the highest (layer N) is the user interface.

An operating-system layer is an implementation of an abstract object that is the encapsulation of data, and of the operations that can manipulate those data. A typical operating-system layer—say layer M —is depicted in Figure 3.8. It consists of data structures and a set of routines that can be invoked by higher-level layers. Layer M , in turn, can invoke operations on lower-level layers.

The main advantage of the layered approach is **modularity**. The layers are selected such that each uses functions (or operations) and services of only lower-level layers. This approach simplifies debugging and system verification. The first layer can be debugged without any concern for the rest of the system, because, by definition, it uses only the basic hardware (which is assumed correct) to implement its functions. Once the first layer is debugged, its correct functioning can be assumed while the second layer is debugged, and so on. If an error is found during the debugging of a particular layer, the error must

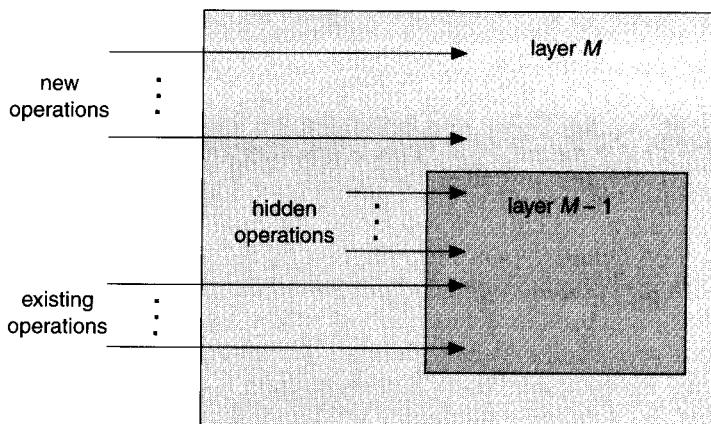


Figure 3.8 An operating-system layer.

be on that layer, because the layers below it are already debugged. Thus, the design and implementation of the system are simplified when the system is broken down into layers.

Each layer is implemented with only those operations provided by lower-level layers. A layer does not need to know how these operations are implemented; it needs to know only what these operations do. Hence, each layer hides the existence of certain data structures, operations, and hardware from higher-level layers.

The major difficulty with the layered approach involves the careful definition of the layers, because a layer can use only those layers below it. For example, the device driver for the disk space used by virtual-memory algorithms must be at a level lower than that of the memory-management routines, because memory management requires the ability to use the disk space.

Other requirements may not be so obvious. The backing-store driver would normally be above the CPU scheduler, because the driver may need to wait for I/O and the CPU can be rescheduled during this time. However, on a large system, the CPU scheduler may have more information about all the active processes than can fit in memory. Therefore, this information may need to be swapped in and out of memory, requiring the backing-store driver routine to be below the CPU scheduler.

A final problem with layered implementations is that they tend to be less efficient than other types. For instance, when a user program executes an I/O operation, it executes a system call that is trapped to the I/O layer, which calls the memory-management layer, which in turn calls the CPU-scheduling layer, which is then passed to the hardware. At each layer, the parameters may be modified, data may need to be passed, and so on. Each layer adds overhead to the system call; the net result is a system call that takes longer than does one on a nonlayered system.

These limitations have caused a small backlash against layering in recent years. Fewer layers with more functionality are being designed, providing most of the advantages of modularized code while avoiding the difficult problems of layer definition and interaction. For instance, OS/2 is a descendant of MS-DOS that adds multitasking and dual-mode operation, as well as other new features. Because of this added complexity and the more powerful hardware for which OS/2 was designed, the system was implemented in a more layered fashion. Compare the MS-DOS structure (Figure 3.6) with that shown in Figure 3.9; from both the system-design and implementation standpoints, OS/2 has the advantage. For instance, direct user access to low-level facilities is not allowed, providing the operating system with more control over the hardware and more knowledge of which resources each user program is using.

As a further example, consider the history of Windows NT. The first release had a highly layer-oriented organization; however, it delivered low performance compared to that of Windows 95. Windows NT 4.0 partially redressed the performance problem by moving layers from user space to kernel space and more closely integrating them.

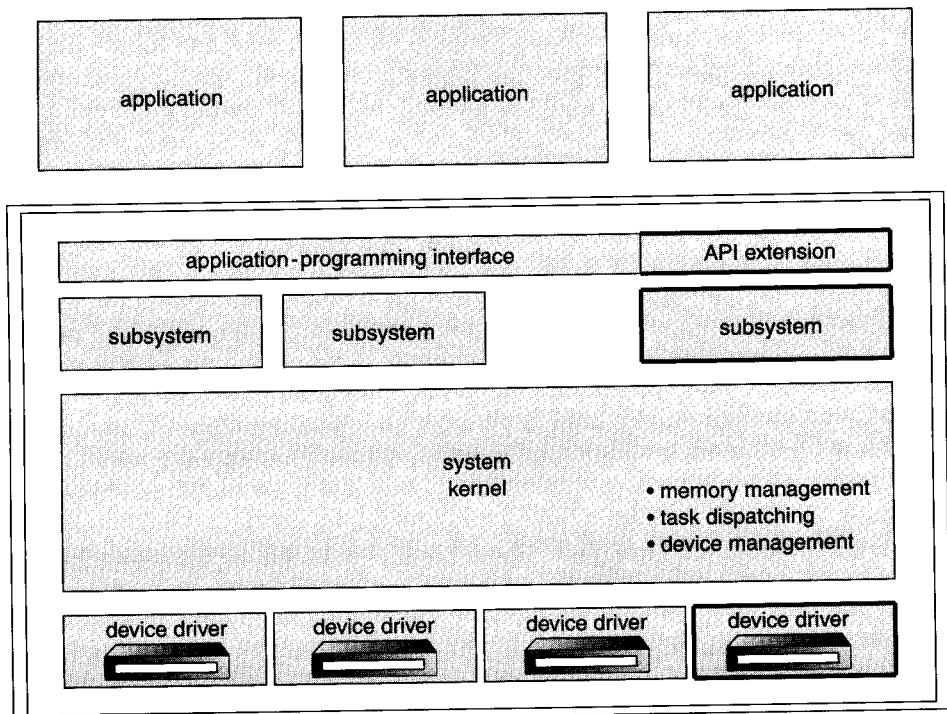


Figure 3.9 OS/2 layer structure.

3.5.3 Microkernels

As the UNIX operating system expanded, the kernel became large and difficult to manage. In the mid-1980s, researchers at Carnegie Mellon University developed an operating system called Mach that modularizes the kernel using the **microkernel** approach. This method structures the operating system by removing all nonessential components from the kernel, and implementing them as system- and user-level programs. The result is a smaller kernel. There is little consensus regarding which services should remain in the kernel and which should be implemented in user space. In general, however, microkernels typically provide minimal process and memory management, in addition to a communication facility.

The main function of the microkernel is to provide a communication facility between the client program and the various services that are also running in user space. Communication is provided by message passing, which was described in Section 3.3.5. For example, if the client program wishes to access a file, it must interact with the file server. The client program and the service never interact directly. Rather, they communicate indirectly by exchanging messages with the microkernel.

The benefits of the microkernel approach include the ease of extending the operating system. All new services are added to user space and consequently do not require modification of the kernel. When the kernel does have to be modified, the changes tend to be fewer, because the microkernel is a smaller kernel. The resulting operating system is easier to port from one hardware design to another. The microkernel also provides more security and reliability, since most services are running as user—rather than kernel—processes. If a service fails, the rest of the operating system remains untouched.

Several contemporary operating systems have used the microkernel approach. Tru64 UNIX (formerly Digital UNIX) provides a UNIX interface to the user, but it is implemented with a Mach kernel. The Mach kernel maps UNIX system calls into messages to the appropriate user-level services. The Apple MacOS X Server operating system is based on the Mach kernel.

QNX is a real-time operating system that is also based upon the microkernel design. The QNX microkernel provides services for message passing and process scheduling. It also handles low-level network communication and hardware interrupts. All other services in QNX are provided by standard processes that run outside the kernel in user mode.

Windows NT uses a hybrid structure. We have already mentioned that part of the Windows NT architecture uses layering. Windows NT is designed to run various applications, including Win32 (native Windows applications), OS/2, and POSIX. It provides a server that runs in user space for each application type. Client programs for each application type also run in user space. The kernel coordinates the message passing between client applications and application servers. The client-server structure of Windows NT is depicted in Figure 3.10.

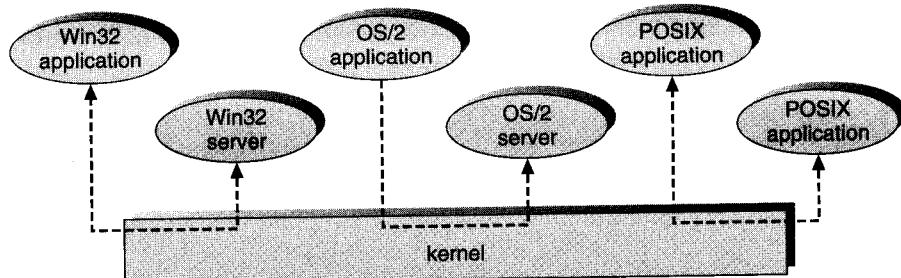


Figure 3.10 Windows NT client-server structure.

3.6 ■ Virtual Machines

Conceptually, a computer system is made up of layers. The hardware is the lowest level in all such systems. The kernel running at the next level uses the hardware instructions to create a set of system calls for use by outer layers. The system programs above the kernel are therefore able to use either system calls or hardware instructions, and in some ways these programs do not differentiate between these two. Thus, although they are accessed differently, they both provide functionality that the program can use to create even more advanced functions. System programs, in turn, treat the hardware and the system calls as though they were both at the same level.

Some systems carry this scheme a step further by allowing the system programs to be called easily by the application programs. As before, although the system programs are at a level higher than that of the other routines, the application programs may view everything under them in the hierarchy as though the latter were part of the machine itself. This layered approach is taken to its logical conclusion in the concept of a **virtual machine**. The VM operating system for IBM systems is the best example of the virtual-machine concept, because IBM pioneered the work in this area.

By using CPU scheduling (Chapter 6) and virtual-memory techniques (Chapter 10), an operating system can create the illusion that a process has its own processor with its own (virtual) memory. Of course, normally, the process has additional features, such as system calls and a file system, that are not provided by the bare hardware. The virtual-machine approach, on the other hand, does not provide any additional functionality, but rather provides an interface that is *identical* to the underlying bare hardware. Each process is provided with a (virtual) copy of the underlying computer (Figure 3.11).

The physical computer shares resources to create the virtual machines. CPU scheduling can share out the CPU to create the appearance that users have their own processors. Spooling and a file system can provide virtual card readers and virtual line printers. A normal user time-sharing terminal provides the function of the virtual-machine operator's console.

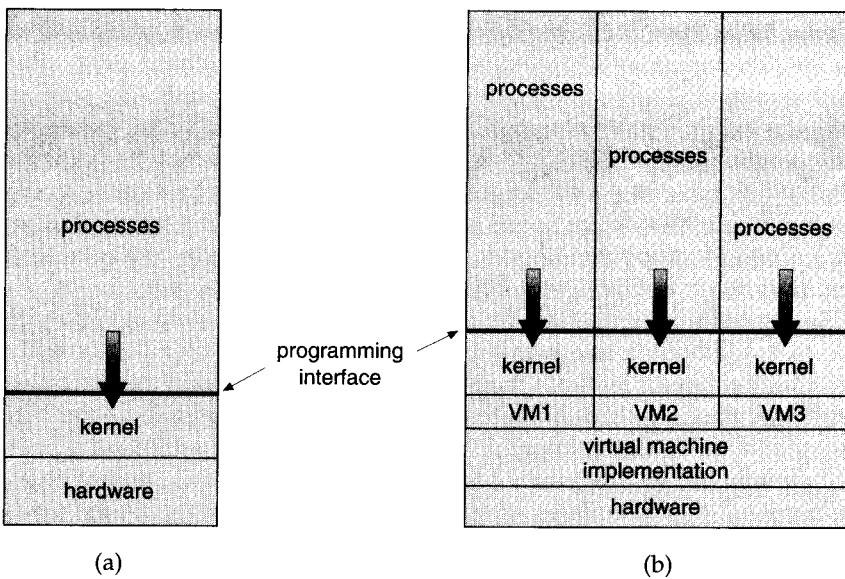


Figure 3.11 System models. (a) Nonvirtual machine. (b) Virtual machine.

A major difficulty with the virtual-machine approach involves disk systems. Suppose that the physical machine has three disk drives but wants to support seven virtual machines. Clearly, it cannot allocate a disk drive to each virtual machine. Remember that the virtual-machine software itself will need substantial disk space to provide virtual memory. The solution is to provide virtual disks, which are identical in all respects except size—termed *minidisks* in IBM's VM operating system. The system implements each minidisk by allocating as many tracks on the physical disks as the minidisk needs. Obviously, the sum of the sizes of all minidisks must be smaller than the size of the physical disk space available.

Users thus are given their own virtual machines. They can then run any of the operating systems or software packages that are available on the underlying machine. For the IBM VM system, a user normally runs CMS—a single-user interactive operating system. The virtual-machine software is concerned with multiprogramming multiple virtual machines onto a physical machine, but it does not need to consider any user-support software. This arrangement may provide a useful partitioning into two smaller pieces of the problem of designing a multiuser interactive system.

3.6.1 Implementation

Although the virtual-machine concept is useful, it is difficult to implement. Much work is required to provide an *exact* duplicate of the underlying machine.

Remember that the underlying machine has two modes: user mode and monitor mode. The virtual-machine software can run in monitor mode, since it is the operating system. The virtual machine itself can execute in only user mode. Just as the physical machine has two modes, however, so must the virtual machine. Consequently, we must have a virtual user mode and a virtual monitor mode, both of which run in a physical user mode. Those actions that cause a transfer from user mode to monitor mode on a real machine (such as a system call or an attempt to execute a privileged instruction) must also cause a transfer from virtual user mode to virtual monitor mode on a virtual machine.

This transfer can generally be done fairly easily. When a system call, for example, is made by a program running on a virtual machine, in virtual user mode, it will cause a transfer to the virtual-machine monitor in the real machine. When the virtual-machine monitor gains control, it can change the register contents and program counter for the virtual machine to simulate the effect of the system call. It can then restart the virtual machine, noting that it is now in virtual monitor mode. If the virtual machine then tries, for example, to read from its virtual card reader, it will execute a privileged I/O instruction. Because the virtual machine is running in physical user mode, this instruction will trap to the virtual-machine monitor. The virtual-machine monitor must then simulate the effect of the I/O instruction. First, it finds the spooled file that implements the virtual card reader. Then, it translates the `read` of the virtual card reader into a `read` on the spooled disk file, and transfers the next virtual “card image” into the virtual memory of the virtual machine. Finally, it can restart the virtual machine. The state of the virtual machine has been modified exactly as though the I/O instruction had been executed with a real card reader for a real machine executing in a real monitor mode.

The major difference, of course, is time. Whereas the real I/O might have taken 100 milliseconds, the virtual I/O might take less time (because it is spooled) or more time (because it is interpreted). In addition, the CPU is being multiprogrammed among many virtual machines, further slowing down the virtual machines in unpredictable ways. In the extreme case, it may be necessary to simulate all instructions to provide a true virtual machine. VM works for IBM machines because normal instructions for the virtual machines can execute directly on the hardware. Only the privileged instructions (needed mainly for I/O) must be simulated and hence execute more slowly.

3.6.2 Benefits

There are two primary advantages to using virtual machines. First, by completely protecting system resources, the virtual machine provides a robust level of security. Second, the virtual machine allows system development to be done without disrupting normal system operation.

Each virtual machine is completely isolated from all other virtual machines, so we have no security problems as the various system resources are completely

protected. For example, untrusted applications downloaded from the Internet could be run within a separate virtual machine. A disadvantage of this environment is that there is no *direct* sharing of resources. Two approaches to provide sharing have been implemented. First, it is possible to share a minidisk. This scheme is modeled after a physical shared disk, but is implemented by software. With this technique, files can be shared. Second, it is possible to define a network of virtual machines, each of which can send information over the virtual communications network. Again, the network is modeled after physical communication networks, but it is implemented in software.

Such a virtual-machine system is a perfect vehicle for operating-systems research and development. Normally, changing an operating system is a difficult task. Because operating systems are large and complex programs, a change in one part may cause obscure bugs in some other part. The power of the operating system makes this situation particularly dangerous. Because the operating system executes in monitor mode, a wrong change in a pointer could cause an error that would destroy the entire file system. Thus, it is necessary to test all changes to the operating system carefully.

The operating system, however, runs on and controls the entire machine. Therefore, the current system must be stopped and taken out of use while changes are made and tested. This period is commonly called **system-development time**. Since it makes the system unavailable to users, system-development time is often scheduled late at night or on weekends, when system load is low.

A virtual-machine system can eliminate much of this problem. System programmers are given their own virtual machine, and system development is done on the virtual machine instead of on the actual physical machine. Normal system operation seldom needs to be disrupted for system development. Despite these advantages, little improvement on the technique had been made until recently.

Virtual machines are increasing in popularity as a means of solving system compatibility problems. For instance, thousands of applications are available for Microsoft Windows on Intel CPU-based systems. Computer vendors such as Sun Microsystems use other, faster processors, but would like their customers to be able to run these Windows applications. The solution is to create a virtual Intel machine on top of the native processor. A Windows program is run in this environment, and its Intel instructions are translated into the native instruction set. Microsoft Windows is also run in this virtual machine, so the program can make its system calls as usual. The net result is a program that appears to be running on an Intel-based system but is really executing on a different processor. If the processor is sufficiently fast, the Windows program will run quickly, even though every instruction is being translated into several native instructions for execution. Similarly, the PowerPC-based Apple Macintosh includes a Motorola 68000 virtual machine to allow execution of binary codes that were written for the older 68000-based Macintosh. Unfortunately, the more

complex the machine being emulated, the more difficult it is to build an accurate virtual machine, and the slower that virtual machine runs.

A more recent example can be shown with the growth of the Linux operating system. Virtual machines now exist that allow Windows applications to run on Linux-based computers. The virtual machine runs both the Windows application and the Windows operating system.

One of the key features of Java is that it runs on a virtual machine, thereby allowing a Java program to run on any computer system that has a Java virtual machine.

3.6.3 Java

Java is a very popular object-oriented language introduced by Sun Microsystems in late 1995. In addition to a language specification and a large API library, Java also provides a specification for a **Java virtual machine (JVM)**.

Java objects are specified with the `class` construct; a Java program consists of one or more classes. For each Java class, the Java compiler produces an architecture-neutral **bytecode** output (`.class`) file that will run on any implementation of the JVM.

The JVM is a specification for an abstract computer. The JVM consists of a **class loader**, a **class verifier**, and a Java interpreter that executes the architecture-neutral bytecodes. The class loader loads `.class` files from both the Java program and the Java API for execution by the Java interpreter. After a class is loaded, the verifier checks that the class file is valid Java bytecode and does not overflow or underflow the stack. It also ensures that the bytecode does not perform pointer arithmetic, which could potentially provide illegal memory access. If the class passes verification, it is run by the Java interpreter. The JVM also automatically manages memory by performing **garbage collection**—the practice of reclaiming memory from objects no longer in use and returning it to the system. Much research focuses on garbage collection algorithms for increasing the performance of Java programs in the virtual machine.

The Java interpreter may be a software module that interprets the bytecodes one at a time, or it may be a **just-in-time (JIT)** compiler that turns the architecture-neutral bytecodes into native machine language for the host computer. Most implementations of the JVM use a JIT compiler for enhanced performance. In other instances, the interpreter may be implemented in hardware that executes Java bytecodes natively. The JVM is presented in Figure 3.12.

The JVM makes it possible to develop programs that are architecture neutral and portable. The implementation of the JVM is specific for each system—such as Windows or UNIX—and it abstracts the system in a standard way to the Java program, providing a clean, architecture-neutral interface. This interface allows a `.class` file to run on any system that has implemented the JVM according to the specification.

Java takes advantage of the complete environment that a virtual machine implements. Its virtual-machine design provides a secure, efficient, object-

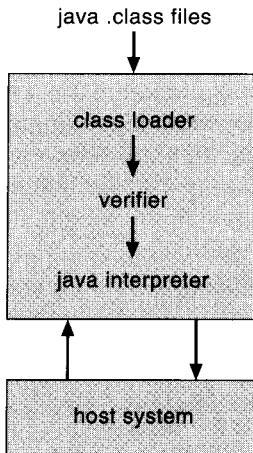


Figure 3.12 The Java virtual machine.

oriented, portable, and architecture-neutral platform on which to run Java programs.

3.7 ■ System Design and Implementation

In this section, we discuss problems we face when designing and implementing a system. No complete solutions to the design problems exist, but some approaches have been successful.

3.7.1 Design Goals

The first problem in designing a system is to define the goals and specifications of the system. At the highest level, the design of the system will be affected by the choice of hardware and type of system: batch, time shared, single user, multiuser, distributed, real time, or general purpose.

Beyond this highest design level, the requirements may be much harder to specify. The requirements can be divided into two basic groups: *user* goals and *system* goals.

Users desire certain obvious properties in a system: The system should be convenient and easy to use, easy to learn, reliable, safe, and fast. Of course, these specifications are not particularly useful in the system design, since there is no general agreement on how to achieve these goals.

A similar set of requirements can be defined by those people who must design, create, maintain, and operate the system: The operating system should be easy to design, implement, and maintain; it should be flexible, reliable, error free, and efficient. Again, these requirements are vague and have no general solution.

We have no unique solution to the problem of defining the requirements for an operating system. The wide range of systems shows that different requirements can result in a large variety of solutions for different environments. For example, the requirements for MS-DOS, a single-user system for microcomputers, must have been substantially different from those for MVS, the large multiuser, multiaccess operating system for IBM mainframes.

3.7.2 Mechanisms and Policies

The specification and design of an operating system is a highly creative task. Although no textbook can tell you how to do it, general **software engineering** principles do exist that are especially applicable to operating systems.

One important principle is the separation of *policy* from *mechanism*. **Mechanisms** determine *how* to do something; **policies** determine *what* will be done. For example, the timer construct (Section 2.5) is a mechanism for ensuring CPU protection, but deciding how long the timer is to be set for a particular user is a policy decision.

The separation of policy and mechanism is important for flexibility. Policies are likely to change across places or over time. In the worst case, each change in policy would require a change in the underlying mechanism. A general mechanism would be more desirable. A change in policy would then require redefinition of only certain parameters of the system. For instance, if, in one computer system, a policy decision is made that I/O-intensive programs should have priority over CPU-intensive ones, then the opposite policy could be instituted easily on some other computer system if the mechanism were properly separated and were policy independent.

Microkernel-based operating systems take the separation of mechanism and policy to one extreme, by implementing a basic set of primitive building blocks. These blocks are almost policy free, allowing more advanced mechanisms and policies to be added via user-created kernel modules, or via user programs themselves. At the other extreme is a system such as the Apple Macintosh operating system, in which both mechanism and policy are encoded in the system to enforce a global look and feel to the system. All applications have similar interfaces, because the interface itself is built into the kernel.

Policy decisions must be made for all resource-allocation and scheduling problems. Whenever the question is *how* rather than *what*, it is a mechanism that must be determined.

3.7.3 Implementation

Once an operating system is designed, it must be implemented. Traditionally, operating systems have been written in assembly language. Now, however, they are often written in higher-level languages such as C or C++.

The first system not written in assembly language was probably the Master Control Program (MCP) for Burroughs computers. MCP was written in a variant of ALGOL. MULTICS, developed at MIT, was written mainly in PL/1. The Primos operating system for Prime computers is written in a dialect of Fortran. The UNIX operating system, OS/2, and Windows NT are mainly written in C. Only some 900 lines of code of the original UNIX were in assembly language, most of which constituted the scheduler and device drivers.

The advantages of using a higher-level language, or at least a systems-implementation language, for implementing operating systems are the same as those accrued when the language is used for application programs: The code can be written faster, is more compact, and is easier to understand and debug. In addition, improvements in compiler technology will improve the generated code for the entire operating system by simple recompilation. Finally, an operating system is far easier to **port**—to move to some other hardware—if it is written in a high-level language. For example, MS-DOS was written in Intel 8088 assembly language. Consequently, it is available on only the Intel family of CPUs.

The UNIX operating system, on the other hand, which is written mostly in C, is available on a number of different CPUs, including Intel 80X86, Pentium, Motorola 680X0, Ultra SPARC, Compaq Alpha, and MIPS RX000.

Opponents of implementing an operating system in a higher-level language claim the major disadvantages are reduced speed and increased storage requirements. Although an expert assembly-language programmer can produce efficient small routines, for large programs a modern compiler can perform complex analysis and apply sophisticated optimizations that produce excellent code. Modern processors have deep pipelining and multiple functional units, which can handle complex dependencies that can overwhelm the limited ability of the human mind to keep track of details.

As is true in other systems, major performance improvements in operating systems are more likely to be the result of better data structures and algorithms than of excellent assembly-language code. In addition, although operating systems are large, only a small amount of the code is critical to high performance; the memory manager and the CPU scheduler are probably the most critical routines. After the system is written and is working correctly, bottleneck routines can be identified and replaced with assembly-language equivalents.

To identify bottlenecks, we must be able to monitor system performance. Code must be added to compute and display measures of system behavior. In a number of systems, the operating system does this task by producing trace listings of system behavior. All interesting events are logged with their time and important parameters and are written to a file. Later, an analysis program can process the log file to determine system performance and to identify bottlenecks and inefficiencies. These same traces can be run as input for a simulation of a suggested improved system. Traces also can help people to find errors in operating-system behavior.

An alternative is to compute and display the performance measures in real time. For example, a timer can trigger a routine to store the current instruction pointer value. The result is a statistical picture of the program locations most frequently used within the program. This approach may allow system operators to become familiar with system behavior and to modify system operation in real time.

3.8 ■ System Generation

We can design, code, and implement an operating system specifically for one machine at one site. More commonly, however, operating systems are designed to run on any of a class of machines at a variety of sites with a variety of peripheral configurations. The system must then be configured or generated for each specific computer site, a process sometimes known as **system generation (SYSGEN)**.

The operating system is normally distributed on disk or tape. To generate a system, we use a special program. The SYSGEN program reads from a given file, or asks the operator of the system for information concerning the specific configuration of the hardware system, or probes the hardware directly to determine what components are there. The following kinds of information must be determined.

- What CPU will be used? What options (extended instruction sets, floating-point arithmetic, and so on) are installed? For multiple-CPU systems, each CPU must be described.
- How much memory is available? Some systems will determine this value themselves by referencing memory location after memory location until an “illegal address” fault is generated. This procedure defines the final legal address and hence the amount of available memory.
- What devices are available? The system will need to know how to address each device (the device number), the device interrupt number, the device’s type and model, and any special device characteristics.
- What operating-system options are desired, or what parameter values are to be used? These options or values might include how many buffers of which sizes should be used, what type of CPU-scheduling algorithm is desired, what the maximum number of processes to be supported is, and so on.

Once this information is determined, it can be used in several ways. At one extreme, a system administrator can use it to modify a copy of the source code of the operating system. The operating system then is completely compiled. Data

declarations, initializations, and constants, along with conditional compilation, produce an output object version of the operating system that is tailored to the system described.

At a slightly less tailored level, the system description can cause the creation of tables and the selection of modules from a precompiled library. These modules are linked together to form the generated operating system. Selection allows the library to contain the device drivers for all supported I/O devices, but only those needed are linked into the operating system. Because the system is not recompiled, system generation is faster, but the resulting system may be overly general.

At the other extreme, a system that is completely table driven can be constructed. All the code is always part of the system, and selection occurs at execution time, rather than at compile or link time. System generation involves simply creating the appropriate tables to describe the system. Most modern operating systems are constructed in this manner. Solaris performs some system-configuration discovery at installation time, and some at boot time. A configuration file can be used by the systems administrator to fine-tune system variables, but hardware support is automatically configured by the kernel. Likewise, Windows 2000 requires no manual configuration on installation or at boot time. Once the basic questions about disk layout and network configuration are answered, the installation program automatically detects the system hardware and installs a properly generated operating system.

The major differences among these approaches are the size and generality of the generated system and the ease of modification as the hardware configuration changes. Consider the cost of modifying the system to support a newly acquired graphics terminal or another disk drive. Balanced against that cost, of course, is the frequency (or infrequency) of such changes.

After an operating system is generated, it must be made available for use by the hardware. But how does the hardware know where the kernel is, or how to load that kernel? The procedure of starting a computer by loading the kernel is known as **booting** the system. Most computer systems have a small piece of code, stored in ROM, known as the **bootstrap program** or **bootstrap loader**. This code is able to locate the kernel, load it into main memory, and start its execution. Some computer systems, such as PCs, use a two-step process in which a simple bootstrap loader fetches a more complex boot program from disk, which in turn loads the kernel. Booting a system is discussed in Section 14.3.2 and in Appendix A.

3.9 ■ Summary

Operating systems provide a number of services. At the lowest level, system calls allow a running program to make requests from the operating system directly. At a higher level, the command interpreter or shell provides a mech-

anism for a user to issue a request without writing a program. Commands may come from files during batch-mode execution, or directly from a keyboard when in an interactive or time-shared mode. System programs are provided to satisfy many common user requests.

The types of requests vary according to the level of the request. The system-call level must provide the basic functions, such as process control and file and device management. Higher-level requests, satisfied by the command interpreter or system programs, are translated into a sequence of system calls. System services can be classified into several categories: program control, status requests, and I/O requests. Program errors can be considered implicit requests for service.

Once the system services are defined, the structure of the operating system can be developed. Various tables are needed to record the information that defines the state of the computer system and the status of the system's jobs.

The design of a new operating system is a major task. The goals of the system must be well defined before the design begins. They form the foundation for choices among various algorithms and strategies that will be necessary.

Since an operating system is large, modularity is important. Designing a system as a sequence of layers or using a microkernel is considered a good technique. The virtual-machine concept takes the layered approach and treats both the kernel of the operating system and the hardware as though they were all hardware. Other operating systems may even be loaded on top of this virtual machine.

Any operating system that has implemented the JVM is able to run all Java programs, because the JVM abstracts the underlying system to the Java program, providing an architecture-neutral interface.

Throughout the entire operating-system design cycle, we must be careful to separate policy decisions from implementation details (or mechanisms). This separation allows maximum flexibility if policy decisions are to be changed later.

Operating systems are now almost always written in a systems-implementation language or in a higher-level language. This feature improves their implementation, maintenance, and portability. To create an operating system for a particular machine configuration, we must perform system generation.

■ Exercises

- 3.1 What are the five major activities of an operating system in regard to process management?
- 3.2 What are the three major activities of an operating system in regard to memory management?

- 3.3 What are the three major activities of an operating system in regard to secondary-storage management?
- 3.4 What are the five major activities of an operating system in regard to file management?
- 3.5 What is the purpose of the command interpreter? Why is it usually separate from the kernel?
- 3.6 List five services provided by an operating system. Explain how each provides convenience to the users. Explain in which cases it would be impossible for user-level programs to provide these services.
- 3.7 What is the purpose of system calls?
- 3.8 Using system calls, write a program in either C or C++ that reads data from one file and copies it to another file. Such a program was described in Section 3.3.
- 3.9 Why does Java provide the ability to call from a Java program native methods that are written in, say, C or C++? Provide an example where a native method is useful.
- 3.10 What is the purpose of system programs?
- 3.11 What is the main advantage of the layered approach to system design?
- 3.12 What is the main advantage of the microkernel approach to system design?
- 3.13 What is the main advantage for an operating-system designer of using a virtual-machine architecture? What is the main advantage for a user?
- 3.14 Why is a just-in-time compiler useful for executing Java programs?
- 3.15 Why is the separation of mechanism and policy a desirable principle?
- 3.16 The experimental Synthesis operating system has an assembler incorporated within the kernel. To optimize system-call performance, the kernel assembles routines within kernel space to minimize the path that the system call must take through the kernel. This approach is the antithesis of the layered approach, in which the path through the kernel is extended so that building the operating system is made easier. Discuss the pros and cons of the Synthesis approach to kernel design and to system-performance optimization.

Bibliographical Notes

Dijkstra [1968] advocated the layered approach to operating-system design. Brinch-Hansen [1970] was an early proponent of the construction of an operating system as a kernel (or nucleus) on which can be built more complete systems. Information of the QNX microkernel can be found at <http://www.qnx.com>.

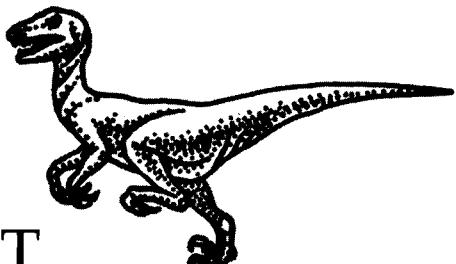
The first operating system to provide a virtual machine was the CP/67 on an IBM 360/67. The commercially available IBM VM/370 operating system was derived from CP/67. General discussions concerning virtual machines have been presented by Hendricks and Hartmann [1979], by MacKinnon [1979], and by Schultz [1988]. A virtual machine allowing Windows applications to run on Linux is shown at <http://www.vmware.com>. Cheung and Loong [1995] explored issues of operating-systems structuring from microkernel to extensible systems. Back et al. [2000] discusses the design of Java operating systems.

MS-DOS, Version 3.1, is described in Microsoft [1986]. Windows NT is described by Solomon [1998]. The Apple Macintosh operating system is described in Apple [1987]. Berkeley UNIX (BSD) is described in McKusick et al. [1996]. The standard AT&T UNIX system V is described in Earhart [1986]. A good description of OS/2 is given by Iacobucci [1988]. Mach is introduced by Accetta et al. [1986], and AIX is presented in Loucks and Sauer [1987]. The experimental Synthesis operating system is discussed by Massalin and Pu [1989]. Linux kernel details are presented by Bar [2000].

The specification for the Java language and the Java virtual machine is presented by Gosling et al. [1996] and by Lindholm and Yellin [1998], respectively. The internal workings of the Java virtual machine are described by Venners [1998] and by Meyer and Downing [1997]. Jones and Lin [1996] provide a thorough coverage of garbage-collection algorithms. More information on Java is available on the Web at <http://www.javasoft.com>.

Part Two

PROCESS MANAGEMENT



A *process* can be thought of as a program in execution. A process will need certain resources—such as CPU time, memory, files, and I/O devices—to accomplish its task. These resources are allocated to the process either when it is created or while it is executing.

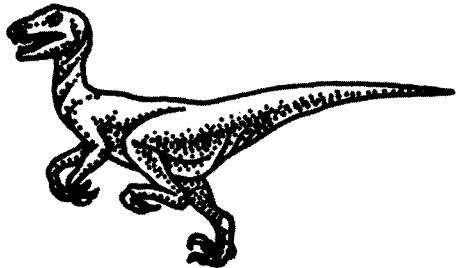
A process is the unit of work in most systems. Such a system consists of a collection of processes: Operating-system processes execute system code, and user processes execute user code. All these processes may execute concurrently.

Although traditionally a process contained only a single *thread* of control as it ran, most modern operating systems now support processes that have multiple threads.

The operating system is responsible for the following activities in connection with process and thread management: the creation and deletion of both user and system processes; the scheduling of processes; and the provision of mechanisms for synchronization, communication, and deadlock handling for processes.

Chapter 4

PROCESSES



Early computer systems allowed only one program to be executed at a time. This program had complete control of the system, and had access to all the system's resources. Current-day computer systems allow multiple programs to be loaded into memory and to be executed concurrently. This evolution required firmer control and more compartmentalization of the various programs. These needs resulted in the notion of a **process**, which is a program in execution. A process is the unit of work in a modern time-sharing system.

The more complex the operating system, the more it is expected to do on behalf of its users. Although its main concern is the execution of user programs, it also needs to take care of various system tasks that are better left outside the kernel itself. A system therefore consists of a collection of processes: Operating-system processes executing system code, and user processes executing user code. All these processes can potentially execute concurrently, with the CPU (or CPUs) multiplexed among them. By switching the CPU between processes, the operating system can make the computer more productive.

4.1 ■ Process Concept

One impediment to our discussion of operating systems is the question of what to call all the CPU activities. A batch system executes *jobs*, whereas a time-shared system has *user programs*, or *tasks*. Even on a single-user system, such as Microsoft Windows and Macintosh OS, a user may be able to run several programs at one time: a word processor, web browser, and e-mail package.

Even if the user can execute only one program at a time, the operating system may need to support its own internal programmed activities, such as memory management. In many respects, all these activities are similar, so we call all of them *processes*.

The terms *job* and *process* are used almost interchangeably in this text. Although we personally prefer the term *process*, much of operating-system theory and terminology was developed during a time when the major activity of operating systems was *job processing*. It would be misleading to avoid the use of commonly accepted terms that include the word *job* (such as *job scheduling*) simply because *process* has superseded *job*.

4.1.1 The Process

Informally, a process is a program in execution. A process is more than the program code, which is sometimes known as the **text section**. It also includes the current activity, as represented by the value of the **program counter** and the contents of the processor's registers. In addition, a process generally includes the process **stack**, which contains temporary data (such as method parameters, return addresses, and local variables), and a **data section**, which contains global variables.

We emphasize that a program by itself is not a process; a program is a *passive* entity, such as the contents of a file stored on disk, whereas a process is an *active* entity, with a program counter specifying the next instruction to execute and a set of associated resources.

Although two processes may be associated with the same program, they are nevertheless considered two separate execution sequences. For instance, several users may be running different copies of the mail program, or the same user may invoke many copies of the editor program. Each of these is a separate process, and, although the text sections are equivalent, the data sections vary. It is also common to have a process that spawns many processes as it runs. We discuss such matters in Section 4.4.

4.1.2 Process State

As a process executes, it changes state. The **state** of a process is defined in part by the current activity of that process. Each process may be in one of the following states:

- **New:** The process is being created.
- **Running:** Instructions are being executed.
- **Waiting:** The process is waiting for some event to occur (such as an I/O completion or reception of a signal).
- **Ready:** The process is waiting to be assigned to a processor.

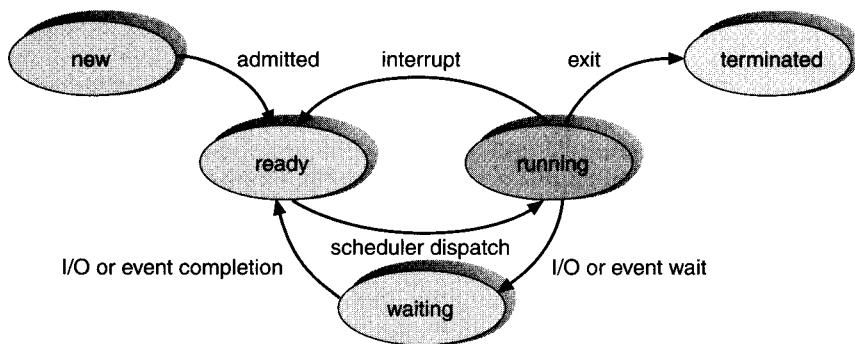


Figure 4.1 Diagram of process state.

- **Terminated:** The process has finished execution.

These state names are arbitrary, and they vary across operating systems. The states that they represent are found on all systems, however. Certain operating systems more finely delineate process states. Only one process can be *running* on any processor at any instant, although many processes may be *ready* and *waiting*. The state diagram corresponding to these states is presented in Figure 4.1.

4.1.3 Process Control Block

Each process is represented in the operating system by a **process control block** (PCB)—also called a task control block. A PCB is shown in Figure 4.2. It contains many pieces of information associated with a specific process, including these:

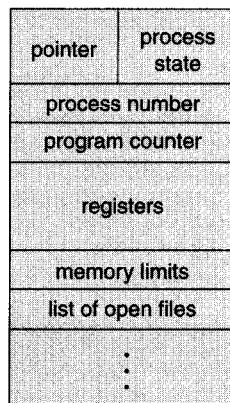


Figure 4.2 Process control block (PCB).

- **Process state:** The state may be new, ready, running, waiting, halted, and so on.
- **Program counter:** The counter indicates the address of the next instruction to be executed for this process.
- **CPU registers:** The registers vary in number and type, depending on the computer architecture. They include accumulators, index registers, stack pointers, and general-purpose registers, plus any condition-code information. Along with the program counter, this state information must be saved when an interrupt occurs, to allow the process to be continued correctly afterward (Figure 4.3).
- **CPU-scheduling information:** This information includes a process priority, pointers to scheduling queues, and any other scheduling parameters. (Chapter 6 describes process scheduling.)
- **Memory-management information:** This information may include such information as the value of the base and limit registers, the page tables, or the segment tables, depending on the memory system used by the operating system (Chapter 9).

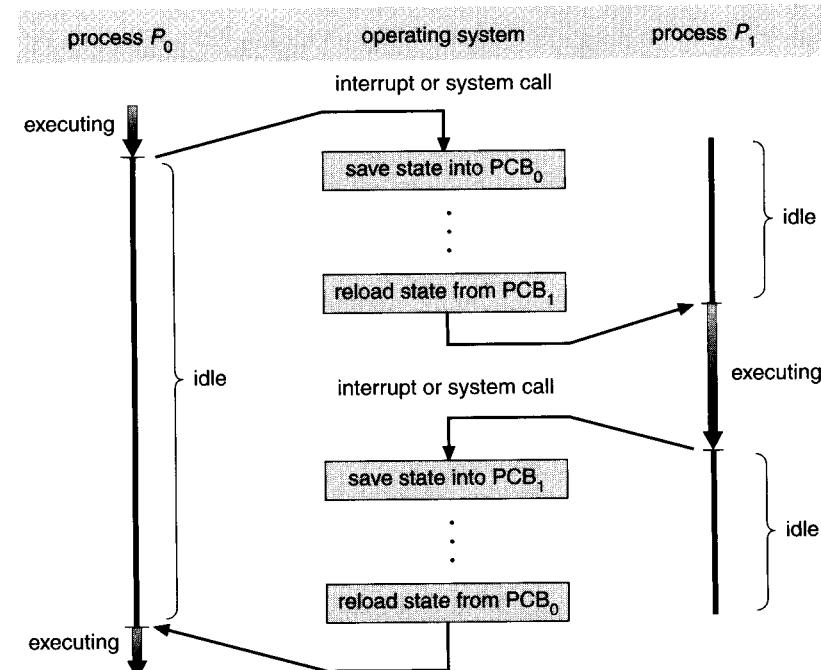


Figure 4.3 Diagram showing CPU switch from process to process.

- **Accounting information:** This information includes the amount of CPU and real time used, time limits, account numbers, job or process numbers, and so on.
- **I/O status information:** The information includes the list of I/O devices allocated to this process, a list of open files, and so on.

The PCB simply serves as the repository for any information that may vary from process to process.

4.1.4 Threads

The process model discussed so far has implied that a process is a program that performs a single thread of execution. For example, if a process is running a word-processor program, a single thread of instructions is being executed. This single thread of control allows the process to perform only one task at one time. For example, the user could not simultaneously type in characters and run the spell checker within the same process. Many modern operating systems have extended the process concept to allow a process to have multiple threads of execution. They thus allow the process to perform more than one task at a time. Chapter 5 explores multithreaded processes.

4.2 ■ Process Scheduling

The objective of multiprogramming is to have some process running at all times, so as to maximize CPU utilization. The objective of time-sharing is to switch the CPU among processes so frequently that users can interact with each program while it is running. A uniprocessor system can have only one running process. If more processes exist, the rest must wait until the CPU is free and can be rescheduled.

4.2.1 Scheduling Queues

As processes enter the system, they are put into a **job queue**. This queue consists of all processes in the system. The processes that are residing in main memory and are ready and waiting to execute are kept on a list called the **ready queue**. This queue is generally stored as a linked list. A ready-queue header contains pointers to the first and final PCBs in the list. We extend each PCB to include a pointer field that points to the next PCB in the ready queue.

The operating system also has other queues. When a process is allocated the CPU, it executes for a while and eventually quits, is interrupted, or waits for the occurrence of a particular event, such as the completion of an I/O request. In the case of an I/O request, such a request may be to a dedicated tape drive, or to a shared device, such as a disk. Since the system has many processes, the disk

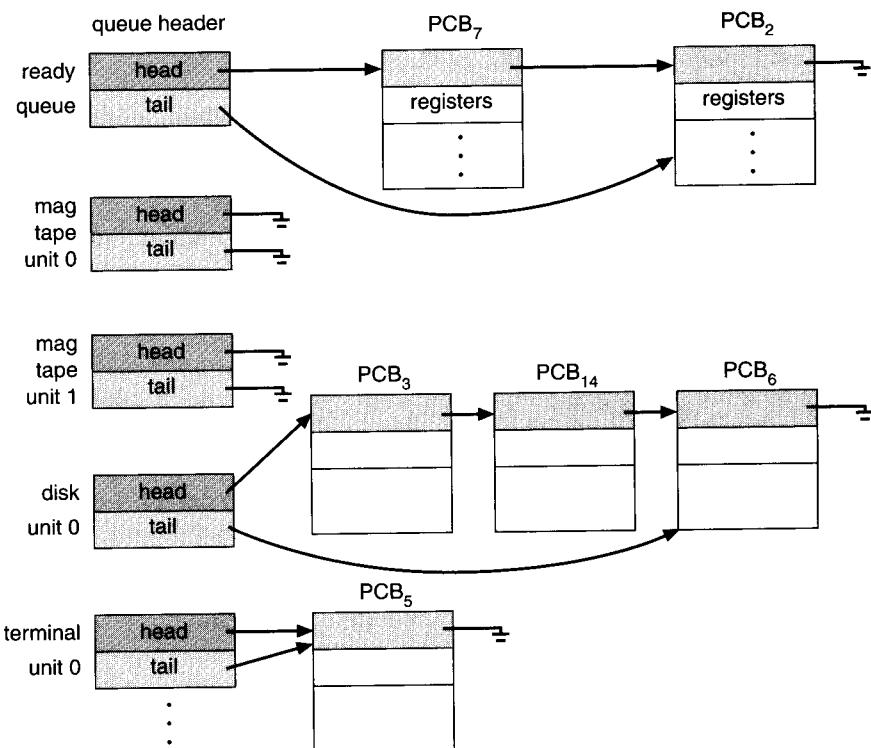


Figure 4.4 The ready queue and various I/O device queues.

may be busy with the I/O request of some other process. The process therefore may have to wait for the disk. The list of processes waiting for a particular I/O device is called a **device queue**. Each device has its own device queue (Figure 4.4).

A common representation of process scheduling is a **queueing diagram**, such as that in Figure 4.5. Each rectangular box represents a queue. Two types of queues are present: the ready queue and a set of device queues. The circles represent the resources that serve the queues, and the arrows indicate the flow of processes in the system.

A new process is initially put in the ready queue. It waits in the ready queue until it is selected for execution (or **dispatched**). Once the process is assigned to the CPU and is executing, one of several events could occur:

- The process could issue an I/O request, and then be placed in an I/O queue.
- The process could create a new subprocess and wait for its termination.
- The process could be removed forcibly from the CPU, as a result of an interrupt, and be put back in the ready queue.

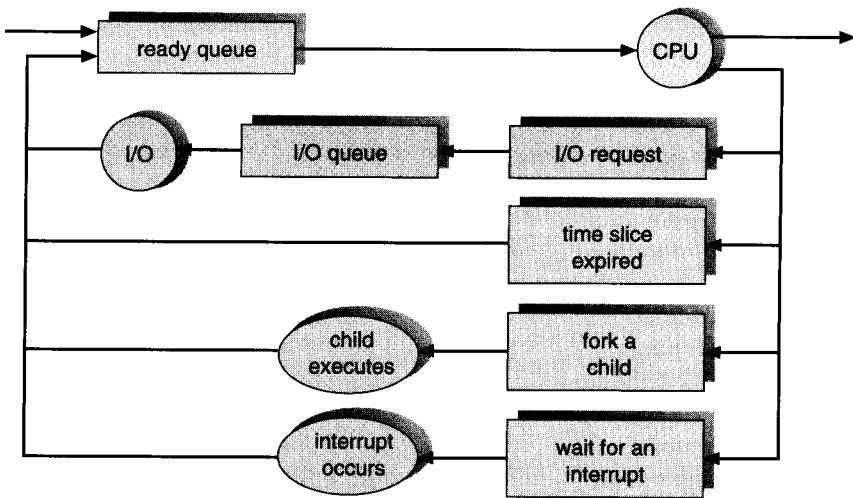


Figure 4.5 Queueing-diagram representation of process scheduling.

In the first two cases, the process eventually switches from the waiting state to the ready state, and is then put back in the ready queue. A process continues this cycle until it terminates, at which time it is removed from all queues and has its PCB and resources deallocated.

4.2.2 Schedulers

A process migrates between the various scheduling queues throughout its lifetime. The operating system must select, for scheduling purposes, processes from these queues in some fashion. The selection process is carried out by the appropriate **scheduler**.

In a batch system, often more processes are submitted than can be executed immediately. These processes are spooled to a mass-storage device (typically a disk), where they are kept for later execution. The **long-term scheduler**, or **job scheduler**, selects processes from this pool and loads them into memory for execution. The **short-term scheduler**, or **CPU scheduler**, selects from among the processes that are ready to execute, and allocates the CPU to one of them.

The primary distinction between these two schedulers is the frequency of their execution. The short-term scheduler must select a new process for the CPU frequently. A process may execute for only a few milliseconds before waiting for an I/O request. Often, the short-term scheduler executes at least once every 100 milliseconds. Because of the brief time between executions, the short-term scheduler must be fast. If it takes 10 milliseconds to decide to execute a process for 100 milliseconds, then $10/(100 + 10) = 9$ percent of the CPU is being used (or wasted) simply for scheduling the work.

The long-term scheduler, on the other hand, executes much less frequently. There may be minutes between the creation of new processes in the system. The long-term scheduler controls the **degree of multiprogramming**—the number of processes in memory. If the degree of multiprogramming is stable, then the average rate of process creation must be equal to the average departure rate of processes leaving the system. Thus, the long-term scheduler may need to be invoked only when a process leaves the system. Because of the longer interval between executions, the long-term scheduler can afford to take more time to select a process for execution.

The long-term scheduler must make a careful selection. In general, most processes can be described as either I/O bound or CPU bound. An **I/O-bound process** spends more of its time doing I/O than it spends doing computations. A **CPU-bound process**, on the other hand, generates I/O requests infrequently, using more of its time doing computation than an I/O-bound process uses. The long-term scheduler should select a good **process mix** of I/O-bound and CPU-bound processes. If all processes are I/O bound, the ready queue will almost always be empty, and the short-term scheduler will have little to do. If all processes are CPU bound, the I/O waiting queue will almost always be empty, devices will go unused, and again the system will be unbalanced. The system with the best performance will have a combination of CPU-bound and I/O-bound processes.

On some systems, the long-term scheduler may be absent or minimal. For example, time-sharing systems such as UNIX often have no long-term scheduler, but simply put every new process in memory for the short-term scheduler. The stability of these systems depends either on a physical limitation (such as the number of available terminals) or on the self-adjusting nature of human users. If the performance declines to unacceptable levels, some users will simply quit.

Some operating systems, such as time-sharing systems, may introduce an additional, intermediate level of scheduling. This **medium-term scheduler**,

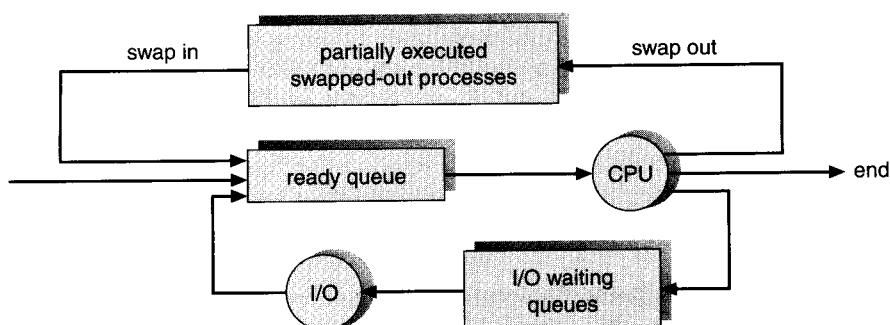


Figure 4.6 Addition of medium-term scheduling to the queueing diagram.

diagrammed in Figure 4.6, removes processes from memory (and from active contention for the CPU), and thus reduces the degree of multiprogramming. At some later time, the process can be reintroduced into memory and its execution can be continued where it left off. This scheme is called **swapping**. The process is swapped out, and is later swapped in, by the medium-term scheduler. Swapping may be necessary to improve the process mix, or because a change in memory requirements has overcommitted available memory, requiring memory to be freed up. Swapping is discussed in Chapter 9.

4.2.3 Context Switch

Switching the CPU to another process requires saving the state of the old process and loading the saved state for the new process. This task is known as a **context switch**. The **context** of a process is represented in the PCB of a process; it includes the value of the CPU registers, the process state (Figure 4.1), and memory-management information. When a context switch occurs, the kernel saves the context of the old process in its PCB and loads the saved context of the new process scheduled to run. Context-switch time is pure overhead, because the system does no useful work while switching. Its speed varies from machine to machine, depending on the memory speed, the number of registers that must be copied, and the existence of special instructions (such as a single instruction to load or store all registers). Typical speeds range from 1 to 1000 microseconds.

Context-switch times are highly dependent on hardware support. For instance, some processors (such as the Sun UltraSPARC) provide multiple sets of registers. A context switch simply includes changing the pointer to the current register set. Of course, if active processes exceed register sets, the system resorts to copying register data to and from memory, as before. Also, the more complex the operating system, the more work must be done during a context switch. As we will see in Chapter 9, advanced memory-management techniques may require extra data to be switched with each context. For instance, the address space of the current process must be preserved as the space of the next task is prepared for use. How the address space is preserved, and what amount of work is needed to preserve it, depend on the memory-management method of the operating system. As we will see in Chapter 5, context switching has become such a performance bottleneck that programmers are using new structures (threads) to avoid it whenever possible.

4.3 ■ Operations on Processes

The processes in the system can execute concurrently, and they must be created and deleted dynamically. Thus, the operating system must provide a mechanism (or facility) for process creation and termination.

4.3.1 Process Creation

A process may create several new processes, via a `create-process` system call, during the course of execution. The creating process is called a **parent** process, whereas the new processes are called the **children** of that process. Each of these new processes may in turn create other processes, forming a *tree* of processes (Figure 4.7).

In general, a process will need certain resources (such as CPU time, memory, files, I/O devices) to accomplish its task. When a process creates a subprocess, that subprocess may be able to obtain its resources directly from the operating system, or it may be constrained to a subset of the resources of the parent process. The parent may have to partition its resources among its children, or it may be able to share some resources (such as memory or files) among several of its children. Restricting a child process to a subset of the parent's resources prevents any process from overloading the system by creating too many subprocesses.

When a process is created it obtains, in addition to the various physical and logical resources, initialization data (or input) that may be passed along from the parent process to the child process. For example, consider a process whose function is to display the status of a file, say F_1 , on the screen of a terminal. When it is created, it will get, as an input from its parent process, the name of the file F_1 , and it will execute using that datum to obtain the desired information.

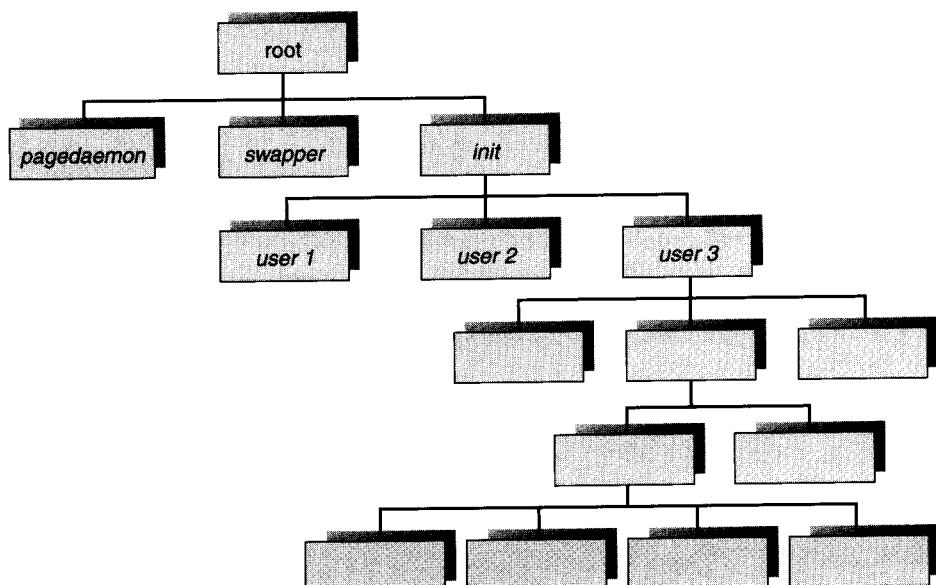


Figure 4.7 A tree of processes on a typical UNIX system.

It may also get the name of the output device. Some operating systems pass resources to child processes. On such a system, the new process may get two open files, F_1 and the terminal device, and may just need to transfer the datum between the two.

When a process creates a new process, two possibilities exist in terms of execution:

1. The parent continues to execute concurrently with its children.
2. The parent waits until some or all of its children have terminated.

There are also two possibilities in terms of the address space of the new process:

1. The child process is a duplicate of the parent process.
2. The child process has a program loaded into it.

To illustrate these different implementations, let us consider the UNIX operating system. In UNIX, each process is identified by its **process identifier**, which is a unique integer. A new process is created by the `fork` system call. The new process consists of a copy of the address space of the original process. This mechanism allows the parent process to communicate easily with its child process. Both processes (the parent and the child) continue execution at the instruction after the `fork` system call, with one difference: The return code for the `fork` system call is zero for the new (child) process, whereas the (nonzero) process identifier of the child is returned to the parent.

Typically, the `execvp` system call is used after a `fork` system call by one of the two processes to replace the process' memory space with a new program. The `execvp` system call loads a binary file into memory—destroying the memory image of the program containing the `execvp` system call—and starts its execution. In this manner, the two processes are able to communicate, and then to go their separate ways. The parent can then create more children, or, if it has nothing else to do while the child runs, it can issue a `wait` system call to move itself off the ready queue until the termination of the child. The C program shown in Figure 4.8 illustrates the UNIX system calls previously described. The parent creates a child process using the `fork` system call. We now have two different processes running a copy of the same program. The value of `pid` for the child process is zero; that for the parent is an integer value greater than zero. The child process overlays its address space with the UNIX command `/bin/ls` (used to get a directory listing) using the `execvp` system call. The parent waits for the child process to complete with the `wait` system call. When the child process completes, the parent process resumes from the call to `wait` where it completes using the `exit` system call.

The DEC VMS operating system, in contrast, creates a new process, loads a specified program into that process, and starts it running. The Microsoft

```

#include <stdio.h>

void main(int argc, char *argv[])
{
    int pid;

    /* fork another process */
    pid = fork();

    if (pid < 0) { /* error occurred */
        fprintf(stderr, "Fork Failed");
        exit(-1);
    }
    else if (pid == 0) { /* child process */
        execlp("/bin/ls", "ls", NULL);
    }
    else { /* parent process */
        /* parent will wait for the child to complete */
        wait(NULL);
        printf("Child Complete");
        exit(0);
    }
}

```

Figure 4.8 C program forking a separate process.

Windows NT operating system supports both models: The parent's address space may be duplicated, or the parent may specify the name of a program for the operating system to load into the address space of the new process.

4.3.2 Process Termination

A process terminates when it finishes executing its final statement and asks the operating system to delete it by using the `exit` system call. At that point, the process may return data (output) to its parent process (via the `wait` system call). All the resources of the process—including physical and virtual memory, open files, and I/O buffers—are deallocated by the operating system.

Termination occurs under additional circumstances. A process can cause the termination of another process via an appropriate system call (for example, `abort`). Usually, only the parent of the process that is to be terminated can invoke such a system call. Otherwise, users could arbitrarily kill each other's jobs. A parent therefore needs to know the identities of its children. Thus, when one process creates a new process, the identity of the newly created process is passed to the parent.

A parent may terminate the execution of one of its children for a variety of reasons, such as these:

- The child has exceeded its usage of some of the resources that it has been allocated. This requires the parent to have a mechanism to inspect the state of its children.
- The task assigned to the child is no longer required.
- The parent is exiting, and the operating system does not allow a child to continue if its parent terminates. On such systems, if a process terminates (either normally or abnormally), then all its children must also be terminated. This phenomenon, referred to as **cascading termination**, is normally initiated by the operating system.

To illustrate process execution and termination, consider that in UNIX we can terminate a process by using the `exit` system call; its parent process may wait for the termination of a child process by using the `wait` system call. The `wait` system call returns the process identifier of a terminated child, so that the parent can tell which of its possibly many children has terminated. If the parent terminates, however, all its children have assigned as their new parent the `init` process. Thus, the children still have a parent to collect their status and execution statistics.

4.4 ■ Cooperating Processes

The concurrent processes executing in the operating system may be either independent processes or cooperating processes. A process is **independent** if it cannot affect or be affected by the other processes executing in the system. Clearly, any process that does not share any data (temporary or persistent) with any other process is independent. On the other hand, a process is **cooperating** if it can affect or be affected by the other processes executing in the system. Clearly, any process that shares data with other processes is a cooperating process.

We may want to provide an environment that allows process cooperation for several reasons:

- **Information sharing:** Since several users may be interested in the same piece of information (for instance, a shared file), we must provide an environment to allow concurrent access to these types of resources.
- **Computation speedup:** If we want a particular task to run faster, we must break it into subtasks, each of which will be executing in parallel with the others. Such a speedup can be achieved only if the computer has multiple processing elements (such as CPUs or I/O channels).

- **Modularity:** We may want to construct the system in a modular fashion, dividing the system functions into separate processes or threads, as we discussed in Chapter 3.
- **Convenience:** Even an individual user may have many tasks on which to work at one time. For instance, a user may be editing, printing, and compiling in parallel.

Concurrent execution of cooperating processes requires mechanisms that allow processes to communicate with one another (Section 4.5) and to synchronize their actions (Chapter 7).

To illustrate the concept of cooperating processes, let us consider the producer–consumer problem, which is a common paradigm for cooperating processes. A **producer** process produces information that is consumed by a **consumer** process. For example, a print program produces characters that are consumed by the printer driver. A compiler may produce assembly code, which is consumed by an assembler. The assembler, in turn, may produce object modules, which are consumed by the loader.

To allow producer and consumer processes to run concurrently, we must have available a buffer of items that can be filled by the producer and emptied by the consumer. A producer can produce one item while the consumer is consuming another item. The producer and consumer must be synchronized, so that the consumer does not try to consume an item that has not yet been produced. In this situation, the consumer must wait until an item is produced.

The **unbounded-buffer** producer–consumer problem places no practical limit on the size of the buffer. The consumer may have to wait for new items, but the producer can always produce new items. The **bounded-buffer** producer–consumer problem assumes a fixed buffer size. In this case, the consumer must wait if the buffer is empty, and the producer must wait if the buffer is full.

The buffer may either be provided by the operating system through the use of an **interprocess-communication (IPC)** facility (Section 4.5), or by explicitly coded by the application programmer with the use of shared memory. Let us illustrate a shared-memory solution to the bounded-buffer problem. The producer and consumer processes share the following variables:

```
#define BUFFER_SIZE 10

typedef struct {
    .
    .
} item;

item buffer[BUFFER_SIZE];
int in = 0;
int out = 0;
```

The shared buffer is implemented as a circular array with two logical pointers: `in` and `out`. The variable `in` points to the next free position in the buffer; `out` points to the first full position in the buffer. The buffer is empty when `in == out`; the buffer is full when $((in + 1) \% \text{BUFFER_SIZE}) == \text{out}$.

The code for the producer and consumer processes follows. The producer process has a local variable `nextProduced` in which the new item to be produced is stored:

```
while (1) {
    /* produce an item in nextProduced */
    while (((in + 1) \% BUFFER_SIZE) == out)
        ; /* do nothing */
    buffer[in] = nextProduced;
    in = (in + 1) \% BUFFER_SIZE;
}
```

The consumer process has a local variable `nextConsumed` in which the item to be consumed is stored:

```
while (1) {
    while (in == out)
        ; // do nothing

    nextConsumed = buffer[out];
    out = (out + 1) \% BUFFER_SIZE;
    /* consume the item in nextConsumed */
}
```

This scheme allows at most `BUFFER_SIZE - 1` items in the buffer at the same time. We leave it as an exercise for you to provide a solution where `BUFFER_SIZE` items can be in the buffer at the same time.

In Chapter 7, we discuss how synchronization among cooperating processes can be implemented effectively in a shared-memory environment.

4.5 ■ Interprocess Communication

In Section 4.4, we showed how cooperating processes can communicate in a shared-memory environment. The scheme requires that these processes share a common buffer pool, and that the code for implementing the buffer be written explicitly by the application programmer. Another way to achieve the same effect is for the operating system to provide the means for cooperating processes to communicate with each other via an interprocess communication (IPC) facility.

IPC provides a mechanism to allow processes to communicate and to synchronize their actions without sharing the same address space. IPC is particularly useful in a distributed environment where the communicating processes may reside on different computers connected with a network. An example is a **chat** program used on the World Wide Web.

IPC is best provided by a message-passing system, and message systems can be defined in many ways. In this section, we look at different issues when designing message-passing systems.

4.5.1 Message-Passing System

The function of a message system is to allow processes to communicate with one another without the need to resort to shared data. We have already seen message passing used as a method of communication in microkernels (Section 3.5.3). In this scheme, services are provided as ordinary user processes. That is, the services operate outside of the kernel. Communication among the user processes is accomplished through the passing of messages. An IPC facility provides at least the two operations: `send(message)` and `receive(message)`.

Messages sent by a process can be of either fixed or variable size. If only fixed-sized messages can be sent, the system-level implementation is straightforward. This restriction, however, makes the task of programming more difficult. On the other hand, variable-sized messages require a more complex system-level implementation, but the programming task becomes simpler.

If processes *P* and *Q* want to communicate, they must send messages to and receive messages from each other; a **communication link** must exist between them. This link can be implemented in a variety of ways. We are concerned here not with the link's physical implementation (such as shared memory, hardware bus, or network, which are covered in Chapter 15), but rather with its logical implementation. Here are several methods for logically implementing a link and the `send/receive` operations:

- Direct or indirect communication
- Symmetric or asymmetric communication
- Automatic or explicit buffering
- Send by copy or send by reference
- Fixed-sized or variable-sized messages

We look at each of these types of message systems next.

4.5.2 Naming

Processes that want to communicate must have a way to refer to each other. They can use either direct or indirect communication.

4.5.2.1 Direct Communication

With **direct communication**, each process that wants to communicate must explicitly name the recipient or sender of the communication. In this scheme, the `send` and `receive` primitives are defined as:

- `send(P, message)` –Send a message to process P.
- `receive(Q, message)` –Receive a message from process Q.

A communication link in this scheme has the following properties:

- A link is established automatically between every pair of processes that want to communicate. The processes need to know only each other's identity to communicate.
- A link is associated with exactly two processes.
- Exactly one link exists between each pair of processes.

This scheme exhibits symmetry in addressing; that is, both the sender and the receiver processes must name the other to communicate. A variant of this scheme employs asymmetry in addressing. Only the sender names the recipient; the recipient is not required to name the sender. In this scheme, the `send` and `receive` primitives are defined as follows:

- `send(P, message)` —Send a message to process P.
- `receive(id, message)` —Receive a message from any process; the variable `id` is set to the name of the process with which communication has taken place.

The disadvantage in both symmetric and asymmetric schemes is the limited modularity of the resulting process definitions. Changing the name of a process may necessitate examining all other process definitions. All references to the old name must be found, so that they can be modified to the new name. This situation is not desirable from the viewpoint of separate compilation.

4.5.2.2 Indirect Communication

With **indirect communication**, the messages are sent to and received from **mailboxes**, or **ports**. A mailbox can be viewed abstractly as an object into

which messages can be placed by processes and from which messages can be removed. Each mailbox has a unique identification. In this scheme, a process can communicate with some other process via a number of different mailboxes. Two processes can communicate only if they share a mailbox. The `send` and `receive` primitives are defined as follows:

- `send(A, message)` – Send a message to mailbox A.
- `receive(A, message)` – Receive a message from mailbox A.

In this scheme, a communication link has the following properties:

- A link is established between a pair of processes only if both members of the pair have a shared mailbox.
- A link may be associated with more than two processes.
- A number of different links may exist between each pair of communicating processes, with each link corresponding to one mailbox.

Now suppose that processes P_1 , P_2 , and P_3 all share mailbox A. Process P_1 sends a message to A, while P_2 and P_3 each execute a `receive` from A. Which process will receive the message sent by P_1 ? The answer depends on the scheme that we choose:

- Allow a link to be associated with at most two processes.
- Allow at most one process at a time to execute a `receive` operation.
- Allow the system to select arbitrarily which process will receive the message (that is, either P_2 or P_3 , but not both, will receive the message). The system may identify the receiver to the sender.

A mailbox may be owned either by a process or by the operating system. If the mailbox is owned by a process (that is, the mailbox is part of the address space of the process), then we distinguish between the owner (who can only receive messages through this mailbox) and the user (who can only send messages to the mailbox). Since each mailbox has a unique owner, there can be no confusion about who should receive a message sent to this mailbox. When a process that owns a mailbox terminates, the mailbox disappears. Any process that subsequently sends a message to this mailbox must be notified that the mailbox no longer exists.

On the other hand, a mailbox owned by the operating system is independent and is not attached to any particular process. The operating system then must provide a mechanism that allows a process to do the following:

- Create a new mailbox.
- Send and receive messages through the mailbox.
- Delete a mailbox.

The process that creates a new mailbox is that mailbox's owner by default. Initially, the owner is the only process that can receive messages through this mailbox. However, the ownership and receive privilege may be passed to other processes through appropriate system calls. Of course, this provision could result in multiple receivers for each mailbox.

4.5.3 Synchronization

Communication between processes takes place by calls to `send` and `receive` primitives. There are different design options for implementing each primitive. Message passing may be either **blocking** or **nonblocking**—also known as **synchronous** and **asynchronous**.

- **Blocking send:** The sending process is blocked until the message is received by the receiving process or by the mailbox.
- **Nonblocking send:** The sending process sends the message and resumes operation.
- **Blocking receive:** The receiver blocks until a message is available.
- **Nonblocking receive:** The receiver retrieves either a valid message or a null.

Different combinations of `send` and `receive` are possible. When both the `send` and `receive` are blocking, we have a **rendezvous** between the sender and the receiver.

4.5.4 Buffering

Whether the communication is direct or indirect, messages exchanged by communicating processes reside in a temporary queue. Basically, such a queue can be implemented in three ways:

- **Zero capacity:** The queue has maximum length 0; thus, the link cannot have any messages waiting in it. In this case, the sender must block until the recipient receives the message.

- **Bounded capacity:** The queue has finite length n ; thus, at most n messages can reside in it. If the queue is not full when a new message is sent, the latter is placed in the queue (either the message is copied or a pointer to the message is kept), and the sender can continue execution without waiting. The link has a finite capacity, however. If the link is full, the sender must block until space is available in the queue.
- **Unbounded capacity:** The queue has potentially infinite length; thus, any number of messages can wait in it. The sender never blocks.

The zero-capacity case is sometimes referred to as a message system with no buffering; the other cases are referred to as automatic buffering.

4.5.5 An Example: Mach

As an example of a message-based operating system, consider the Mach operating system, developed at Carnegie Mellon University. The Mach kernel supports the creation and destruction of multiple tasks, which are similar to processes but have multiple threads of control. Most communication in Mach—including most of the system calls and all intertask information—is carried out by *messages*. Messages are sent to and received from mailboxes, called *ports* in Mach.

Even system calls are made by messages. When each task is created, two special mailboxes—the **Kernel** mailbox and the **Notify** mailbox—are also created. The kernel uses the Kernel mailbox to communicate with the task. The kernel sends notification of event occurrences to the Notify port. Only three system calls are needed for message transfer. The `msg_send` call sends a message to a mailbox. A message is received via `msg_receive`. **Remote procedure calls (RPCs)** are executed via `msg_rpc`, which sends a message and waits for exactly one return message from the sender. In this way, RPC model a typical subroutine procedure call, but can work between systems.

The `port_allocate` system call creates a new mailbox and allocates space for its queue of messages. The maximum size of the message queue defaults to eight messages. The task that creates the mailbox is that mailbox's owner. The owner also is given receive access to the mailbox. Only one task at a time can either own or receive from a mailbox, but these rights can be sent to other tasks if desired.

The mailbox has an initially empty queue of messages. As messages are sent to the mailbox, the messages are copied into the mailbox. All messages have the same priority. Mach guarantees that multiple messages from the same sender are queued in **first-in, first-out (FIFO)** order, but does not guarantee an absolute ordering. For instance, messages sent from each of two senders may be queued in any order.

The messages themselves consist of a fixed-length header, followed by a variable-length data portion. The header includes the length of the message

and two mailbox names. When a message is sent, one mailbox name is the mailbox to which the message is being sent. Commonly, the sending thread expects a reply; the mailbox name of the sender is passed on to the receiving task, which may use it as a “return address” to send messages back.

The variable part of a message is a list of typed data items. Each entry in the list has a type, size, and value. The type of the objects specified in the message is important, since operating-system-defined objects—such as the ownership or receive access rights, task states, and memory segments—may be sent in messages.

The `send` and `receive` operations themselves are flexible. For instance, when a message is sent to a mailbox, the mailbox may be full. If the mailbox is not full, the message is copied to the mailbox and the sending thread continues. If the mailbox is full, the sending thread has four options:

1. Wait indefinitely until there is room in the mailbox.
2. Wait at most n milliseconds.
3. Do not wait at all, but rather return immediately.
4. Temporarily cache a message. One message can be given to the operating system to keep, even though the mailbox to which it is being sent is full. When the message can be put in the mailbox, a notification message is sent back to the sender; only one such message to a full mailbox can be pending at any time for a given sending thread.

The final option is meant for server tasks, such as a line-printer driver. After finishing a request, these tasks may need to send a one-time reply to the task that had requested service, but must also continue with other service requests, even if the reply mailbox for a client is full.

The `receive` operation must specify from which mailbox or mailbox set to receive a message. A **mailbox set** is a collection of mailboxes, as declared by the task, which can be grouped together and treated as one mailbox for the purposes of the task. Threads in a task can receive from only a mailbox or mailbox set for which that task has receive access. A `port_status` system call returns the number of messages in a given mailbox. The `receive` operation attempts to receive from either of the following:

1. any mailbox in a mailbox set
2. a specific (named) mailbox

If no message is waiting to be received, the receiving thread may wait, wait at most n milliseconds, or not wait.

The Mach system was especially designed for distributed systems, which we discuss in Chapters 15 through 17, but Mach is also suitable for single-processor systems. The major problem with message systems has generally

been poor performance caused by copying the message first from the sender to the mailbox, and then from the mailbox to the receiver. The Mach message system attempts to avoid double-copy operations by using virtual-memory —management techniques (Chapter 10). Essentially, Mach maps the address space containing the sender’s message into the receiver’s address space. The message itself is never actually copied. This message-management technique provides a large performance boost, but works only for intrasystem messages. The Mach operating system is discussed in an extra chapter that is posted on our web site (<http://www.bell-labs.com/topic/books/os-book>).

4.5.6 An Example: Windows 2000

The Windows 2000 operating system is an example of modern design that employs modularity to increase functionality and decrease the time needed to implement new features. Windows 2000 provides support for multiple operating environments or *subsystems*, with which application programs communicate via a message-passing mechanism. The application programs can be considered to be clients of the Windows 2000 subsystem server.

The message-passing facility in Windows 2000 is called the **local procedure-call (LPC)** facility. The LPC in Windows 2000 communicates between two processes that are on the same machine. It is similar to the standard RPC mechanism that is widely used, but it is optimized for and specific to Windows 2000. Like Mach, Windows 2000 uses a port object to establish and maintain a connection between two processes. Every client that calls a subsystem needs a communication channel, which is provided by a port object and is never inherited. Windows 2000 uses two types of ports: connection ports and communication ports. They are really the same but are given different names according to how they are used. Connection ports are named *objects*, which are visible to all processes; they give applications a way to set up a communication channel (Chapter 21). This communication works as follows:

- The client opens a handle to the subsystem’s connection port object.
- The client sends a connection request.
- The server creates two private communication ports, and returns the handle to one of them to the client.
- The client and server use the corresponding port handle to send messages or callbacks and to listen for replies.

Windows 2000 uses three types of message-passing techniques over a port that the client specifies when it establishes the channel. The simplest, which is used for small messages, uses the port’s message queue as intermediate storage

and copies the message from one process to the other. Under this method, messages of up to 256 bytes can be sent.

If a client needs to send a larger message, it passes the message through a section object (or shared memory). The client has to decide, when it sets up the channel, whether or not it will need to send a large message. If the client determines that it does want to send large messages, it asks for a section object to be created. Likewise, if the server decides that replies will be large, it creates a section object. So that the section object can be used, a small message is sent that contains a pointer and size information about that section object. This method is more complicated than the first method, but it avoids the data copying. In both cases, a callback mechanism can be used when either the client or the server cannot respond immediately to a request. The callback mechanism allows them to perform asynchronous message handling.

4.6 ■ Communication in Client - Server Systems

Consider a user who needs access to data located at some server. For example, a user may wish to find out the total number of lines, words, and characters in a file located at server *A*. This request is handled by a remote server *A*, which accesses the file, computes the desired result, and eventually transfers the actual data back to the user.

4.6.1 Sockets

A **socket** is defined as an endpoint for communication. A pair of processes communicating over a network employ a pair of sockets—one for each process. A socket is made up of an IP address concatenated with a port number. In general, sockets use a client–server architecture. The server waits for incoming client requests by listening to a specified port. Once a request is received, the server accepts a connection from the client socket to complete the connection.

Servers implementing specific services (such as telnet, ftp, and http) listen to well-known ports (a telnet server listens to port 23, an ftp server listens to port 21, and a web (or http) server listens to port 80). All ports below 1024 are considered well known; we can use them to implement standard services.

When a client process initiates a request for a connection, it is assigned a port by the host computer. This port is some arbitrary number greater than 1024. For example, if a client on host *X* with IP address 146.86.5.20 wishes to establish a connection with a web server (which is listening on port 80) at address 161.25.19.8, host *X* may be assigned port 1625. The connection will consist of a pair of sockets: (146.86.5.20:1625) on host *X*, and (161.25.19.8:80) on the web server. This situation is illustrated in Figure 4.9. The packets traveling between the hosts are delivered to the appropriate process, based on the destination port number.

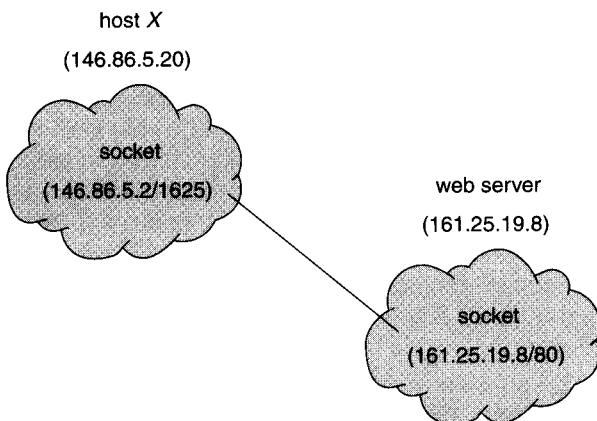


Figure 4.9 Communication using sockets.

All connections must be unique. Therefore, if another process also on host X wished to establish another connection with the same web server, it would be assigned a port number greater than 1024 and not equal to 1625. This ensures that all connections consist of a unique pair of sockets.

Although most program examples in this text use C, we will illustrate sockets using Java as it provides a much easier interface to sockets and has a rich library for networking utilities. Those interested in socket programming in C or C++ should consult the Bibliographical Notes.

Java provides three different types of sockets. **Connection-oriented (TCP) sockets** are implemented with the `Socket` class. **Connectionless (UDP) sockets** use the `DatagramSocket` class. A third type is the `MulticastSocket` class, which is a subclass of the `DatagramSocket` class. A multicast socket allows data to be sent to multiple recipients.

As an example of Java-based sockets, we now present a Java class that implements a time-of-day server. The operation allows clients to request the time of day from the server. The server listens to port 5155, although the port could be any arbitrary number greater than 1024. When a connection is received, the server returns the time of day to the client.

The time-of-day server is shown in Figure 4.10. The server creates a `ServerSocket` that specifies it will listen to port 5155. It then begins listening to the port with the `accept` method. The server blocks on the `accept` method waiting for a client to request a connection. When a connection request is received, `accept` returns a socket that the server can use to communicate with the client.

The details illustrating how the server communicates with the socket are as follows. The server first establishes a `PrintWriter` object that it will use to communicate with the client. A `PrintWriter` object allows the server to write to the socket using the normal `print` and `println` methods for output. The

```

import java.net.*;
import java.io.*;

public class Server
{
    public static void main(String[] args) throws IOException {
        Socket client = null;
        PrintWriter pout = null;
        ServerSocket sock = null;

        try {
            sock = new ServerSocket(5155);
            // now listen for connections

            while (true) {
                client = sock.accept();

                // we have a connection
                pout = new PrintWriter(client.getOutputStream(), true);

                // write the Date to the socket
                pout.println(new java.util.Date().toString());

                pout.close();
                client.close();
            }
        }
        catch (IOException ioe) {
            System.err.println(ioe);
        }
        finally {
            if (client != null)
                client.close();
            if (sock != null)
                sock.close();
        }
    }
}

```

Figure 4.10 Time-of-day server.

server process sends the time of day to the client calling the method `println`. Once it has written the time of day to the socket, the server closes the socket to the client and resumes listening for more requests.

A client communicates with the server by creating a socket and connecting to the port the server is listening on. We implement such a client in the Java program shown in Figure 4.11. The client creates a `Socket` and requests a connection with the server at IP address 127.0.0.1 on port 5155. Once the connection is made, the client can read from the socket using normal stream I/O statements. After it has received the time of day from the server, the client closes the socket and exits. The IP address 127.0.0.1 is a special IP address known as the **local host**. When a computer refers to IP address 127.0.0.1, it is referring to itself. This mechanism allows the client and server on the same host to communicate using the TCP/IP protocol. The IP address 127.0.0.1 could be replaced with the IP address of another host running the time-of-day server.

```
import java.net.*;
import java.io.*;

public class Client
{
    public static void main(String[] args) throws IOException {
        InputStream in = null;
        BufferedReader bin = null;
        Socket sock = null;

        try {
            //make connection to socket
            sock = new Socket("127.0.0.1",5155);

            in = sock.getInputStream();
            bin = new BufferedReader(new InputStreamReader(in));

            String line;
            while ( (line = bin.readLine()) != null)
                System.out.println(line);
        }
        catch (IOException ioe) {
            System.err.println(ioe);
        }
        finally {
            if (sock != null)
                sock.close();
        }
    }
}
```

Figure 4.11 The client.

Communication using sockets—although common and efficient—is considered a low-level form of communication between distributed processes. One reason is that sockets allow only an unstructured stream of bytes to be exchanged between the communicating threads. It is the responsibility of the client or server application to impose a structure on the data. In the next two subsections, we look at two alternative, higher-level methods of communication: remote procedure calls (RPCs) and remote method invocation (RMI).

4.6.2 Remote Procedure Calls

One of the most common forms of remote service is the RPC paradigm, which we discussed briefly in Section 4.5.4. The RPC was designed as a way to abstract the procedure-call mechanism for use between systems with network connections. It is similar in many respects to the IPC mechanism described in Section 4.5, and it is usually built on top of such a system. Because we are dealing with an environment in which the processes are executing on separate systems, we must use a message-based communication scheme to provide remote service. In contrast to the IPC facility, the messages exchanged for RPC communication are well structured and are thus no longer just packets of data. They are addressed to an RPC daemon listening to a port on the remote system, and contain an identifier of the function to execute and the parameters to pass to that function. The function is then executed as requested, and any output is sent back to the requester in a separate message.

A *port* is simply a number included at the start of a message packet. Whereas a system normally has one network address, it can have many ports within that address to differentiate the many network services it supports. If a remote process needs a service, it addresses its messages to the proper port. For instance, if a system wished to allow other systems to be able to list the current users on it, it would have a daemon supporting such an RPC attached to a port —say, port 3027. Any remote system could obtain the needed information (that is, the list of current users) by sending an RPC message to port 3027 on the server; the data would be received in a reply message.

The semantics of RPCs allow a client to invoke a procedure on a remote host as it would invoke a procedure locally. The RPC system hides the necessary details allowing the communication to take place. The RPC system does this by providing a **stub** on the client side. Typically, a separate stub exists for each separate remote procedure. When the client invokes a remote procedure, the RPC system calls the appropriate stub, passing it the parameters provided to the remote procedure. This stub locates the port on the server and *marshalls* the parameters. Parameter marshalling involves packaging the parameters into a form which may be transmitted over a network. The stub then transmits a message to the server using message passing. A similar stub on the server side receives this message and invokes the procedure on the server. If necessary, return values are passed back to the client using the same technique.

One issue that must be dealt with concerns differences in data representation on the client and server machines. Consider the representation of 32-bit integers. Some systems use the high memory address to store the most significant byte (known as *big-endian*), while other systems store the least significant byte at the high memory address (known as *little-endian*). To resolve this, many RPC systems define a machine-independent representation of data. One such representation is known as **external data representation (XDR)**. On the client side, parameter marshalling involves converting the machine-dependent data into XDR before being sent to the server. On the server side, the XDR data is unmarshalled and converted into the machine-dependent representation for the server.

The RPC mechanism is common on networked systems, so we should discuss several other issues in regard to its operation. One important issue is the semantics of a call. Whereas local procedure calls fail only under extreme circumstances, RPCs can fail, or be duplicated and executed more than once, due to common network errors. Because we are dealing with message transfer over unreliable communication links, it is much easier for an operating system to ensure that a message was acted on at most once, than it is to ensure that the message was acted on exactly once. Because local procedure calls have the latter meaning, most systems attempt to duplicate that functionality. They do so by attaching to each message a timestamp. The server must keep a history of all the timestamps of messages it has already processed, or a history large enough to ensure that repeated messages are detected. Incoming messages that have a timestamp already in the history are ignored. Generation of these timestamps is discussed in Section 17.1.

Another important issue concerns the communication between a server and a client. With standard procedure calls, some form of binding takes place during link, load, or execution time (Chapter 9), such that a procedure call's name is replaced by the memory address of the procedure call. The RPC scheme requires a similar binding of the client and the server port, but how does a client know the port numbers on the server? Neither system has full information about the other because they do not share memory. Two approaches are common. First, the binding information may be predetermined, in the form of fixed port addresses. At compile time, an RPC call has a fixed port number associated with it. Once a program is compiled, the server cannot change the port number of the requested service. Second, binding can be done dynamically by a rendezvous mechanism. Typically, an operating system provides a rendezvous (also called a **matchmaker**) daemon on a fixed RPC port. A client then sends a message, containing the name of the RPC, to the rendezvous daemon requesting the port address of the RPC it needs to execute. The port number is returned, and the RPC calls may be sent to that port until the process terminates (or the server crashes). This method requires the extra overhead of the initial request, but is more flexible than the first approach. Figure 4.12 shows a sample interaction.

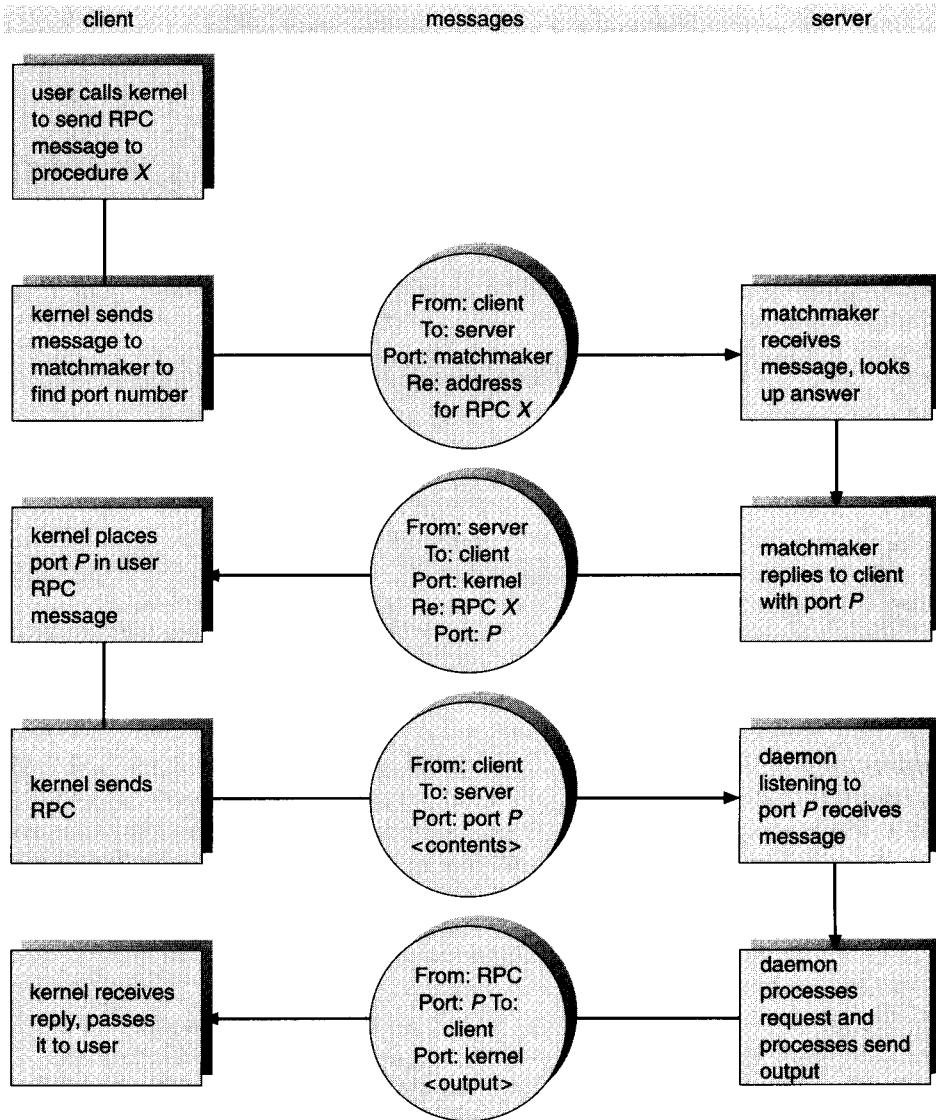


Figure 4.12 Execution of a remote procedure call (RPC).

The RPC scheme is useful in implementing a distributed file system (Chapter 16). Such a system can be implemented as a set of RPC daemons and clients. The messages are addressed to the DFS port on a server on which a file operation is to take place. The message contains the disk operation to be performed. Disk operations might be `read`, `write`, `rename`, `delete`, or `status`, corresponding to the usual file-related system calls. The return message contains any data

resulting from that call, which is executed by the DFS daemon on behalf of the client. For instance, a message might contain a request to transfer a whole file to a client, or be limited to simple block requests. In the latter case, several such requests might be needed if a whole file is to be transferred.

4.6.3 Remote Method Invocation

The **remote method invocation (RMI)** is a Java feature similar to RPCs. RMI allows a thread to invoke a method on a remote object. Objects are considered remote if they reside in a different Java virtual machine (JVM). Therefore, the remote object may be in a different JVM on the same computer or on a remote host connected by a network. This situation is illustrated in Figure 4.13. RMI and RPCs differ in two fundamental ways. First, RPCs support procedural programming whereby only remote procedures or functions may be called. RMI is object-based: It supports invocation of methods on remote objects. Second, the parameters to remote procedures are ordinary data structures in RPC; with RMI it is possible to pass objects as parameters to remote methods. By allowing a Java program to invoke methods on remote objects, RMI makes it possible for users to develop Java applications that are distributed across a network.

To make remote methods transparent to both the client and the server, RMI implements the remote object using stubs and skeletons. A **stub** is a proxy for the remote object; it resides with the client. When a client invokes a remote method, this stub for the remote object is called. This client-side stub is responsible for creating a **parcel** consisting of the name of the method to be invoked on the server and the marshalled parameters for the method. The stub then sends this parcel to the server, where the skeleton for the remote object receives it. The **skeleton** is responsible for unmarshalling the parameters and invoking the desired method on the server. The skeleton then marshalls the return value (or exception, if any) into a parcel and returns this parcel to the client. The stub unmarshalls the return value and passes it to the client.

Let us demonstrate how this process works. Assume that a client wishes to invoke a method on a remote object **Server** with the signature

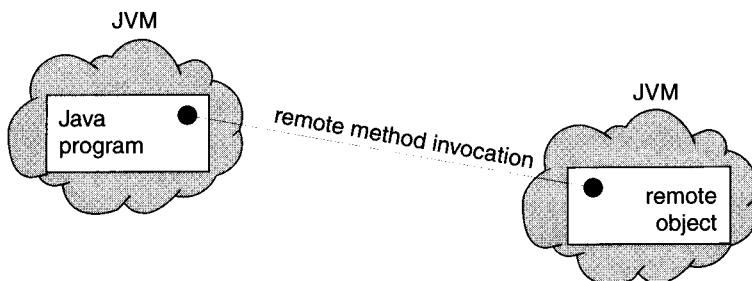


Figure 4.13 Remote method invocation.

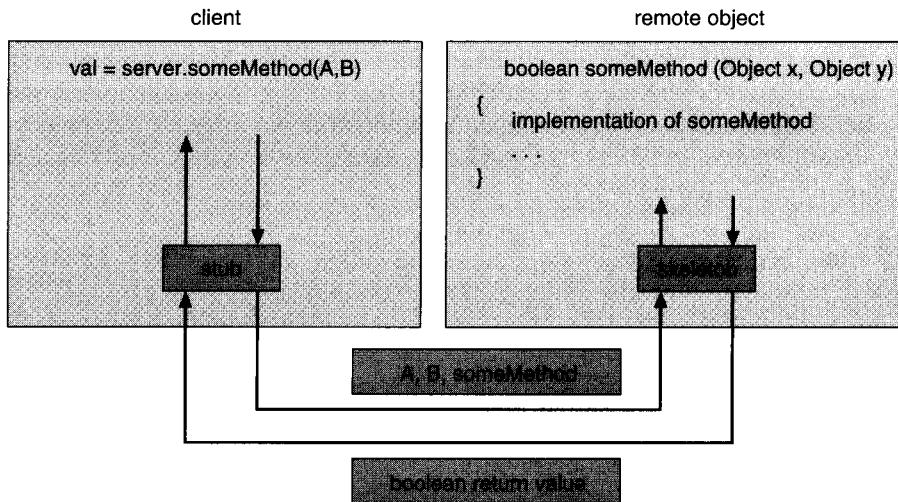


Figure 4.14 Marshalling parameters.

`someMethod(Object, Object)` that returns a `boolean` value. The client executes the statement

```
boolean val = Server.someMethod(A, B);
```

The call to `someMethod` with the parameters `A` and `B` invokes the stub for the remote object. The stub marshalls into a parcel the parameters `A` and `B` and the name of the method that is to be invoked on the server, then sends this parcel to the server. The skeleton on the server unmarshalls the parameters and invokes the method `someMethod`. The actual implementation of `someMethod` resides on the server. Once the method is completed, the skeleton marshalls the `Boolean` value returned from `someMethod` and sends this value back to the client. The stub unmarshalls this return value and passes it to the client. The process is shown in Figure 4.14.

Fortunately, the level of abstraction that RMI provides makes the stubs and skeletons transparent, allowing Java developers to write programs that invoke distributed methods just as they would invoke local methods. It is crucial, however, that you understand a few rules about the behavior of parameter passing.

- If the marshalled parameters are **local** (or **nonremote**) objects, they are passed by copy using a technique known as **object serialization**. However, if the parameters are also remote objects, they are passed by reference. In our example, if `A` is a local object and `B` a remote object, `A` is serialized and passed by copy, and `B` is passed by reference. This would in turn allow the server to invoke methods on `B` remotely.

- If local objects are to be passed as parameters to remote objects, they must implement the interface `java.io.Serializable`. Many objects in the core Java API implement `Serializable`, allowing them to be used with RMI. Object serialization allows the state of an object to be written to a byte stream.

4.7 ■ Summary

A process is a program in execution. As a process executes, it changes state. The state of a process is defined by that process' current activity. Each process may be in one of the following states: new, ready, running, waiting, or terminated. Each process is represented in the operating system by its own process-control block (PCB).

A process, when it is not executing, is placed in some waiting queue. The two major classes of queues in an operating system are I/O request queues and the ready queue. The ready queue contains all the processes that are ready to execute and are waiting for the CPU. Each process is represented by a PCB, and the PCBs can be linked together to form a ready queue. Long-term (or job) scheduling is the selection of processes to be allowed to contend for the CPU. Normally, long-term scheduling is heavily influenced by resource-allocation considerations, especially memory management. Short-term (or CPU) scheduling is the selection of one process from the ready queue.

The processes in the system can execute concurrently. There are several reasons for allowing concurrent execution: information sharing, computation speedup, modularity, and convenience. Concurrent execution requires mechanisms for process creation and deletion.

The processes executing in the operating system may be either independent processes or cooperating processes. Cooperating processes must have the means to communicate with each other. Principally, two complementary communication schemes exist: shared memory and message systems. The shared-memory method requires communicating processes to share some variables. The processes are expected to exchange information through the use of these shared variables. In a shared-memory system, the responsibility for providing communication rests with the application programmers; the operating system needs to provide only the shared memory. The message-system method allows the processes to exchange messages. The responsibility for providing communication may rest with the operating system itself. These two schemes are not mutually exclusive, and can be used simultaneously within a single operating system.

A socket is defined as an endpoint for communication. A connection between a pair of applications consists of a pair of sockets, one at each end of the communication channel. RPCs are another form of distributed communication. An RPC occurs when a process (or thread) calls a procedure on a remote

application. RMI is the Java version of an RPC. RMI allows a thread to invoke a method on a remote object just as it would invoke a method on a local object. The primary distinction between RPCs and RMIs is that the data being passed to a remote procedure are in the form of an ordinary data structure. RMI allows objects to be passed in remote method calls.

■ Exercises

- 4.1 MS-DOS provided no means of concurrent processing. Discuss three major complications that concurrent processing adds to an operating system.
- 4.2 Describe the differences among short-term, medium-term, and long-term scheduling.
- 4.3 A DECSYSTEM-20 computer has multiple register sets. Describe the actions of a context switch if the new context is already loaded into one of the register sets. What else must happen if the new context is in memory rather than in a register set, and all the register sets are in use?
- 4.4 Describe the actions taken by a kernel to switch context between processes.
- 4.5 What are the benefits and the detriments of each of the following? Consider both the systems and the programmers' levels.
 - a. Direct and indirect communication
 - b. Symmetric and asymmetric communication
 - c. Automatic and explicit buffering
 - d. Send by copy and send by reference
 - e. Fixed-sized and variable-sized messages
- 4.6 The correct producer–consumer algorithm in Section 4.4 allows only $n - 1$ buffers to be full at any one time. Modify the algorithm to allow all buffers to be utilized fully.
- 4.7 Consider the interprocess-communication scheme where mailboxes are used.
 - a. Suppose a process P wants to wait for two messages, one from mailbox A and one from mailbox B . What sequence of `send` and `receive` should it execute?
 - b. What sequence of `send` and `receive` should P execute if P wants to wait for one message from mailbox A or from mailbox B (or from both)?

- c. A *receive* operation makes a process wait until the mailbox is nonempty. Devise a scheme that allows a process to wait until a mailbox is empty, or explain why such a scheme cannot exist.
- 4.8** Write a socket-based *Fortune Teller* server. Your program should create a server that listens to a specified port. When a client receives a connection, the server should respond with a random fortune chosen from its database of fortunes.

Bibliographical Notes

The subject of interprocess communication was discussed by Brinch-Hansen [1970] with respect to the RC 4000 system. Schlichting and Schneider [1982] discussed asynchronous message-passing primitives. The IPC facility implemented at the user level was described by Bershad et al. [1990].

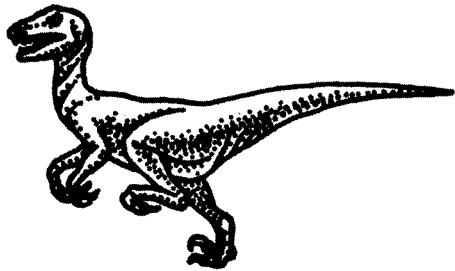
Details of interprocess communication in UNIX systems were presented by Gray [1997]. Barrera [1991] and Vahalia [1996] presented interprocess communication in the Mach system. Solomon and Russinovich [2000] outlined interprocess communication in Windows 2000. A thorough discussion of socket programming can be found in Stevens [1997].

Discussions concerning the implementation of RPCs were presented by Birrell and Nelson [1984]. A design of a reliable RPC mechanism was presented by Shrivastava and Panzieri [1982]. A survey of RPCs was presented by Tay and Ananda [1990]. Stankovic [1982] and Staunstrup [1982] discussed procedure calls versus message-passing communication.

Tanenbaum [1996] describes sockets and RPCs. Waldo [1988] and Farley [1998] discusses RPCs and RMI. Niemeyer and Peck [1997] and Horstmann and Cornell [1998] give good introductions to using RMI. The RMI homepage at [<http://www.javasoft.com/products/jdk/rmi>] lists up-to-date resources.

Chapter 5

THREADS



The process model introduced in Chapter 4 assumed that a process was an executing program with a single thread of control. Many modern operating systems now provide features for a process to contain multiple threads of control. This chapter introduces many concepts associated with multithreaded computer systems, including a discussion of the **Pthread** API and Java threads. We will look at many issues related to multithreaded programming and how it affects the design of operating systems. Finally, we will explore how several modern operating systems support threads at the kernel level.

5.1 ■ Overview

A thread, sometimes called a **lightweight process (LWP)**, is a basic unit of CPU utilization; it comprises a thread ID, a program counter, a register set, and a stack. It shares with other threads belonging to the same process its code section, data section, and other operating-system resources, such as open files and signals. A traditional (or **heavyweight**) process has a single thread of control. If the process has multiple threads of control, it can do more than one task at a time. Figure 5.1 illustrates the difference between a traditional single-threaded process and a multithreaded process.

5.1.1 Motivation

Many software packages that run on modern desktop PCs are **multithreaded**. An application typically is implemented as a separate process with several

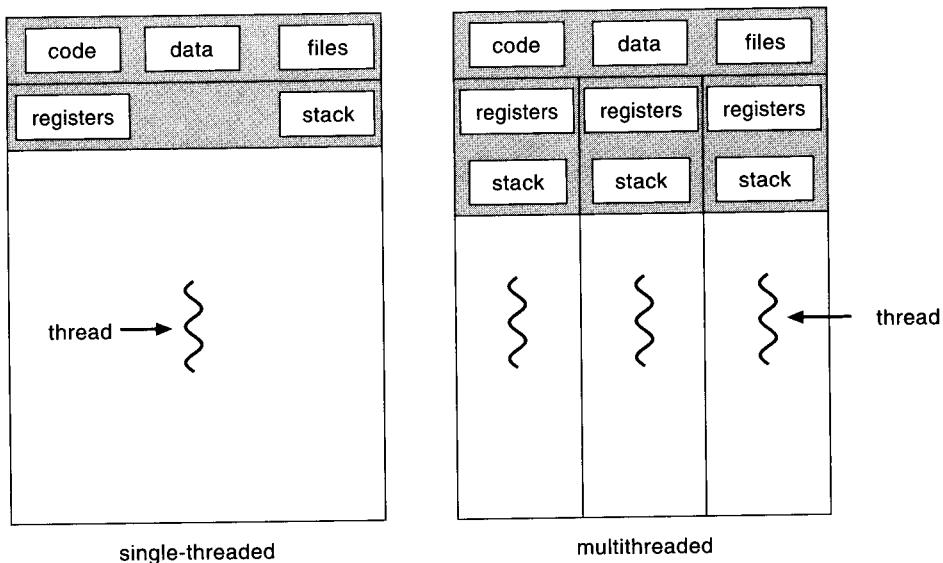


Figure 5.1 Single- and multithreaded processes.

threads of control. A web browser might have one thread display images or text while another thread retrieves data from the network. A word processor may have a thread for displaying graphics, another thread for reading keystrokes from the user, and a third thread for performing spelling and grammar checking in the background.

In certain situations a single application may be required to perform several similar tasks. For example, a web server accepts client requests for web pages, images, sound, and so forth. A busy web server may have several (perhaps hundreds) of clients concurrently accessing it. If the web server ran as a traditional **single-threaded** process, it would be able to service only one client at a time. The amount of time that a client might have to wait for its request to be serviced could be enormous.

One solution is to have the server run as a single process that accepts requests. When the server receives a request, it creates a separate process to service that request. In fact, this process-creation method was in common use before threads became popular. Process creation is very heavyweight, as was shown in the previous chapter. If the new process will perform the same tasks as the existing process, why incur all that overhead? It is generally more efficient for one process that contains multiple threads to serve the same purpose. This approach would multithread the web-server process. The server would create a separate thread that would listen for client requests; when a request was made, rather than creating another process, it would create another thread to service the request.

Threads also play a vital role in remote procedure call (RPC) systems. Recall from Chapter 4 that RPCs allow interprocess communication by providing a communication mechanism similar to ordinary function or procedure calls. Typically, RPC servers are multithreaded. When a server receives a message, it services the message using a separate thread. This allows the server to service several concurrent requests.

5.1.2 Benefits

The benefits of multithreaded programming can be broken down into four major categories:

1. **Responsiveness:** Multithreading an interactive application may allow a program to continue running even if part of it is blocked or is performing a lengthy operation, thereby increasing responsiveness to the user. For instance, a multithreaded web browser could still allow user interaction in one thread while an image is being loaded in another thread.
2. **Resource sharing:** By default, threads share the memory and the resources of the process to which they belong. The benefit of code sharing is that it allows an application to have several different threads of activity all within the same address space.
3. **Economy:** Allocating memory and resources for process creation is costly. Alternatively, because threads share resources of the process to which they belong, it is more economical to create and context switch threads. It can be difficult to gauge empirically the difference in overhead for creating and maintaining a process rather than a thread, but in general it is much more time consuming to create and manage processes than threads. In Solaris 2, creating a process is about 30 times slower than is creating a thread, and context switching is about five times slower.
4. **Utilization of multiprocessor architectures:** The benefits of multithreading can be greatly increased in a multiprocessor architecture, where each thread may be running in parallel on a different processor. A single-threaded process can only run on one CPU, no matter how many are available. Multithreading on a multi-CPU machine increases concurrency. In a single-processor architecture, the CPU generally moves between each thread so quickly as to create an illusion of parallelism, but in reality only one thread is running at a time.

5.1.3 User and Kernel Threads

Our discussion so far has treated threads in a generic sense. However, support for threads may be provided at either the user level, for *user threads*, or by the kernel, for *kernel threads*.

- **User threads** are supported above the kernel and are implemented by a thread library at the user level. The library provides support for thread creation, scheduling, and management with no support from the kernel. Because the kernel is unaware of user-level threads, all thread creation and scheduling are done in user space without the need for kernel intervention. Therefore, user-level threads are generally fast to create and manage; they have drawbacks, however. For instance, if the kernel is single-threaded, then any user-level thread performing a blocking system call will cause the entire process to block, even if other threads are available to run within the application. User-thread libraries include **POSIX Pthreads**, **Mach C-threads**, and **Solaris 2 UI-threads**.
- **Kernel threads** are supported directly by the operating system: The kernel performs thread creation, scheduling, and management in kernel space. Because thread management is done by the operating system, kernel threads are generally slower to create and manage than are user threads. However, since the kernel is managing the threads, if a thread performs a blocking system call, the kernel can schedule another thread in the application for execution. Also, in a multiprocessor environment, the kernel can schedule threads on different processors. Most contemporary operating systems—including Windows NT, Windows 2000, Solaris 2, BeOS, and Tru64 UNIX (formerly Digital UNIX)—support kernel threads.

We will cover Pthreads in Section 5.4 as an example of a user-level thread library. We will also cover Windows 2000 (Section 5.6) and Solaris 2 (Section 5.5) as examples of operating systems with kernel-thread support. We will discuss how Linux provides support for threads in Section 5.7 (although Linux does not quite refer to them as *threads*).

Java provides support for threads as well. However, as Java threads are created and managed by the Java virtual machine (JVM), they do not easily fall under the realm of either user or kernel threads. We will cover Java threads in Section 5.8.

5.2 ■ Multithreading Models

Many systems provide support for both user and kernel threads, resulting in different multithreading models. We look at three common types of threading implementation.

5.2.1 Many-to-One Model

The many-to-one model (Figure 5.2) maps many user-level threads to one kernel thread. Thread management is done in user space, so it is efficient, but

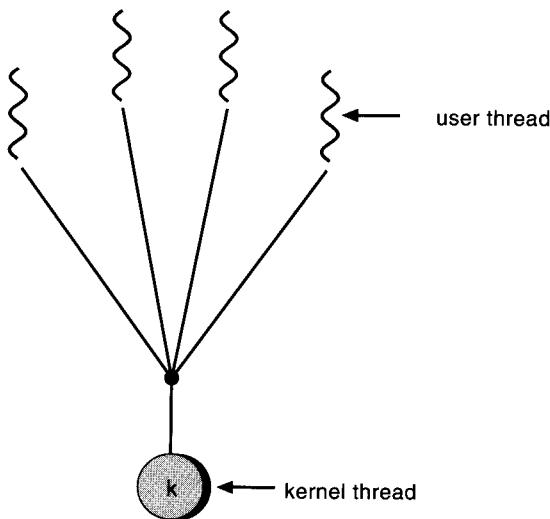


Figure 5.2 Many-to-one model.

the entire process will block if a thread makes a blocking system call. Also, because only one thread can access the kernel at a time, multiple threads are unable to run in parallel on multiprocessors. **Green threads**—a thread library available for Solaris 2—uses this model. In addition, user-level thread libraries implemented on operating systems that do not support kernel threads use the many-to-one model.

5.2.2 One-to-One Model

The one-to-one model (Figure 5.3) maps each user thread to a kernel thread. It provides more concurrency than the many-to-one model by allowing another thread to run when a thread makes a blocking system call; it also allows multiple threads to run in parallel on multiprocessors. The only drawback to this model is that creating a user thread requires creating the corresponding kernel thread. Because the overhead of creating kernel threads can burden the performance of an application, most implementations of this model restrict the number of threads supported by the system. Windows NT, Windows 2000, and OS/2 implement the one-to-one model.

5.2.3 Many-to-Many Model

The many-to-many model (Figure 5.4) multiplexes many user-level threads to a smaller or equal number of kernel threads. The number of kernel threads may be specific to either a particular application or a particular machine (an application may be allocated more kernel threads on a multiprocessor than on

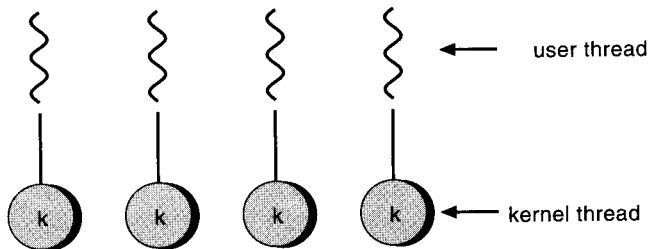


Figure 5.3 One-to-one model.

a uniprocessor). Whereas the many-to-one model allows the developer to create as many user threads as she wishes, true concurrency is not gained because the kernel can schedule only one thread at a time. The one-to-one model allows for greater concurrency, but the developer has to be careful not to create too many threads within an application (and in some instances may be limited in the number of threads she can create). The many-to-many model suffers from neither of these shortcomings: Developers can create as many user threads as necessary, and the corresponding kernel threads can run in parallel on a multiprocessor. Also, when a thread performs a blocking system call, the kernel can schedule another thread for execution. Solaris 2, IRIX, HP-UX, and Tru64 UNIX support this model.

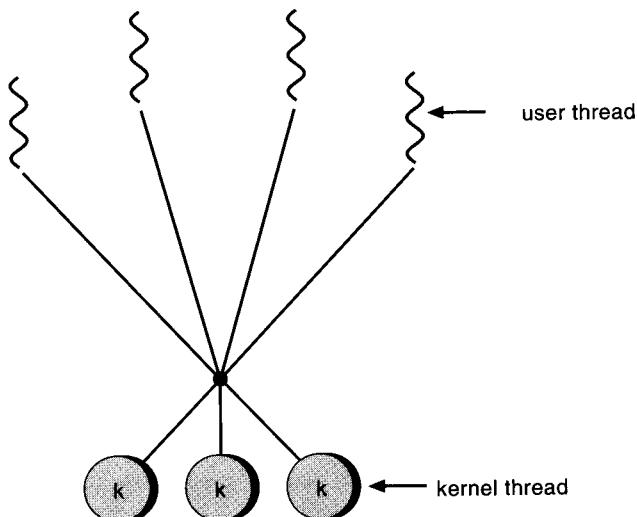


Figure 5.4 Many-to-many model.

5.3 ■ Threading Issues

In this section we discuss some of the issues to consider with multithreaded programs.

5.3.1 The fork and exec System Calls

In Chapter 4 we described how the `fork` system call is used to create a separate, duplicate process. In a multithreaded program, the semantics of the `fork` and `exec` system calls change. If one thread in a program calls `fork`, does the new process duplicate all threads or is the new process single-threaded? Some UNIX systems have chosen to have two versions of `fork`, one that duplicates all threads and another that duplicates only the thread that invoked the `fork` system call. The `exec` system call typically works in the same way as described in Chapter 4. That is, if a thread invokes the `exec` system call, the program specified in the parameter to `exec` will replace the entire process—including all threads and LWPs.

Usage of the two versions of `fork` depends upon the application. If `exec` is called immediately after forking, then duplicating all threads is unnecessary, as the program specified in the parameters to `exec` will replace the process. In this instance, duplicating only the calling thread is appropriate. If, however, the separate process does not call `exec` after forking, the separate process should duplicate all threads.

5.3.2 Cancellation

Thread cancellation is the task of terminating a thread before it has completed. For example, if multiple threads are concurrently searching through a database and one thread returns the result, the remaining threads might be cancelled. Another situation might occur when a user presses a button on a web browser that stops a web page from loading any further. Often a web page is loaded in a separate thread. When a user presses the *stop* button, the thread loading the page is cancelled.

A thread that is to be cancelled is often referred to as the **target thread**. Cancellation of a target thread may occur in two different scenarios:

1. **Asynchronous cancellation:** One thread immediately terminates the target thread.
2. **Deferred cancellation:** The target thread can periodically check if it should terminate, allowing the target thread an opportunity to terminate itself in an orderly fashion.

The difficulty with cancellation occurs in situations where resources have been allocated to a cancelled thread or if a thread was cancelled while in

the middle of updating data it is sharing with other threads. This becomes especially troublesome with asynchronous cancellation. The operating system will often reclaim system resources from a cancelled thread, but often will not reclaim all resources. Therefore, cancelling a thread asynchronously may not free a necessary system-wide resource.

Alternatively, deferred cancellation works by one thread indicating that a target thread is to be cancelled. However, cancellation will occur only when the target thread checks to determine if it should be cancelled or not. This allows a thread to check if it should be cancelled at a point when it can safely be cancelled. Pthreads refers to such points as **cancellation points**.

Most operating systems allow a process or thread to be cancelled asynchronously. However, the Pthread API provides deferred cancellation. This means that an operating system implementing the Pthread API will allow deferred cancellation.

5.3.3 Signal Handling

A **signal** is used in UNIX systems to notify a process that a particular event has occurred. A signal may be received either synchronously or asynchronously, depending upon the source and the reason for the event being signalled. Whether a signal is synchronous or asynchronous, all signals follow the same pattern:

1. A signal is generated by the occurrence of a particular event.
2. A generated signal is delivered to a process.
3. Once delivered, the signal must be handled.

An example of a synchronous signal includes an illegal memory access or division by zero. In this instance, if a running program performs either of these actions, a signal is generated. Synchronous signals are delivered to the same process that performed the operation causing the signal (hence the reason they are considered synchronous).

When a signal is generated by an event external to a running process, that process receives the signal asynchronously. Examples of such signals include terminating a process with specific keystrokes (such as <control><C>) or having a timer expire. Typically an asynchronous signal is sent to another process.

Every signal may be *handled* by one of two possible handlers:

1. A default signal handler
2. A user-defined signal handler

Every signal has a **default signal handler** that is run by the kernel when handling the signal. This default action may be overridden by a **user-defined signal handler** function. In this instance, the user-defined function is called to handle the signal rather than the default action. Both synchronous and asynchronous signals may be handled in different ways. Some signals may be simply ignored (such as changing the size of a window); others may be handled by terminating the program (such as an illegal memory access).

Handling signals in single-threaded programs is straightforward; signals are always delivered to a process. However, delivering signals is more complicated in multithreaded programs, as a process may have several threads. Where then should a signal be delivered?

In general, the following options exist:

1. Deliver the signal to the thread to which the signal applies.
2. Deliver the signal to every thread in the process.
3. Deliver the signal to certain threads in the process.
4. Assign a specific thread to receive all signals for the process.

The method for delivering a signal depends upon the type of signal generated. For example, synchronous signals need to be delivered to the thread that generated the signal and not to other threads in the process. However, the situation with asynchronous signals is not as clear. Some asynchronous signals—such as a signal that terminates a process (`<control><C>`, for example)—should be sent to all threads. Some multithreaded versions of UNIX allow a thread to specify which signals it will accept and which it will block. Therefore, some asynchronous signals may be delivered to only those threads that are not blocking the signal. However, because signals need to be handled only once, typically a signal is delivered only to the first thread found in a process that is not blocking the signal. Solaris 2 implements the fourth option: it creates a specific thread within each process solely for signal handling. When an asynchronous signal is sent to a process, it is sent to this special thread, which then delivers the signal to the first thread that is not blocking the signal.

Although Windows 2000 does not explicitly provide support for signals, they can be emulated using **asynchronous procedure calls (APCs)**. The APC facility allows a user thread to specify a function that is to be called when the user thread receives notification of a particular event. As indicated by its name, an APC is roughly equivalent to an asynchronous signal in UNIX. However, whereas UNIX must contend with how to deal with signals in a multithreaded environment, the APC facility is more straightforward as an APC is delivered to a particular thread rather than process.

5.3.4 Thread Pools

In Section 5.1, we described the scenario of multithreading a web server. In this situation, whenever the server receives a request, it creates a separate thread to service the request. Whereas creating a separate thread is certainly superior to creating a separate process, a multithreaded server nonetheless has potential problems. The first concerns the amount of time required to create the thread prior to servicing the request, compounded with the fact that this thread will be discarded once it has completed its work. The second issue is more problematic: If we allow all concurrent requests to be serviced in a new thread, we have not placed a bound on the number of threads concurrently active in the system. Unlimited threads could exhaust system resources, such as CPU time or memory. One solution to this issue is to use **thread pools**.

The general idea behind a thread pool is to create a number of threads at process startup and place them into a *pool*, where they sit and wait for work. When a server receives a request, it awakens a thread from this pool—if one is available—passing it the request to service. Once the thread completes its service, it returns to the pool awaiting more work. If the pool contains no available thread, the server waits until one becomes free.

In particular, the benefits of thread pools are:

1. It is usually faster to service a request with an existing thread than waiting to create a thread.
2. A thread pool limits the number of threads that exist at any one point. This is particularly important on systems that cannot support a large number of concurrent threads.

The number of threads in the pool can be set heuristically based upon factors such as the number of CPUs in the system, the amount of physical memory, and the expected number of concurrent client requests. More sophisticated thread-pool architectures can dynamically adjust the number of threads in the pool according to usage patterns. Such architectures provide the further benefit of having a smaller pool—thereby consuming less memory—when the load on the system is low.

5.3.5 Thread-Specific Data

Threads belonging to a process share the data of the process. Indeed, this sharing of data provides one of the benefits of multithreaded programming. However, each thread might need its own copy of certain data in some circumstances. We will call such data **thread-specific data**. For example, in a transaction-processing system, we might service each transaction in a separate thread. Furthermore, each transaction may be assigned a unique identifier. To

associate each thread with its unique identifier we could use thread-specific data. Most thread libraries—including Win32 and Pthreads—provide some form of support for thread-specific data. Java provides support as well.

5.4 ■ Pthreads

Pthreads refers to the POSIX standard (IEEE 1003.1c) defining an API for thread creation and synchronization. This is a *specification* for thread behavior, not an *implementation*. Operating system designers may implement the specification in any way they wish. Generally, libraries implementing the Pthreads specification are restricted to UNIX-based systems such as Solaris 2. The Windows operating systems have generally not supported Pthreads, although *shareware* versions are available in the public domain.

In this section we introduce some of the Pthread API as an example of a user-level thread library. We refer to it as a user-level library because no distinct relationship exists between a thread created using the Pthread API and any associated kernel threads. The C program shown in Figure 5.5 demonstrates the basic Pthread API for constructing a multithreaded program. If you are interested in more details on the Pthread API, we encourage you to consult the Bibliographical Notes.

The program shown in Figure 5.5 creates a separate thread that determines the summation of a non-negative integer. In a Pthread program, separate threads begin execution in a specified function. In Figure 5.5, this is the `runner` function. When this program begins, a single thread of control begins in `main`. After some initialization, `main` creates a second thread that begins control in the `runner` function.

We will now provide a more detailed overview of this program. All Pthread programs must include the `pthread.h` header file. The statement `pthread_t tid` declares the identifier for the thread we will create. Each thread has a set of attributes including stack size and scheduling information. The `pthread_attr_t attr` declaration represents the attributes for the thread. We will set the attributes in the function call `pthread_attr_init(&attr)`. Because we did not explicitly set any attributes, we will use the default attributes provided. A separate thread is created with the `pthread_create` function call. In addition to passing the thread identifier and the attributes for the thread, we also pass the name of the function where the new thread will begin execution, in this case the `runner` function. Lastly, we pass the integer parameter that was provided on the command line, `argv[1]`.

At this point, the program has two threads: the initial thread in `main` and the thread performing the summation in the `runner` function. After creating the second thread, the `main` thread will wait for the `runner` thread to complete by calling the `pthread_join` function. The `runner` thread will complete when it calls the function `pthread_exit`.

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

int sum; /* this data is shared by the thread(s) */
void *runner(void *param); /* the thread */

main(int argc, char *argv[])
{
    pthread_t tid; /* the thread identifier */
    pthread_attr_t attr; /* set of thread attributes */
    if (argc != 2) {
        fprintf(stderr,"usage: a.out <integer value>\n");
        exit();
    }
    if (atoi(argv[1]) < 0) {
        fprintf(stderr,"%d must be <= 0\n",atoi(argv[1]));
        exit();
    }
    /* get the default attributes */
    pthread_attr_init(&attr);
    /* create the thread */
    pthread_create(&tid,&attr,runner,argv[1]);
    /* now wait for the thread to exit */
    pthread_join(tid,NULL);
    printf("sum = %d\n",sum);
}

/* The thread will begin control in this function */
void *runner(void *param)
{
    int upper = atoi(param);
    int i;
    sum = 0;
    if (upper > 0) {
        for (i = 1; i <= upper; i++)
            sum += i;
    }
    pthread_exit(0);
}
```

Figure 5.5 Multithreaded C program using the Pthread API.

5.5 ■ Solaris 2 Threads

Solaris 2 is a version of UNIX with support for threads at the kernel and user levels, SMP, and real-time scheduling. Solaris 2 implements the Pthread API discussed in Section 5.4 in addition to supporting user-level threads with a library containing APIs for thread creation and management (known as *UI threads*). The differences between these two libraries are insignificant, although most developers now opt for the Pthread library. Solaris 2 defines an intermediate level of threads as well. Between user- and kernel-level threads are lightweight processes (LWPs). Each process contains at least one LWP. The thread library multiplexes user-level threads on the pool of LWPs for the process, and only user-level threads currently connected to an LWP accomplish work. The rest are either blocked or waiting for an LWP on which they can run.

Standard kernel-level threads execute all operations within the kernel. Each LWP has a kernel-level thread, and some kernel-level threads run on the kernel's behalf and have no associated LWP (for instance, a thread to service disk requests). Kernel-level threads are the only objects scheduled within the system (Chapter 6). Solaris 2 implements the many-to-many model; its entire thread system is depicted in Figure 5.6.

User-level threads may be either bound or unbound. A **bound** user-level thread is permanently attached to an LWP. Only that thread runs on the LWP, and by request the LWP can be dedicated to a single processor (see the rightmost thread in Figure 5.6). Binding a thread is useful in situations that require quick response time, such as a real-time application. An **unbound** thread is not permanently attached to any LWP. All unbound threads in an application are multiplexed onto the pool of available LWPs for the application. Threads are

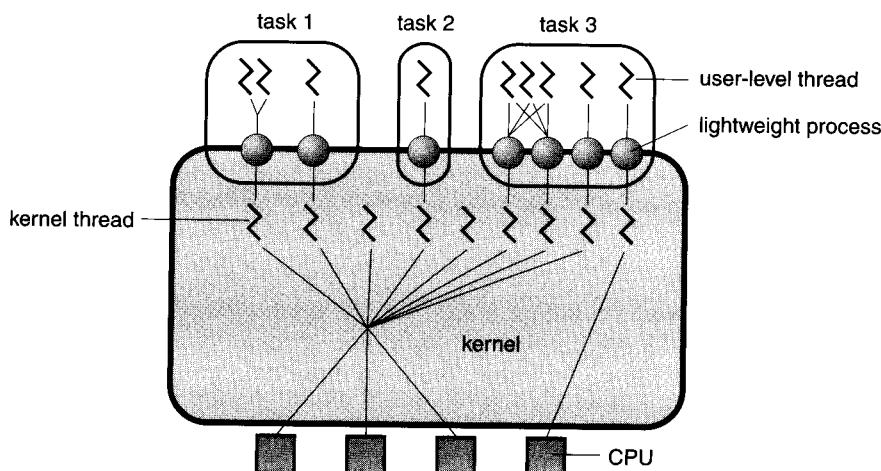


Figure 5.6 Solaris 2 threads.

unbound by default. Solaris 8 also provides an alternate thread library that, by default, binds all threads with an associated LWP.

Consider the system in operation: Any one process may have many user-level threads. These user-level threads may be scheduled and switched among the LWPs by the thread library without kernel intervention. User-level threads are extremely efficient because no kernel support is required for thread creation or destruction, or for the thread library to context switch from one user-level thread to another.

Each LWP is connected to exactly one kernel-level thread, whereas each user-level thread is independent of the kernel. Many LWPs may be in a process, but they are needed only when the thread needs to communicate with the kernel. For instance, one LWP is needed for every thread that may block concurrently in system calls. Consider five different file-read requests that occur simultaneously. Then, five LWPs would be needed, because they could all be waiting for I/O completion in the kernel. If a task had only four LWPs, then the fifth request would have to wait for one of the LWPs to return from the kernel. Adding a sixth LWP would gain nothing if there were only enough work for five.

The kernel threads are scheduled by the kernel's scheduler and execute on the CPU or CPUs in the system. If a kernel thread blocks (such as while waiting for an I/O operation to complete), the processor is free to run another kernel thread. If the thread that blocked was running on behalf of an LWP, the LWP blocks as well. Up the chain, the user-level thread currently attached to the LWP also blocks. If a process has more than one LWP, the kernel can schedule another LWP.

The thread library dynamically adjusts the number of LWPs in the pool to ensure the best performance for the application. For example, if all the LWPs in a process are blocked and other threads are able to run, the thread library automatically creates another LWP to be assigned to a waiting thread. A program is thus prevented from being blocked by a lack of unblocked LWPs. Also, LWPs are expensive kernel resources to maintain if they are not being used. The thread library "ages" LWPs and deletes them when they are unused for a long time, typically about 5 minutes.

The developers used the following data structures to implement threads on Solaris 2:

- A user-level thread contains a thread ID; register set (including a program counter and stack pointer); stack; and priority (used by the thread library for scheduling purposes). None of these data structures are kernel resources; all exist within user space.
- An LWP has a register set for the user-level thread it is running, as well as memory and accounting information. An LWP is a kernel data structure, and it resides in kernel space.

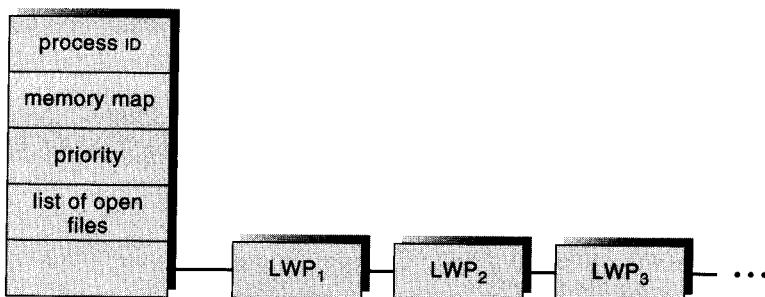


Figure 5.7 Solaris 2 process.

- A kernel thread has only a small data structure and a stack. The data structure includes a copy of the kernel registers, a pointer to the LWP to which it is attached, and priority and scheduling information.

Each process in Solaris 2 contains much of the information described in the process control block (PCB), which was discussed in Section 4.1.3. In particular, a Solaris 2 process contains a process ID (PID); memory map; list of open files; priority; and pointer to a list of kernel threads associated with the process (Figure 5.7).

5.6 ■ Window 2000 Threads

Windows 2000 implements the Win32 API. The Win32 API is the primary API for the family of Microsoft operating systems (Windows 95/98/NT and Windows 2000). Indeed, much of what is mentioned in this section applies to this family of operating systems.

A Windows application runs as a separate process where each process may contain one or more threads. Windows 2000 uses the one-to-one mapping described in Section 5.2.2 where each user-level thread maps to an associated kernel thread. However, Windows also provides support for a **fiber** library, which provides the functionality of the many-to-many model (Section 5.2.3). Every thread belonging to a process can access the virtual address space of the process.

The general components of a thread include:

- A thread ID uniquely identifying the thread.
- A register set representing the status of the processor.
- A user stack used when the thread is running in user mode. Similarly, each thread also has a kernel stack used when the thread is running in kernel mode.
- A private storage area used by various run-time libraries and dynamic link libraries (DLLs).

The register set, stacks, and private storage area are known as the *context* of the thread and are architecture-specific to the hardware on which the operating system is running. The primary data structures of a thread include:

- ETHREAD (executive thread block).
- KTHREAD (kernel thread block).
- TEB (thread environment block).

The key components of the ETHREAD include a pointer to the process to which the thread belongs and the address of the routine in which the thread starts control. The ETHREAD also contains a pointer to the corresponding KTHREAD.

The KTHREAD includes scheduling and synchronization information for the thread. In addition, the KTHREAD includes the kernel stack (used when the thread is running in kernel mode) and a pointer to the TEB.

The ETHREAD and the KTHREAD exist entirely in kernel space; this means only the kernel can access them. The TEB is a user-space data structure that is accessed when the thread is running in user mode. Among other fields, the TEB contains a user mode stack and an array for thread-specific data (which Windows terms *thread-local storage*).

5.7 ■ Linux Threads

The Linux kernel introduced threads in version 2.2. Linux provides a `fork` system call with the traditional functionality of duplicating a process. Linux also provides the `clone` system call that is analogous to creating a thread. `clone` behaves much like `fork`, except that instead of creating a copy of the calling process, it creates a separate process that shares the address space of the calling process. It is through this sharing of the address space of the parent process that a cloned task behaves much like a separate thread.

The sharing of the address space is allowed because of the representation of a process in the Linux kernel. A unique kernel data structure exists for each process in the system. However, rather than storing the data for each process in this data structure, it contains pointers to other data structures where this data is stored. For example, this per-process data structure contains pointers to other data structures that represent the list of open files, signal-handling information, and virtual memory. When `fork` is invoked, a new process is created along with a *copy* of all the associated data structures of the parent process. When the `clone` system call is made, a new process is created. However, rather than copying all data structures, the new process *points* to the data structures of the parent process, thereby allowing the child process to share the memory and other process resources of the parent. A set of flags is passed as a parameter to the `clone` system call. This set of flags is used to indicate how much of

the parent process is to be shared with the child. If none of the flags is set, no sharing occurs and `clone` acts just like `fork`. If all five flags are set, the child process shares everything with the parent. Other combinations of the flags allow various levels of sharing between these two extremes.

Interestingly, Linux does not distinguish between processes and threads. In fact, Linux generally uses the term *task*—rather than process or thread—when referring to a flow of control within a program. Aside from the cloned process, Linux does not support multithreading, separate data structures, or kernel routines. However, various Pthreads implementations are available for user-level multithreading.

5.8 ■ Java Threads

As we have already seen, support for threads may be provided at the user level with a library such as Pthreads. Furthermore, most operating systems provide support for threads at the kernel level as well. Java is one of a small number of languages that provide support at the language level for the creation and management of threads. However, because threads are managed by the Java Virtual Machine (JVM), not by a user-level library or kernel, it is difficult to classify Java threads as either user- or kernel-level. In this section we present Java threads as an alternative to the strict user- or kernel-level models. Later in this section, we will discuss how a Java thread may be mapped to the underlying kernel thread.

All Java programs comprise at least a single thread of control. Even a simple Java program consisting of only a `main` method runs as a single thread in the JVM. In addition, Java provides commands that allow the developer to create and manipulate additional threads of control within the program.

5.8.1 Thread Creation

One way to create a thread explicitly is to create a new class that is derived from the `Thread` class, and to override the `run` method of the `Thread` class. This approach is shown in Figure 5.8, the Java version of a multithreaded program that determines the summation of a non-negative integer.

An object of this derived class will run as a separate thread of control in the JVM. However, creating an object that is derived from the `Thread` class does not specifically create the new thread; rather, it is the `start` method that actually creates the new thread. Calling the `start` method for the new object does two things:

1. It allocates memory and initializes a new thread in the JVM.
2. It calls the `run` method, making the thread eligible to be run by the JVM. (Note that you do not ever call the `run` method directly. Rather, call the `start` method, and it will call the `run` method on your behalf.)

```

class Summation extends Thread
{
    public Summation(int n) {
        upper = n;
    }

    public void run() {
        int sum = 0;

        if (upper > 0) {
            for (int i = 1; i <= upper; i++)
                sum += i;
        }

        System.out.println("The sum of "+upper+" is "+sum);
    }

    private int upper;
}

public class ThreadTester
{
    public static void main(String[] args) {
        if (args.length > 0) {
            if (Integer.parseInt(args[0]) < 0)
                System.err.println(args[0] + " must be >= 0.");
            else {
                Summation thrd = new Summation(Integer.parseInt(args[0]));
                thrd.start();
            }
        }
        else
            System.err.println("Usage: Summation <integer value>");
    }
}

```

Figure 5.8 Java program for the summation of a non-negative integer.

When the summation program runs, two threads are created by the JVM. The first is the thread associated with the application—the thread that starts execution at the `main` method. The second thread is the `Summation` thread that is created explicitly with the `start` method. The `Summation` thread begins execution in its `run` method. The thread terminates when it exits from its `run` method.

5.8.2 The JVM and the Host Operating System

The typical implementation of the JVM is on top of a host operating system. This setup allows the JVM to hide the implementation details of the underlying operating system and to provide a consistent, abstract environment that allows Java programs to operate on any platform that supports a JVM. The specification for the JVM does not indicate how Java threads are to be mapped to the underlying operating system, instead leaving that decision to the particular implementation of the JVM. Windows 95/98/NT and Windows 2000 use the one-to-one model; therefore, each Java thread for a JVM running on these operating systems maps to a kernel thread. Solaris 2 initially implemented the JVM using the many-to-one model (called *green threads*). However, as of Version 1.1 of the JVM with Solaris 2.6, it was implemented using the many-to-many model.

5.9 ■ Summary

A thread is a flow of control within a process. A multithreaded process contains several different flows of control within the same address space. The benefits of multithreading include increased responsiveness to the user, resource sharing within the process, economy, and the ability to take advantage of multiprocessor architectures.

User-level threads are threads that are visible to the programmer and are unknown to the kernel. A thread library in user space typically manages user-level threads. The operating-system kernel supports and manages kernel-level threads. In general, user-level threads are faster to create and manage than are kernel threads. Three different types of models relate user and kernel threads: The many-to-one model maps many user threads to a single kernel thread. The one-to-one model maps each user thread to a corresponding kernel thread. The many-to-many model multiplexes many user threads to a smaller or equal number of kernel threads.

Multithreaded programs introduce many challenges for the programmer, including the semantics of the `fork` and `exec` system calls. Other issues include thread cancellation, signal handling, and thread-specific data. Many modern operating systems provide kernel support for threads; among these are Windows NT and Windows 2000, Solaris 2, and Linux. The Pthread API provides a set of functions to create and manage threads at the user level. Java provides a similar API for supporting threads. However, because Java threads are managed by the JVM and not by a user-level thread library or kernel, they do not fall under the category of either user- or kernel-level threads.

■ Exercises

- 5.1 Provide two programming examples of multithreading that improve performance over a single-threaded solution.

- 5.2 Provide two programming examples of multithreading that do *not* improve performance over a single-threaded solution.
- 5.3 What are two differences between user-level threads and kernel-level threads? Under what circumstances is one type better than the other?
- 5.4 Describe the actions taken by a kernel to context switch between kernel-level threads.
- 5.5 Describe the actions taken by a thread library to context switch between user-level threads.
- 5.6 What resources are used when a thread is created? How do they differ from those used when a process is created?
- 5.7 Assume an operating system maps user-level threads to the kernel using the many-to-many model where the mapping is done through LWPs. Furthermore, the system allows the developers to create real-time threads. Is it necessary to bind a real-time thread to an LWP? Explain.
- 5.8 Write a multithreaded Pthread or Java program that generates the Fibonacci series. This program should work as follows: The user will run the program and will enter on the command line the number of Fibonacci numbers that the program is to generate. The program will then create a separate thread that will generate the Fibonacci numbers.
- 5.9 Write a multithreaded Pthread or Java program that outputs prime numbers. This program should work as follows: The user will run the program and will enter a number on the command line. The program will then create a separate thread that outputs all the prime numbers less than or equal to the number that the user entered.

Bibliographical Notes

Thread performance issues were discussed by Anderson et al. [1989], who continued their work in Anderson et al. [1991] by evaluating the performance of user-level threads with kernel support. Marsh et al. [1991] discussed first-class user-level threads. Bershad et al. [1990] described combining threads with RPC. Draves et al. [1991] discussed the use of continuations to implement thread management and communication in operating systems. Engelschall [2000] discusses a technique for supporting user-level threads. An analysis of an optimal thread pool size can be found in Ling et al. [2000].

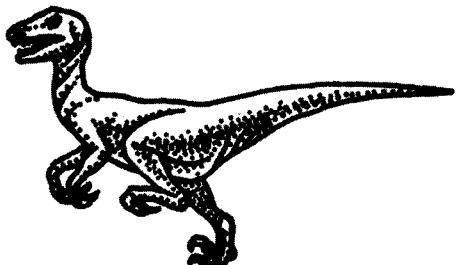
The IBM OS/2 operating system is a multithreaded operating system that runs on personal computers (Kogan and Rawson [1988]). Peacock [1992] discussed the multithreading of the file system in Solaris 2. Vahalia [1996] covers threading in several versions of UNIX. Mauro and McDougall [2001] describe

recent developments in threading the Solaris 2 kernel. Zabatta and Young [1998] compare Windows NT and Solaris 2 threads on a symmetric multiprocessor.

Information on multithreaded programming is given in Lewis and Berg [1988], Sun [1995], and Kleiman et al. [1996], although these references tend to be biased toward Pthreads. Oaks and Wong [1999], Lea [2000], and Hartley [1998] discuss multithreading in Java. Solomon [1998] and Solomon and Russinovich [2000] describe how threads are implemented in Windows NT and Windows 2000, respectively. Beveridge and Wiener [1997] discuss multithreading using Win32, and Pham and Garg [1996] describe multithreading programming using Windows NT.

Chapter 6

CPU SCHEDULING



CPU scheduling is the basis of multiprogrammed operating systems. By switching the CPU among processes, the operating system can make the computer more productive. In this chapter, we introduce the basic scheduling concepts and present several different CPU-scheduling algorithms. We also consider the problem of selecting an algorithm for a particular system.

6.1 ■ Basic Concepts

The objective of multiprogramming is to have some process running at all times, in order to maximize CPU utilization. In a uniprocessor system, only one process may run at a time; any other processes must wait until the CPU is free and can be rescheduled.

The idea of multiprogramming is relatively simple. A process is executed until it must wait, typically for the completion of some I/O request. In a simple computer system, the CPU would then sit idle; all this waiting time is wasted. With multiprogramming, we try to use this time productively. Several processes are kept in memory at one time. When one process has to wait, the operating system takes the CPU away from that process and gives the CPU to another process. This pattern continues.

Scheduling is a fundamental operating-system function. Almost all computer resources are scheduled before use. The CPU is, of course, one of the primary computer resources. Thus, its scheduling is central to operating-system design.

6.1.1 CPU–I/O Burst Cycle

The success of CPU scheduling depends on the following observed property of processes: Process execution consists of a **cycle** of CPU execution and I/O wait. Processes alternate between these two states. Process execution begins with a **CPU burst**. That is followed by an **I/O burst**, then another CPU burst, then another I/O burst, and so on. Eventually, the last CPU burst will end with a system request to terminate execution, rather than with another I/O burst (Figure 6.1).

The durations of these CPU bursts have been extensively measured. Although they vary greatly by process and by computer, they tend to have a frequency curve similar to that shown in Figure 6.2. The curve is generally characterized as exponential or hyperexponential, with many short CPU bursts, and a few long CPU bursts. An I/O-bound program would typically have many very short CPU bursts. A CPU-bound program might have a few very long

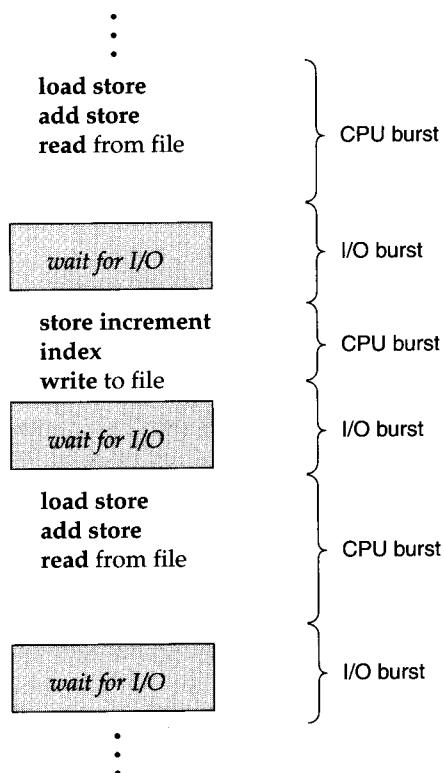


Figure 6.1 Alternating sequence of CPU and I/O bursts.

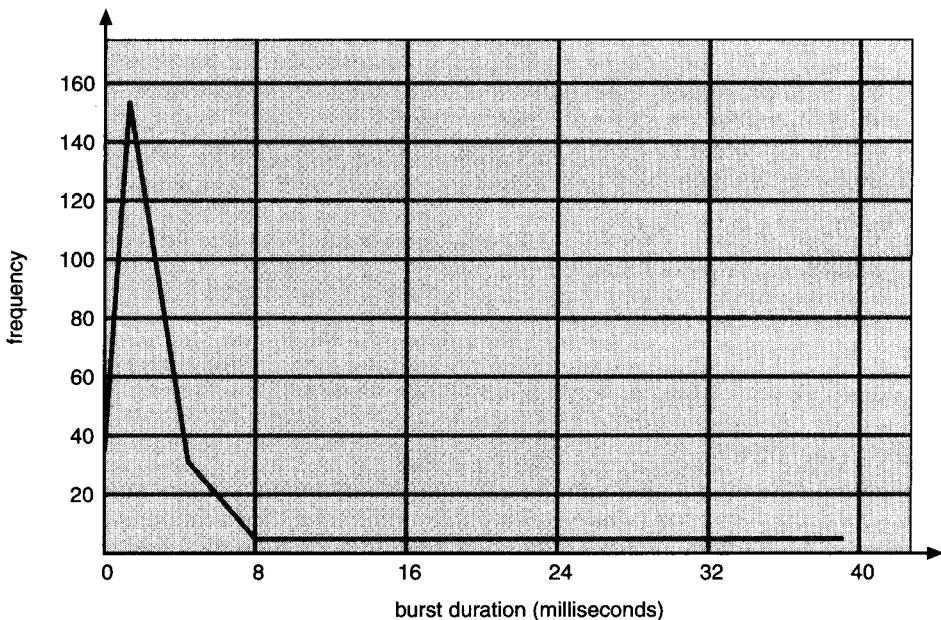


Figure 6.2 Histogram of CPU-burst times.

CPU bursts. This distribution can help us select an appropriate CPU-scheduling algorithm.

6.1.2 CPU Scheduler

Whenever the CPU becomes idle, the operating system must select one of the processes in the ready queue to be executed. The selection process is carried out by the **short-term scheduler** (or CPU scheduler). The scheduler selects from among the processes in memory that are ready to execute, and allocates the CPU to one of them.

The ready queue is not necessarily a first-in, first-out (FIFO) queue. As we shall see when we consider the various scheduling algorithms, a ready queue may be implemented as a FIFO queue, a priority queue, a tree, or simply an unordered linked list. Conceptually, however, all the processes in the ready queue are lined up waiting for a chance to run on the CPU. The records in the queues are generally process control blocks (PCBs) of the processes.

6.1.3 Preemptive Scheduling

CPU scheduling decisions may take place under the following four circumstances:

1. When a process switches from the running state to the waiting state (for example, I/O request, or invocation of wait for the termination of one of the child processes)
2. When a process switches from the running state to the ready state (for example, when an interrupt occurs)
3. When a process switches from the waiting state to the ready state (for example, completion of I/O)
4. When a process terminates

In circumstances 1 and 4, there is no choice in terms of scheduling. A new process (if one exists in the ready queue) must be selected for execution. There is a choice, however, in circumstances 2 and 3.

When scheduling takes place only under circumstances 1 and 4, we say the scheduling scheme is **nonpreemptive**; otherwise, the scheduling scheme is **preemptive**. Under nonpreemptive scheduling, once the CPU has been allocated to a process, the process keeps the CPU until it releases the CPU either by terminating or by switching to the waiting state. This scheduling method is used by the Microsoft Windows 3.1 and by the Apple Macintosh operating systems. It is the only method that can be used on certain hardware platforms, because it does not require the special hardware (for example, a timer) needed for preemptive scheduling.

Unfortunately, preemptive scheduling incurs a cost. Consider the case of two processes sharing data. One may be in the midst of updating the data when it is preempted and the second process is run. The second process may try to read the data, which are currently in an inconsistent state. New mechanisms thus are needed to coordinate access to shared data; this topic is discussed in Chapter 7.

Preemption also has an effect on the design of the operating-system kernel. During the processing of a system call, the kernel may be busy with an activity on behalf of a process. Such activities may involve changing important kernel data (for instance, I/O queues). What happens if the process is preempted in the middle of these changes, and the kernel (or the device driver) needs to read or modify the same structure? Chaos could ensue. Some operating systems, including most versions of UNIX, deal with this problem by waiting either for a system call to complete, or for an I/O block to take place, before doing a context switch. This scheme ensures that the kernel structure is simple, since the kernel will not preempt a process while the kernel data structures are in an inconsistent state. Unfortunately, this kernel-execution model is a poor one for supporting real-time computing and multiprocessing. These problems, and their solutions, are described in Sections 6.4 and 6.5.

In the case of UNIX, sections of code are still at risk. Because interrupts can, by definition, occur at any time, and because they cannot always be ignored

by the kernel, the sections of code affected by interrupts must be guarded from simultaneous use. The operating system needs to accept interrupts at almost all times, otherwise input might be lost or output overwritten. So that these code sections are not accessed concurrently by several processes, they disable interrupts at entry and reenable interrupts at exit. Unfortunately, disabling and enabling interrupts is time consuming, especially on multiprocessor systems. For systems to *scale* efficiently beyond a few CPUs, interrupt state changes must be minimized and fine-grained locking maximized. For instance, this is a challenge to the scalability of Linux.

6.1.4 Dispatcher

Another component involved in the CPU scheduling function is the **dispatcher**. The dispatcher is the module that gives control of the CPU to the process selected by the short-term scheduler. This function involves:

- Switching context
- Switching to user mode
- Jumping to the proper location in the user program to restart that program

The dispatcher should be as fast as possible, given that it is invoked during every process switch. The time it takes for the dispatcher to stop one process and start another running is known as the **dispatch latency**.

6.2 ■ Scheduling Criteria

Different CPU-scheduling algorithms have different properties and may favor one class of processes over another. In choosing which algorithm to use in a particular situation, we must consider the properties of the various algorithms.

Many criteria have been suggested for comparing CPU-scheduling algorithms. The characteristics used for comparison can make a substantial difference in the determination of the best algorithm. The criteria include the following:

- **CPU utilization:** We want to keep the CPU as busy as possible. CPU utilization may range from 0 to 100 percent. In a real system, it should range from 40 percent (for a lightly loaded system) to 90 percent (for a heavily used system).
- **Throughput:** If the CPU is busy executing processes, then work is being done. One measure of work is the number of processes completed per time unit, called **throughput**. For long processes, this rate may be 1 process per hour; for short transactions, throughput might be 10 processes per second.

- **Turnaround time:** From the point of view of a particular process, the important criterion is how long it takes to execute that process. The interval from the time of submission of a process to the time of completion is the **turnaround time**. Turnaround time is the sum of the periods spent waiting to get into memory, waiting in the ready queue, executing on the CPU, and doing I/O.
- **Waiting time:** The CPU-scheduling algorithm does not affect the amount of time during which a process executes or does I/O; it affects only the amount of time that a process spends waiting in the ready queue. Waiting time is the sum of the periods spent waiting in the ready queue.
- **Response time:** In an interactive system, turnaround time may not be the best criterion. Often, a process can produce some output fairly early, and can continue computing new results while previous results are being output to the user. Thus, another measure is the time from the submission of a request until the first response is produced. This measure, called **response time**, is the amount of time it takes to start responding, but not the time that it takes to output that response. The turnaround time is generally limited by the speed of the output device.

We want to maximize CPU utilization and throughput, and to minimize turnaround time, waiting time, and response time. In most cases, we optimize the average measure. However, in some circumstances we want to optimize the minimum or maximum values, rather than the average. For example, to guarantee that all users get good service, we may want to minimize the maximum response time.

For interactive systems (such as time-sharing systems), some analysts suggest that minimizing the **variance** in the response time is more important than minimizing the average response time. A system with reasonable and **predictable** response time may be considered more desirable than a system that is faster on the average, but is highly variable. However, little work has been done on CPU-scheduling algorithms to minimize variance.

As we discuss various CPU-scheduling algorithms, we want to illustrate their operation. An accurate illustration should involve many processes, each being a sequence of several hundred CPU bursts and I/O bursts. For simplicity of illustration, we consider only one CPU burst (in milliseconds) per process in our examples. Our measure of comparison is the average waiting time. More elaborate evaluation mechanisms are discussed in Section 6.6.

6.3 ■ Scheduling Algorithms

CPU scheduling deals with the problem of deciding which of the processes in the ready queue is to be allocated the CPU. In this section, we describe several of the many CPU-scheduling algorithms that exist.

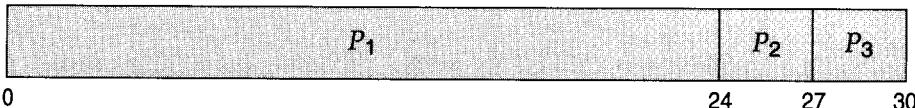
6.3.1 First-Come, First-Served Scheduling

By far the simplest CPU-scheduling algorithm is the **first-come, first-served (FCFS) scheduling algorithm**. With this scheme, the process that requests the CPU first is allocated the CPU first. The implementation of the FCFS policy is easily managed with a FIFO queue. When a process enters the ready queue, its PCB is linked onto the tail of the queue. When the CPU is free, it is allocated to the process at the head of the queue. The running process is then removed from the queue. The code for FCFS scheduling is simple to write and understand.

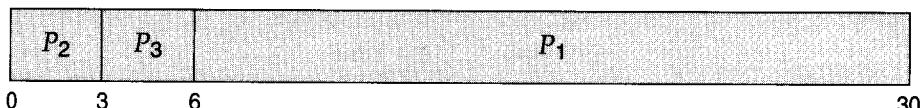
The average waiting time under the FCFS policy, however, is often quite long. Consider the following set of processes that arrive at time 0, with the length of the CPU-burst time given in milliseconds:

Process	Burst Time
P_1	24
P_2	3
P_3	3

If the processes arrive in the order P_1, P_2, P_3 , and are served in FCFS order, we get the result shown in the following Gantt chart:



The waiting time is 0 milliseconds for process P_1 , 24 milliseconds for process P_2 , and 27 milliseconds for process P_3 . Thus, the average waiting time is $(0 + 24 + 27)/3 = 17$ milliseconds. If the processes arrive in the order P_2, P_3, P_1 , however, the results will be as shown in the following Gantt chart:



The average waiting time is now $(6 + 0 + 3)/3 = 3$ milliseconds. This reduction is substantial. Thus, the average waiting time under a FCFS policy is generally

not minimal, and may vary substantially if the process CPU-burst times vary greatly.

In addition, consider the performance of FCFS scheduling in a dynamic situation. Assume we have one CPU-bound process and many I/O-bound processes. As the processes flow around the system, the following scenario may result. The CPU-bound process will get the CPU and hold it. During this time, all the other processes will finish their I/O and move into the ready queue, waiting for the CPU. While the processes wait in the ready queue, the I/O devices are idle. Eventually, the CPU-bound process finishes its CPU burst and moves to an I/O device. All the I/O-bound processes, which have very short CPU bursts, execute quickly and move back to the I/O queues. At this point, the CPU sits idle. The CPU-bound process will then move back to the ready queue and be allocated the CPU. Again, all the I/O processes end up waiting in the ready queue until the CPU-bound process is done. There is a **convoy effect**, as all the other processes wait for the one big process to get off the CPU. This effect results in lower CPU and device utilization than might be possible if the shorter processes were allowed to go first.

The FCFS scheduling algorithm is nonpreemptive. Once the CPU has been allocated to a process, that process keeps the CPU until it releases the CPU, either by terminating or by requesting I/O. The FCFS algorithm is particularly troublesome for time-sharing systems, where each user needs to get a share of the CPU at regular intervals. It would be disastrous to allow one process to keep the CPU for an extended period.

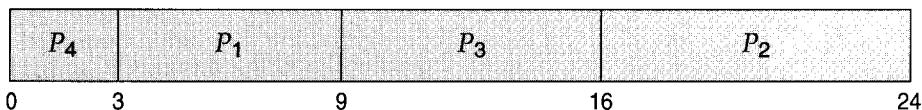
6.3.2 Shortest-Job-First Scheduling

A different approach to CPU scheduling is the **shortest-job-first (SJF) scheduling algorithm**. This algorithm associates with each process the length of the latter's next CPU burst. When the CPU is available, it is assigned to the process that has the smallest next CPU burst. If two processes have the same length next CPU burst, FCFS scheduling is used to break the tie. Note that a more appropriate term would be the *shortest next CPU burst*, because the scheduling is done by examining the length of the next CPU burst of a process, rather than its total length. We use the term SJF because most people and textbooks refer to this type of scheduling discipline as SJF.

As an example, consider the following set of processes, with the length of the CPU-burst time given in milliseconds:

Process	Burst Time
P_1	6
P_2	8
P_3	7
P_4	3

Using SJF scheduling, we would schedule these processes according to the following Gantt chart:



The waiting time is 3 milliseconds for process P_1 , 16 milliseconds for process P_2 , 9 milliseconds for process P_3 , and 0 milliseconds for process P_4 . Thus, the average waiting time is $(3 + 16 + 9 + 0)/4 = 7$ milliseconds. If we were using the FCFS scheduling scheme, then the average waiting time would be 10.25 milliseconds.

The SJF scheduling algorithm is provably *optimal*, in that it gives the minimum average waiting time for a given set of processes. By moving a short process before a long one, the waiting time of the short process decreases more than it increases the waiting time of the long process. Consequently, the *average* waiting time decreases.

The real difficulty with the SJF algorithm is knowing the length of the next CPU request. For long-term (or job) scheduling in a batch system, we can use as the length the process time limit that a user specifies when he submits the job. Thus, users are motivated to estimate the process time limit accurately, since a lower value may mean faster response. (Too low a value will cause a time-limit-exceeded error and require resubmission.) SJF scheduling is used frequently in long-term scheduling.

Although the SJF algorithm is optimal, it cannot be implemented at the level of short-term CPU scheduling. There is no way to know the length of the next CPU burst. One approach is to try to approximate SJF scheduling. We may not *know* the length of the next CPU burst, but we may be able to *predict* its value. We expect that the next CPU burst will be similar in length to the previous ones. Thus, by computing an approximation of the length of the next CPU burst, we can pick the process with the shortest predicted CPU burst.

The next CPU burst is generally predicted as an exponential average of the measured lengths of previous CPU bursts. Let t_n be the length of the n th CPU burst, and let τ_{n+1} be our predicted value for the next CPU burst. Then, for α , $0 \leq \alpha \leq 1$, define

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

This formula defines an **exponential average**. The value of t_n contains our most recent information; τ_n stores the past history. The parameter α controls the relative weight of recent and past history in our prediction. If $\alpha = 0$, then $\tau_{n+1} = \tau_n$, and recent history has no effect (current conditions are assumed to be transient); if $\alpha = 1$, then $\tau_{n+1} = t_n$, and only the most recent CPU burst matters

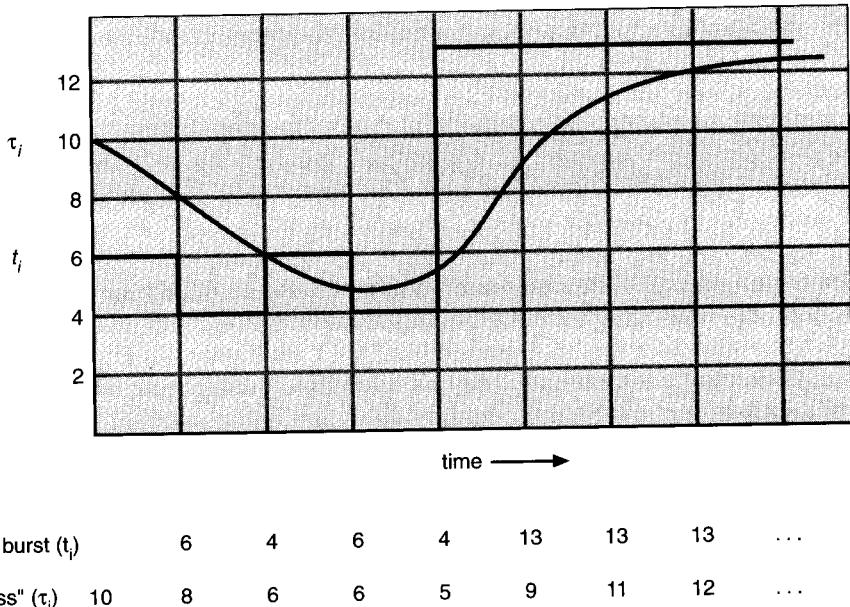


Figure 6.3 Prediction of the length of the next CPU burst.

(history is assumed to be old and irrelevant). More commonly, $\alpha = 1/2$, so recent history and past history are equally weighted. The initial τ_0 can be defined as a constant or as an overall system average. Figure 6.3 shows an exponential average with $\alpha = 1/2$ and $\tau_0 = 10$.

To understand the behavior of the exponential average, we can expand the formula for τ_{n+1} by substituting for τ_n , to find

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

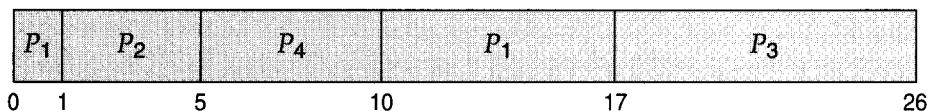
Since both α and $(1 - \alpha)$ are less than or equal to 1, each successive term has less weight than its predecessor.

The SJF algorithm may be either *preemptive* or *nonpreemptive*. The choice arises when a new process arrives at the ready queue while a previous process is executing. The new process may have a shorter next CPU burst than what is left of the currently executing process. A preemptive SJF algorithm will preempt the currently executing process, whereas a nonpreemptive SJF algorithm will allow the currently running process to finish its CPU burst. Preemptive SJF scheduling is sometimes called **shortest-remaining-time-first** scheduling.

As an example, consider the following four processes, with the length of the CPU-burst time given in milliseconds:

Process	Arrival Time	Burst Time
P_1	0	8
P_2	1	4
P_3	2	9
P_4	3	5

If the processes arrive at the ready queue at the times shown and need the indicated burst times, then the resulting preemptive SJF schedule is as depicted in the following Gantt chart:



Process P_1 is started at time 0, since it is the only process in the queue. Process P_2 arrives at time 1. The remaining time for process P_1 (7 milliseconds) is larger than the time required by process P_2 (4 milliseconds), so process P_1 is preempted, and process P_2 is scheduled. The average waiting time for this example is $((10 - 1) + (1 - 1) + (17 - 2) + (5 - 3)) / 4 = 26 / 4 = 6.5$ milliseconds. A nonpreemptive SJF scheduling would result in an average waiting time of 7.75 milliseconds.

6.3.3 Priority Scheduling

The SJF algorithm is a special case of the general **priority-scheduling algorithm**. A priority is associated with each process, and the CPU is allocated to the process with the highest priority. Equal-priority processes are scheduled in FCFS order.

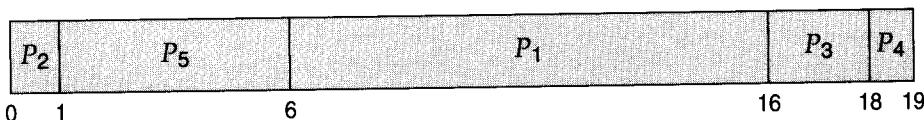
An SJF algorithm is simply a priority algorithm where the priority (p) is the inverse of the (predicted) next CPU burst. The larger the CPU burst, the lower the priority, and vice versa.

Note that we discuss scheduling in terms of *high* priority and *low* priority. Priorities are generally some fixed range of numbers, such as 0 to 7, or 0 to 4,095. However, there is no general agreement on whether 0 is the highest or lowest priority. Some systems use low numbers to represent low priority; others use low numbers for high priority. This difference can lead to confusion. In this text, we use low numbers to represent high priority.

As an example, consider the following set of processes, assumed to have arrived at time 0, in the order P_1, P_2, \dots, P_5 , with the length of the CPU-burst time given in milliseconds:

Process	Burst Time	Priority
P_1	10	3
P_2	1	1
P_3	2	4
P_4	1	5
P_5	5	2

Using priority scheduling, we would schedule these processes according to the following Gantt chart:



The average waiting time is 8.2 milliseconds.

Priorities can be defined either internally or externally. Internally defined priorities use some measurable quantity or quantities to compute the priority of a process. For example, time limits, memory requirements, the number of open files, and the ratio of average I/O burst to average CPU burst have been used in computing priorities. External priorities are set by criteria that are external to the operating system, such as the importance of the process, the type and amount of funds being paid for computer use, the department sponsoring the work, and other, often political, factors.

Priority scheduling can be either preemptive or nonpreemptive. When a process arrives at the ready queue, its priority is compared with the priority of the currently running process. A preemptive priority-scheduling algorithm will preempt the CPU if the priority of the newly arrived process is higher than the priority of the currently running process. A nonpreemptive priority-scheduling algorithm will simply put the new process at the head of the ready queue.

A major problem with priority-scheduling algorithms is **indefinite blocking** (or **starvation**). A process that is ready to run but lacking the CPU can be considered blocked—waiting for the CPU. A priority-scheduling algorithm can leave some low-priority processes waiting indefinitely for the CPU. In a heavily loaded computer system, a steady stream of higher-priority processes can prevent a low-priority process from ever getting the CPU. Generally, one of two things will happen. Either the process will eventually be run (at 2 A.M. Sunday, when the system is finally lightly loaded), or the computer system will eventually crash and lose all unfinished low-priority processes. (Rumor has it that, when they shut down the IBM 7094 at MIT in 1973, they found a low-priority process that had been submitted in 1967 and had not yet been run.)

A solution to the problem of indefinite blockage of low-priority processes is **aging**. **Aging** is a technique of gradually increasing the priority of processes that wait in the system for a long time. For example, if priorities range from

127 (low) to 0 (high), we could decrement the priority of a waiting process by 1 every 15 minutes. Eventually, even a process with an initial priority of 127 would have the highest priority in the system and would be executed. In fact, it would take no more than 32 hours for a priority 127 process to age to a priority 0 process.

6.3.4 Round-Robin Scheduling

The **round-robin (RR) scheduling algorithm** is designed especially for time-sharing systems. It is similar to FCFS scheduling, but preemption is added to switch between processes. A small unit of time, called a **time quantum** (or **time slice**), is defined. A time quantum is generally from 10 to 100 milliseconds. The ready queue is treated as a circular queue. The CPU scheduler goes around the ready queue, allocating the CPU to each process for a time interval of up to 1 time quantum.

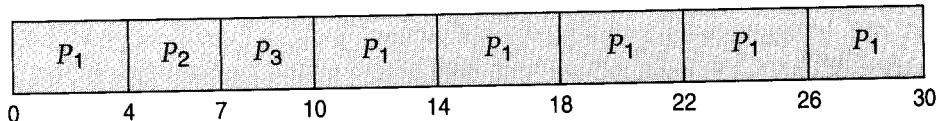
To implement RR scheduling, we keep the ready queue as a FIFO queue of processes. New processes are added to the tail of the ready queue. The CPU scheduler picks the first process from the ready queue, sets a timer to interrupt after 1 time quantum, and dispatches the process.

One of two things will then happen. The process may have a CPU burst of less than 1 time quantum. In this case, the process itself will release the CPU voluntarily. The scheduler will then proceed to the next process in the ready queue. Otherwise, if the CPU burst of the currently running process is longer than 1 time quantum, the timer will go off and will cause an interrupt to the operating system. A context switch will be executed, and the process will be put at the **tail** of the ready queue. The CPU scheduler will then select the next process in the ready queue.

The average waiting time under the RR policy, however, is often quite long. Consider the following set of processes that arrive at time 0, with the length of the CPU-burst time given in milliseconds:

Process	Burst Time
P_1	24
P_2	3
P_3	3

If we use a time quantum of 4 milliseconds, then process P_1 gets the first 4 milliseconds. Since it requires another 20 milliseconds, it is preempted after the first time quantum, and the CPU is given to the next process in the queue, process P_2 . Since process P_2 does not need 4 milliseconds, it quits before its time quantum expires. The CPU is then given to the next process, process P_3 . Once each process has received 1 time quantum, the CPU is returned to process P_1 for an additional time quantum. The resulting RR schedule is



The average waiting time is $17/3 = 5.66$ milliseconds.

In the RR scheduling algorithm, no process is allocated the CPU for more than 1 time quantum in a row. If a process' CPU burst exceeds 1 time quantum, that process is *preempted* and is put back in the ready queue. The RR scheduling algorithm is preemptive.

If there are n processes in the ready queue and the time quantum is q , then each process gets $1/n$ of the CPU time in chunks of at most q time units. Each process must wait no longer than $(n - 1) \times q$ time units until its next time quantum. For example, if there are five processes, with a time quantum of 20 milliseconds, then each process will get up to 20 milliseconds every 100 milliseconds.

The performance of the RR algorithm depends heavily on the size of the time quantum. At one extreme, if the time quantum is very large (infinite), the RR policy is the same as the FCFS policy. If the time quantum is very small (say 1 microsecond), the RR approach is called **processor sharing**, and appears (in theory) to the users as though each of n processes has its own processor running at $1/n$ the speed of the real processor. This approach was used in Control Data Corporation (CDC) hardware to implement 10 peripheral processors with only one set of hardware and 10 sets of registers. The hardware executes one instruction for one set of registers, then goes on to the next. This cycle continues, resulting in 10 slow processors rather than one fast processor. (Actually, since the processor was much faster than memory and each instruction referenced memory, the processors were not much slower than 10 real processors would have been.)

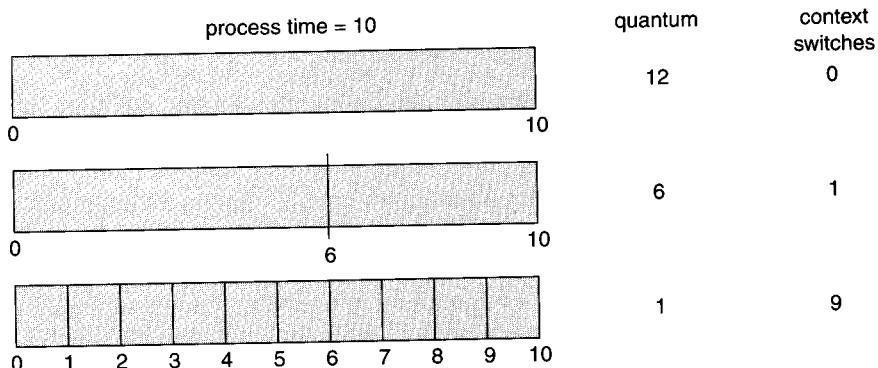


Figure 6.4 Showing how a smaller time quantum increases context switches.

In software, however, we need also to consider the effect of context switching on the performance of RR scheduling. Let us assume that we have only one process of 10 time units. If the quantum is 12 time units, the process finishes in less than 1 time quantum, with no overhead. If the quantum is 6 time units, however, the process requires 2 quanta, resulting in 1 context switch. If the time quantum is 1 time unit, then 9 context switches will occur, slowing the execution of the process accordingly (Figure 6.4).

Thus, we want the time quantum to be large with respect to the context-switch time. If the context-switch time is approximately 10 percent of the time quantum, then about 10 percent of the CPU time will be spent in context switch.

Turnaround time also depends on the size of the time quantum. As we can see from Figure 6.5, the average turnaround time of a set of processes does not necessarily improve as the time-quantum size increases. In general, the average turnaround time can be improved if most processes finish their next CPU burst in a single time quantum. For example, given three processes of 10 time units each and a quantum of 1 time unit, the average turnaround time is 29. If the time quantum is 10, however, the average turnaround time drops to 20. If context-switch time is added in, the average turnaround time increases for a smaller time quantum, since more context switches will be required.

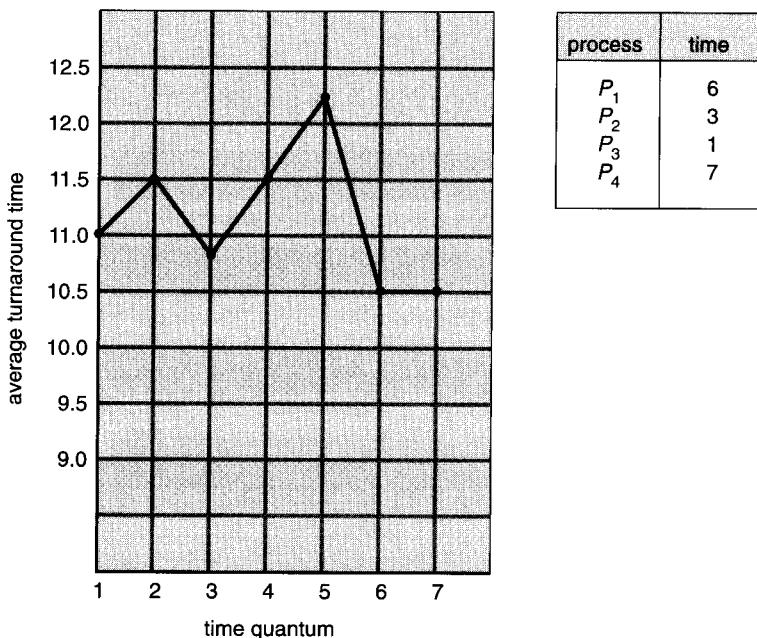


Figure 6.5 Showing how turnaround time varies with the time quantum.

On the other hand, if the time quantum is too large, RR scheduling degenerates to FCFS policy. A rule of thumb is that 80 percent of the CPU bursts should be shorter than the time quantum.

6.3.5 Multilevel Queue Scheduling

Another class of scheduling algorithms has been created for situations in which processes are easily classified into different groups. For example, a common division is made between **foreground** (or **interactive**) processes and **background** (or **batch**) processes. These two types of processes have different response-time requirements, and so might have different scheduling needs. In addition, foreground processes may have priority (or externally defined) over background processes.

A **multilevel queue-scheduling algorithm** partitions the ready queue into several separate queues (Figure 6.6). The processes are permanently assigned to one queue, generally based on some property of the process, such as memory size, process priority, or process type. Each queue has its own scheduling algorithm. For example, separate queues might be used for foreground and background processes. The foreground queue might be scheduled by an RR algorithm, while the background queue is scheduled by an FCFS algorithm.

In addition, there must be scheduling among the queues, which is commonly implemented as fixed-priority preemptive scheduling. For example, the foreground queue may have absolute priority over the background queue.

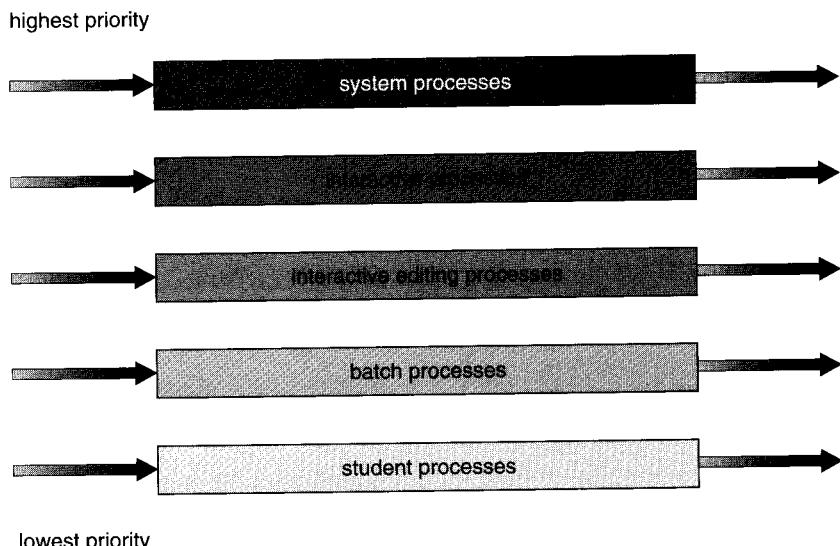


Figure 6.6 Multilevel queue scheduling.

Let us look at an example of a multilevel queue-scheduling algorithm with five queues:

1. System processes
2. Interactive processes
3. Interactive editing processes
4. Batch processes
5. Student processes

Each queue has absolute priority over lower-priority queues. No process in the batch queue, for example, could run unless the queues for system processes, interactive processes, and interactive editing processes were all empty. If an interactive editing process entered the ready queue while a batch process was running, the batch process would be preempted. Solaris 2 uses a form of this algorithm.

Another possibility is to time slice between the queues. Each queue gets a certain portion of the CPU time, which it can then schedule among the various processes in its queue. For instance, in the foreground-background queue example, the foreground queue can be given 80 percent of the CPU time for RR scheduling among its processes, whereas the background queue receives 20 percent of the CPU to give to its processes in a FCFS manner.

6.3.6 Multilevel Feedback Queue Scheduling

Normally, in a multilevel queue-scheduling algorithm, processes are permanently assigned to a queue on entry to the system. Processes do not move between queues. If there are separate queues for foreground and background processes, for example, processes do not move from one queue to the other, since processes do not change their foreground or background nature. This setup has the advantage of low scheduling overhead, but the disadvantage of being inflexible.

Multilevel feedback queue scheduling, however, allows a process to move between queues. The idea is to separate processes with different CPU-burst characteristics. If a process uses too much CPU time, it will be moved to a lower-priority queue. This scheme leaves I/O-bound and interactive processes in the higher-priority queues. Similarly, a process that waits too long in a lower-priority queue may be moved to a higher-priority queue. This form of aging prevents starvation.

For example, consider a multilevel feedback queue scheduler with three queues, numbered from 0 to 2 (Figure 6.7). The scheduler first executes all processes in queue 0. Only when queue 0 is empty will it execute processes in queue 1. Similarly, processes in queue 2 will be executed only if queues 0

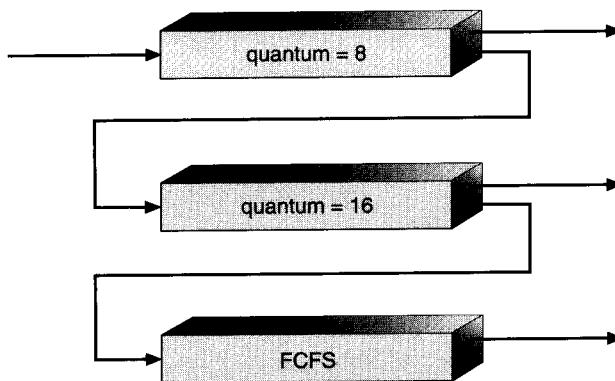


Figure 6.7 Multilevel feedback queues.

and 1 are empty. A process that arrives for queue 1 will preempt a process in queue 2. A process that arrives for queue 0 will, in turn, preempt a process in queue 1.

A process entering the ready queue is put in queue 0. A process in queue 0 is given a time quantum of 8 milliseconds. If it does not finish within this time, it is moved to the tail of queue 1. If queue 0 is empty, the process at the head of queue 1 is given a quantum of 16 milliseconds. If it does not complete, it is preempted and is put into queue 2. Processes in queue 2 are run on an FCFS basis, only when queues 0 and 1 are empty.

This scheduling algorithm gives highest priority to any process with a CPU burst of 8 milliseconds or less. Such a process will quickly get the CPU, finish its CPU burst, and go off to its next I/O burst. Processes that need more than 8, but less than 24, milliseconds are also served quickly, although with lower priority than shorter processes. Long processes automatically sink to queue 2 and are served in FCFS order with any CPU cycles left over from queues 0 and 1.

In general, a multilevel feedback queue scheduler is defined by the following parameters:

- The number of queues
- The scheduling algorithm for each queue
- The method used to determine when to upgrade a process to a higher-priority queue
- The method used to determine when to demote a process to a lower-priority queue
- The method used to determine which queue a process will enter when that process needs service

The definition of a multilevel feedback queue scheduler makes it the most general CPU-scheduling algorithm. It can be configured to match a specific system under design. Unfortunately, it also requires some means of selecting values for all the parameters to define the best scheduler. Although a multilevel feedback queue is the most general scheme, it is also the most complex.

6.4 ■ Multiple-Processor Scheduling

Our discussion thus far has focused on the problems of scheduling the CPU in a system with a single processor. If multiple CPUs are available, the scheduling problem is correspondingly more complex. Many possibilities have been tried, and, as we saw with single-processor CPU scheduling, there is no one best solution. In the following, we discuss briefly some of the issues concerning multiprocessor scheduling. (Complete coverage of multiprocessor scheduling is beyond the scope of this text; for more information, please refer to the Bibliographical Notes.) We concentrate on systems where the processors are identical (or **homogeneous**) in terms of their functionality; any available processor can then be used to run any processes in the queue. We also assume **uniform memory access (UMA)**. In Chapters 15 through 17 we discuss systems where processors are different (a **heterogeneous** system). Only programs compiled for a given processor's instruction set could be run on that processor.

Even within a homogeneous multiprocessor, there are sometimes limitations on scheduling. Consider a system with an I/O device attached to a private bus of one processor. Processes wishing to use that device must be scheduled to run on that processor, otherwise the device would not be available.

If several identical processors are available, then **load sharing** can occur. It would be possible to provide a separate queue for each processor. In this case, however, one processor could be idle, with an empty queue, while another processor was very busy. To prevent this situation, we use a common ready queue. All processes go into one queue and are scheduled onto any available processor.

In such a scheme, one of two scheduling approaches may be used. In one approach, each processor is self-scheduling. Each processor examines the common ready queue and selects a process to execute. As we shall see in Chapter 7, if we have multiple processors trying to access and update a common data structure, each processor must be programmed very carefully. We must ensure that two processors do not choose the same process, and that processes are not lost from the queue. The other approach avoids this problem by appointing one processor as scheduler for the other processors, thus creating a master-slave structure.

Some systems carry this structure one step further, by having all scheduling decisions, I/O processing, and other system activities handled by one single processor—the master server. The other processors only execute user code.

This **asymmetric multiprocessing** is far simpler than symmetric multiprocessing, because only one processor accesses the system data structures, alleviating the need for data sharing. However, it is also not as efficient. I/O-bound processes may bottleneck on the one CPU that is performing all of the operations. Typically, asymmetric multiprocessing is implemented first within an operating system, and is then upgraded to symmetric multiprocessing as the system evolves.

6.5 ■ Real-Time Scheduling

In Chapter 1, we gave an overview of real-time operating systems and discussed their growing importance. Here, we continue the discussion by describing the scheduling facility needed to support real-time computing within a general-purpose computer system.

Real-time computing is divided into two types. **Hard real-time** systems are required to complete a critical task within a guaranteed amount of time. Generally, a process is submitted along with a statement of the amount of time in which it needs to complete or perform I/O. The scheduler then either admits the process, guaranteeing that the process will complete on time, or rejects the request as impossible. This is known as **resource reservation**. Such a guarantee requires that the scheduler know exactly how long each type of operating-system function takes to perform, and therefore each operation must be guaranteed to take a maximum amount of time. Such a guarantee is impossible in a system with secondary storage or virtual memory, as we shall show in the next few chapters, because these subsystems cause unavoidable and unforeseeable variation in the amount of time to execute a particular process. Therefore, hard real-time systems are composed of special-purpose software running on hardware dedicated to their critical process, and lack the full functionality of modern computers and operating systems.

Soft real-time computing is less restrictive. It requires that critical processes receive priority over less fortunate ones. Although adding soft real-time functionality to a time-sharing system may cause an unfair allocation of resources and may result in longer delays, or even starvation, for some processes, it is at least possible to achieve. The result is a general-purpose system that can also support multimedia, high-speed interactive graphics, and a variety of tasks that would not function acceptably in an environment that does not support soft real-time computing.

Implementing soft real-time functionality requires careful design of the scheduler and related aspects of the operating system. First, the system must have priority scheduling, and real-time processes must have the highest priority. The priority of real-time processes must not degrade over time, even though the priority of non-real-time processes may. Second, the dispatch latency must be small. The smaller the latency, the faster a real-time process can start executing once it is runnable.

It is relatively simple to ensure that the former property holds. For example, we can disallow process aging on real-time processes, thereby guaranteeing that the priority of the various processes does not change. However, ensuring the latter property is much more involved. The problem is that many operating systems, including most versions of UNIX, are forced to wait either for a system call to complete or for an I/O block to take place before doing a context switch. The dispatch latency in such systems can be long, since some system calls are complex and some I/O devices are slow.

To keep dispatch latency low, we need to allow system calls to be preemptible. There are several ways to achieve this goal. One is to insert **preemption points** in long-duration system calls, that check to see whether a high-priority process needs to be run. If so, a context switch takes place and, when the high-priority process terminates, the interrupted process continues with the system call. Preemption points can be placed at only “safe” locations in the kernel—only where kernel data structures are not being modified. Even with preemption points dispatch latency can be large, because only a few preemption points can be practically added to a kernel.

Another method for dealing with preemption is to make the entire kernel preemptible. So that correct operation is ensured, all kernel data structures must be protected through the use of various synchronization mechanisms that we discuss in Chapter 7. With this method, the kernel can always be preemptible, because any kernel data being updated are protected from modification by the high-priority process. This is the most effective (and complex) method in widespread use; it is used in Solaris 2.

But what happens if the higher-priority process needs to read or modify kernel data currently being accessed by another, lower-priority process? The high-priority process would be waiting for a lower-priority one to finish. This situation is known as **priority inversion**. In fact, a chain of processes could all be accessing resources that the high-priority process needs. This problem can be solved via the **priority-inheritance protocol**, in which all these processes (the ones accessing resources that the high-priority process needs) inherit the high priority until they are done with the resource in question. When they are finished, their priority reverts to its original value.

In Figure 6.8, we show the makeup of dispatch latency. The **conflict phase** of dispatch latency has two components:

1. Preemption of any process running in the kernel
2. Release by low-priority processes resources needed by the high-priority process

As an example, in Solaris 2, the dispatch latency with preemption disabled is over 100 milliseconds. However, the dispatch latency with preemption enabled is usually reduced to 2 milliseconds.

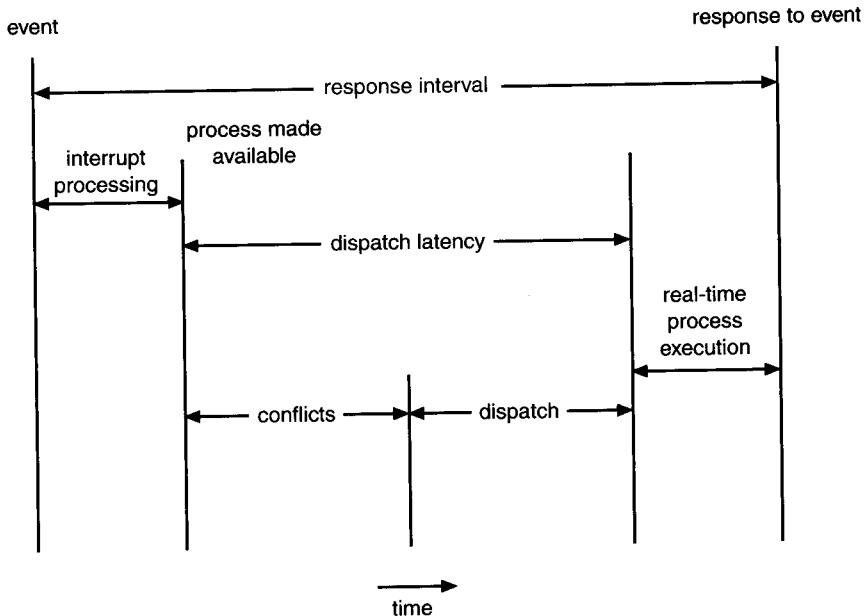


Figure 6.8 Dispatch latency.

6.6 ■ Algorithm Evaluation

How do we select a CPU-scheduling algorithm for a particular system? As we saw in Section 6.3, there are many scheduling algorithms, each with its own parameters. As a result, selecting an algorithm can be difficult.

The first problem is defining the criteria to be used in selecting an algorithm. As we saw in Section 6.2, criteria are often defined in terms of CPU utilization, response time, or throughput. To select an algorithm, we must first define the relative importance of these measures. Our criteria may include several measures, such as:

- Maximize CPU utilization under the constraint that the maximum response time is 1 second.
- Maximize throughput such that turnaround time is (on average) linearly proportional to total execution time.

Once the selection criteria have been defined, we want to evaluate the various algorithms under consideration. We describe the different evaluation methods in Sections 6.6.1 through 6.6.4.

6.6.1 Deterministic Modeling

One major class of evaluation methods is called analytic evaluation. **Analytic evaluation** uses the given algorithm and the system workload to produce a formula or number that evaluates the performance of the algorithm for that workload.

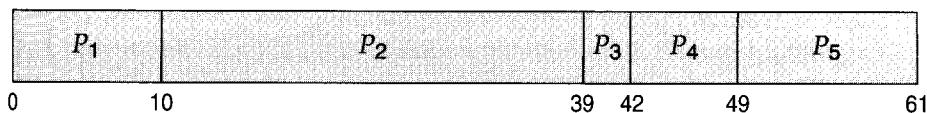
One type of analytic evaluation is **deterministic modeling**. This method takes a particular predetermined workload and defines the performance of each algorithm for that workload.

For example, assume that we have the workload shown. All five processes arrive at time 0, in the order given, with the length of the CPU-burst time given in milliseconds:

Process	Burst Time
P_1	10
P_2	29
P_3	3
P_4	7
P_5	12

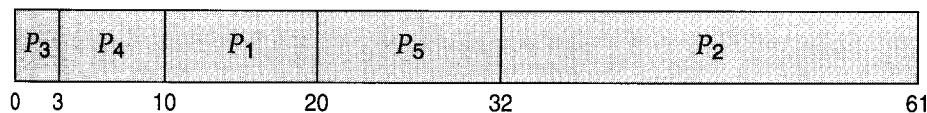
Consider the FCFS, SJF, and RR (quantum = 10 milliseconds) scheduling algorithms for this set of processes. Which algorithm would give the minimum average waiting time?

For the FCFS algorithm, we would execute the processes as



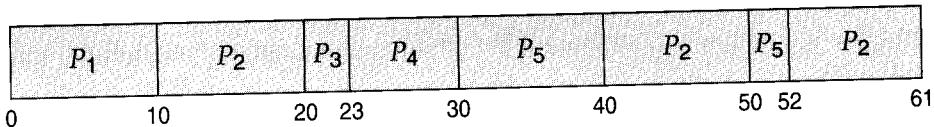
The waiting time is 0 milliseconds for process P_1 , 10 milliseconds for process P_2 , 39 milliseconds for process P_3 , 42 milliseconds for process P_4 , and 49 milliseconds for process P_5 . Thus, the average waiting time is $(0 + 10 + 39 + 42 + 49)/5 = 28$ milliseconds.

With nonpreemptive SJF scheduling, we execute the processes as



The waiting time is 10 milliseconds for process P_1 , 32 milliseconds for process P_2 , 0 milliseconds for process P_3 , 3 milliseconds for process P_4 , and 20 milliseconds for process P_5 . Thus, the average waiting time is $(10 + 32 + 0 + 3 + 20)/5 = 13$ milliseconds.

With the RR algorithm, we execute the processes as



The waiting time is 0 milliseconds for process P_1 , 32 milliseconds for process P_2 , 20 milliseconds for process P_3 , 23 milliseconds for process P_4 , and 40 milliseconds for process P_5 . Thus, the average waiting time is $(0 + 32 + 20 + 23 + 40)/5 = 23$ milliseconds.

We see that, *in this case*, the SJF policy results in less than one-half the average waiting time obtained with FCFS scheduling; the RR algorithm gives us an intermediate value.

Deterministic modeling is simple and fast. It gives exact numbers, allowing the algorithms to be compared. However, it requires exact numbers for input, and its answers apply to only those cases. The main uses of deterministic modeling are in describing scheduling algorithms and providing examples. In cases where we may be running the same programs over and over again and can measure the program's processing requirements exactly, we may be able to use deterministic modeling to select a scheduling algorithm. Over a set of examples, deterministic modeling may indicate trends that can then be analyzed and proved separately. For example, it can be shown that, for the environment described (all processes and their times available at time 0), the SJF policy will always result in the minimum waiting time.

In general, however, deterministic modeling is too specific, and requires too much exact knowledge, to be useful.

6.6.2 Queueing Models

The processes that are run on many systems vary from day to day, so there is no static set of processes (and times) to use for deterministic modeling. What can be determined, however, is the distribution of CPU and I/O bursts. These distributions may be measured and then approximated or simply estimated. The result is a mathematical formula describing the probability of a particular CPU burst. Commonly, this distribution is exponential and is described by its mean. Similarly, the distribution of times when processes arrive in the system—the arrival-time distribution—must be given.

The computer system is described as a network of servers. Each server has a queue of waiting processes. The CPU is a server with its ready queue, as is the I/O system with its device queues. Knowing arrival rates and service rates, we can compute utilization, average queue length, average wait time, and so on. This area of study is called **queueing-network analysis**.

As an example, let n be the average queue length (excluding the process being serviced), let W be the average waiting time in the queue, and let λ be the average arrival rate for new processes in the queue (such as three processes per second). Then, we expect that during the time W that a process waits, $\lambda \times W$ new processes will arrive in the queue. If the system is in a steady state, then the number of processes leaving the queue must be equal to the number of processes that arrive. Thus,

$$n = \lambda \times W.$$

This equation is known as **Little's formula**. Little's formula is particularly useful because it is valid for any scheduling algorithm and arrival distribution.

We can use Little's formula to compute one of the three variables, if we know the other two. For example, if we know that seven processes arrive every second (on average), and that there are normally 14 processes in the queue, then we can compute the average waiting time per process as 2 seconds.

Queueing analysis can be useful in comparing scheduling algorithms, but it also has limitations. At the moment, the classes of algorithms and distributions that can be handled are fairly limited. The mathematics of complicated algorithms or distributions can be difficult to work with. Thus, arrival and service distributions are often defined in unrealistic, but mathematically tractable, ways. It is also generally necessary to make a number of independent assumptions, that may not be accurate. Thus, so that they will be able to compute an answer, queueing models are often only an approximation of a real system. As a result, the accuracy of the computed results may be questionable.

6.6.3 Simulations

To get a more accurate evaluation of scheduling algorithms, we can use **simulations**. Simulations involve programming a model of the computer system. Software data structures represent the major components of the system. The simulator has a variable representing a clock; as this variable's value is increased, the simulator modifies the system state to reflect the activities of the devices, the processes, and the scheduler. As the simulation executes, statistics that indicate algorithm performance are gathered and printed.

The data to drive the simulation can be generated in several ways. The most common method uses a random-number generator, which is programmed to generate processes, CPU-burst times, arrivals, departures, and so on, according to probability distributions. The distributions may be defined mathematically (uniform, exponential, Poisson) or empirically. If the distribution is to be defined empirically, measurements of the actual system under study are taken. The results are used to define the actual distribution of events in the real system, and this distribution can then be used to drive the simulation.

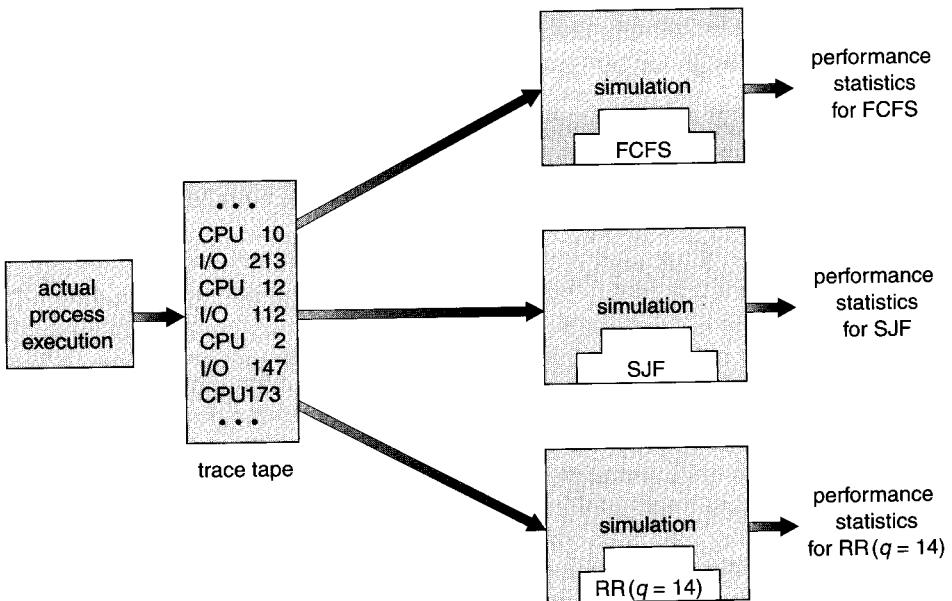


Figure 6.9 Evaluation of CPU schedulers by simulation.

A distribution-driven simulation may be inaccurate, however, due to relationships between successive events in the real system. The frequency distribution indicates only how many of each event occur; it does not indicate anything about the order of their occurrence. To correct this problem, we can use **trace tapes**. We create a trace tape by monitoring the real system, recording the sequence of actual events (Figure 6.9). This sequence is then used to drive the simulation. Trace tapes provide an excellent way to compare two algorithms on exactly the same set of real inputs. This method can produce accurate results for its inputs.

Simulations can be expensive, however, often requiring hours of computer time. A more detailed simulation provides more accurate results, but also requires more computer time. In addition, trace tapes can require large amounts of storage space. Finally, the design, coding, and debugging of the simulator can be a major task.

6.6.4 Implementation

Even a simulation is of limited accuracy. The only completely accurate way to evaluate a scheduling algorithm is to code it, put it in the operating system, and see how it works. This approach puts the actual algorithm in the real system for evaluation under real operating conditions.

The major difficulty is the cost of this approach. The expense is incurred not only in coding the algorithm and modifying the operating system to support it as well as its required data structures, but also in the reaction of the users to a constantly changing operating system. Most users are not interested in building a better operating system; they merely want to get their processes executed and to use their results. A constantly changing operating system does not help the users to get their work done. A form of this method is used commonly for new computer installations. For instance, a new web facility may have simulated user loads generated against it before it “goes live”, to determine any bottlenecks in the facility and to estimate how many users the system can support.

The other difficulty with any algorithm evaluation is that the environment in which the algorithm is used will change. The environment will change not only in the usual way, as new programs are written and the types of problems change, but also as a result of the performance of the scheduler. If short processes are given priority, then users may break larger processes into sets of smaller processes. If interactive processes are given priority over noninteractive processes, then users may switch to interactive use.

For example, in DEC TOPS-20, the system classified interactive and noninteractive processes automatically by looking at the amount of terminal I/O. If a process did not input or output to the terminal in a 1-minute interval, the process was classified as noninteractive and was moved to a lower-priority queue. This policy resulted in a situation where one programmer modified his programs to write an arbitrary character to the terminal at regular intervals of less than 1 minute. The system gave his programs a high priority, even though the terminal output was completely meaningless.

The most flexible scheduling algorithms can be altered by the system managers or by the users. During operating-system build time, boot time, or run time, the variables used by the schedulers can be changed to reflect the expected future use of the system. The need for flexible scheduling is another instance where the separation of mechanism from policy is useful. For instance, if paychecks need to be processed and printed immediately, but are normally done as a low-priority batch job, the batch queue could be given a higher priority temporarily. Unfortunately, few operating systems allow this type of tunable scheduling.

6.7 ■ Process Scheduling Models

In this section we will cover process scheduling in the Solaris 2, Windows 2000, and Linux operating systems. However, prior to looking at these different scheduling models, we first relate threads to process scheduling.

In Chapter 5, we introduced threads to the process model, thus allowing a single process to have multiple threads of control. Furthermore, we dis-

tinguished between *user-level* and *kernel-level* threads. User-level threads are managed by a thread library, and the kernel is unaware of them. To run on a CPU, user-level threads are ultimately mapped to an associated kernel-level thread, although this mapping may be indirect and use a lightweight process (LWP). One distinction between user-level and kernel-level threads lies in how they are scheduled. The thread library schedules user-level threads to run on an available LWP, a scheme known as **process local scheduling**, in which thread scheduling is done local to the application. Conversely, the kernel uses **system global scheduling** to decide which kernel thread to schedule. We do not cover in detail how different thread libraries locally schedule threads; thread scheduling is a software-library concern rather than an operating-system concern. We cover global scheduling because it is performed by the operating system.

6.7.1 An Example: Solaris 2

Solaris 2 uses priority-based process scheduling. It has four classes of scheduling, which are, in order of priority, real time, system, time sharing, and interactive. Each class includes different priorities and scheduling algorithms, although time sharing and interactive use the same scheduling policies. Solaris 2 scheduling is illustrated in Figure 6.10.

A process starts with one LWP and is able to create new LWPs as needed. Each LWP inherits the scheduling class and priority of the parent process. The default scheduling class for a process is time sharing. The scheduling policy for time sharing dynamically alters priorities and assigns time slices of different lengths using a multilevel feedback queue. By default, there is an inverse relationship between priorities and time slices: The higher the priority, the smaller the time slice, and the lower the priority, the larger the time slice. Interactive processes typically have a higher priority, CPU-bound processes a lower priority. This scheduling policy gives good response time for interactive processes and good throughput for CPU-bound processes. Solaris 2.4 introduced the interactive class to process scheduling. The interactive class uses the same scheduling policy as the time-sharing class, but it gives windowing applications a higher priority for better performance.

Solaris 2 uses the system class to run kernel processes, such as the scheduler and paging daemon. Once established, the priority of a system process does not change. The system class is reserved for kernel use (user processes running in kernel mode are not in the system class). The scheduling policy for the system class does not time-slice. Rather, a thread belonging to the system class runs until it either blocks or is preempted by a higher priority thread.

Threads in the real-time class are given the highest priority to run among all classes. This assignment allows a real-time process to have a guaranteed response from the system within a bounded period of time. A real-time process will run before a process in any other class. In general, few processes belong to the real-time class.

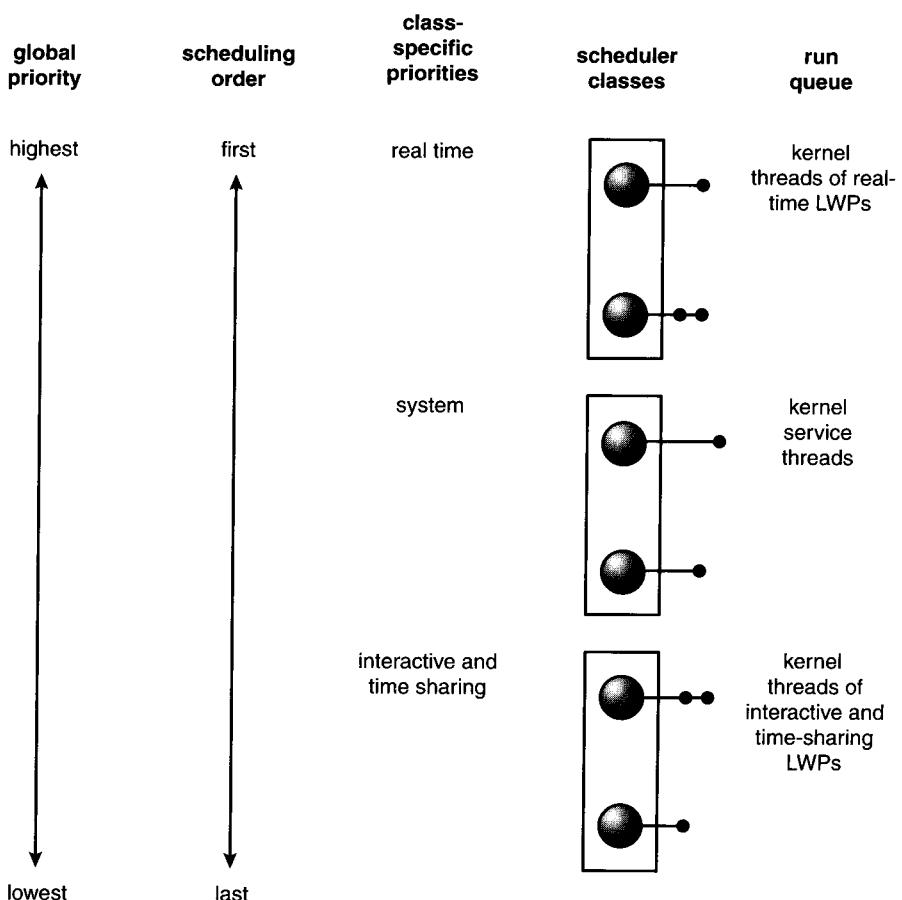


Figure 6.10 Solaris 2 scheduling.

Each scheduling class includes a set of priorities. However, the scheduler converts the class-specific priorities into global priorities, and selects to run the thread with the highest global priority. The selected thread runs on the CPU until one of the following occurs:

1. It blocks
2. It uses its time slice (if it is not a system thread)
3. It is preempted by a higher-priority thread

If multiple threads have the same priority, the scheduler uses a round-robin queue.

6.7.2 An Example: Windows 2000

Windows 2000 schedules threads using a priority-based, preemptive scheduling algorithm. The Windows 2000 scheduler ensures the highest-priority thread will always run. The portion of the Windows 2000 kernel that handles scheduling is called the *dispatcher*. A thread selected to run by the dispatcher will run until it is preempted by a higher-priority thread, until it terminates, until its time quantum ends, or until it calls a blocking system call, such as for I/O. If a higher-priority real-time thread becomes ready while a lower-priority thread is running, the lower-priority thread will be preempted. This preemption gives a real-time thread preferential access to the CPU when the thread needs such access. Windows 2000 is not a hard real-time operating system, however, because it does not guarantee that a real-time thread will start to execute within any particular time limit.

The dispatcher uses a 32-level priority scheme to determine the order of thread execution. Priorities are divided into two classes: the **variable class** contains threads having priorities from 1 to 15, and the **real-time class** contains threads with priorities ranging from 16 to 31. (There is also a thread running at priority 0 that is used for memory management.) The dispatcher uses a queue for each scheduling priority, and traverses the set of queues from highest to lowest until it finds a thread that is ready to run. If no ready thread is found, the dispatcher will execute a special thread called the **idle thread**.

There is a relationship between the numeric priorities of the Windows 2000 kernel and the Win32 API. The Win32 API identifies several priority classes that a process may belong to. These include:

- REALTIME_PRIORITY_CLASS
- HIGH_PRIORITY_CLASS
- ABOVE_NORMAL_PRIORITY_CLASS
- NORMAL_PRIORITY_CLASS
- BELOW_NORMAL_PRIORITY_CLASS
- IDLE_PRIORITY_CLASS

All priority classes except the REALTIME_PRIORITY.CLASS are variable class priorities, meaning that the priority of a thread belonging to one of these classes can change.

Within each of these priority classes is a relative priority. The values for relative priority include:

- TIME_CRITICAL
- HIGHEST

- ABOVE_NORMAL
- NORMAL
- BELOW_NORMAL
- LOWEST
- IDLE

The priority of each thread is based upon the priority class it belongs to and the relative priority within the class. This relationship is shown in Figure 6.11. The values of each priority class appear in the top row. The left column contains the values for the different relative priorities. For example, if the relative priority of a thread in the ABOVE_NORMAL_PRIORITY_CLASS is NORMAL, the numeric priority of that thread is 10.

Furthermore, each thread has a base priority representing a value in the priority range for the class the thread belongs to. By default, the base priority is the value of the NORMAL relative priority for that specific class. The base priorities for each priority class are:

- REALTIME_PRIORITY_CLASS—24.
- HIGH_PRIORITY_CLASS—13.
- ABOVE_NORMAL_PRIORITY_CLASS—10.
- NORMAL_PRIORITY_CLASS—8.
- BELOW_NORMAL_PRIORITY_CLASS—6.
- IDLE_PRIORITY_CLASS—4.

	real-time	high	above normal	normal	below normal	idle priority
time-critical	31	15	15	15	15	15
highest	26	15	12	10	8	6
above normal	25	14	11	9	7	5
normal	24	13	10	8	6	4
below normal	23	12	9	7	5	3
lowest	22	11	8	6	4	2
idle	16	1	1	1	1	1

Figure 6.11 Windows 2000 priorities.

Processes are typically members of the NORMAL_PRIORITY_CLASS unless the parent of the process was of the IDLE_PRIORITY_CLASS, or another class was specified when the process was created. The initial priority of a thread is typically the base priority of the process the thread belongs to.

When a thread's time quantum runs out, that thread is interrupted; if the thread is in the variable-priority class, its priority is lowered. The priority is never lowered below the base priority, however. Lowering the thread's priority tends to limit the CPU consumption of compute-bound threads. When a variable-priority thread is released from a wait operation, the dispatcher boosts the priority. The amount of the boost depends on what the thread was waiting for; for example, a thread that was waiting for keyboard I/O would get a large priority increase, whereas a thread waiting for a disk operation would get a moderate one. This strategy tends to give good response times to interactive threads that are using the mouse and windows, and enables I/O-bound threads to keep the I/O devices busy, while permitting compute-bound threads to use spare CPU cycles in the background. This strategy is used by several time-sharing operating systems, including UNIX. In addition, the current window with which the user is interacting also receives a priority boost to enhance its response time.

When a user is running an interactive program, the system needs to provide especially good performance for that process. For this reason, Windows 2000 has a special scheduling rule for processes in the NORMAL_PRIORITY_CLASS. Windows 2000 distinguishes between the *foreground process* that is currently selected on the screen, and the *background processes* that are not currently selected. When a process moves into the foreground, Windows 2000 increases the scheduling quantum by some factor—typically by 3. This increase gives the foreground process three times longer to run before a time-sharing preemption occurs.

6.7.3 An Example: Linux

Linux provides two separate process-scheduling algorithms. One is a time-sharing algorithm for fair preemptive scheduling among multiple processes; the other is designed for real-time tasks where absolute priorities are more important than fairness. In Section 6.5, we described a situation in which real-time systems must allow the kernel to be preempted to keep dispatch latency low. Linux allows only processes running in user mode to be preempted. A process may not be preempted while it is running in kernel mode, even if a real-time process with a higher priority is available to run.

Part of every process' identity is a scheduling class, that defines which of these algorithms to apply to the process. The scheduling classes used by Linux are defined in the POSIX standard's extensions for real-time computing (POSIX.4, now known as POSIX.1b).

The first scheduling class is for time-sharing processes. For conventional, time-shared processes, Linux uses a prioritized, **credit-based** algorithm. Each process possesses a certain number of scheduling credits; when a new task must be chosen to run, the process with the most credits is selected. Every time that a timer interrupt occurs, the currently running process loses one credit; when its credits reaches zero, it is suspended and another process is chosen.

If no runnable processes have any credits, then Linux performs a recreditting operation, adding credits to *every* process in the system (rather than to just the runnable ones), according to the following rule:

$$\text{credits} = \frac{\text{credits}}{2} + \text{priority}$$

This algorithm tends to mix two factors: the process' history and its priority. One-half of the credits that a process still holds since the previous recreditting operation will be retained after the algorithm has been applied, retaining some history of the process' recent behavior. Processes that are running all the time tend to exhaust their credits rapidly, but processes that spend much of their time suspended can accumulate credits over multiple recreditings and consequently end up with a higher credit count after a recredit. This crediting system automatically gives high priority to interactive or I/O-bound processes, for which a rapid response time is important.

The use of a process priority in calculating new credits allows the priority of a process to be fine-tuned. Background batch jobs can be given a low priority; they will automatically receive fewer credits than interactive users' jobs, and hence will receive a smaller percentage of the CPU time than will similar jobs with higher priorities. Linux uses this priority system to implement the standard UNIX *nice* process-priority mechanism. Linux's real-time scheduling is simpler still. Linux implements the two real-time scheduling classes required by POSIX.1b: first come, first served (FCFS), and round-robin (RR) (Sections 6.3.1 and 6.3.4, respectively). In both cases, each process has a priority in addition to its scheduling class. In time-sharing scheduling, however, processes of different priorities can still compete with one another to some extent; in real-time scheduling, the scheduler always runs the process with the highest priority. Among processes of equal priority, it runs the process that has been waiting longest. The only difference between FCFS and RR scheduling is that FCFS processes continue to run until they either exit or block, whereas a round-robin process will be preempted after a while and will be moved to the end of the scheduling queue, so round-robin processes of equal priority will automatically time share among themselves.

Note that Linux's real-time scheduling is soft—rather than hard—real time. The scheduler offers strict guarantees about the relative priorities of real-time processes, but the kernel does not offer any guarantees about how quickly a real-time process will be scheduled once that process becomes runnable. Remember that Linux kernel code can never be preempted by user-mode

code. If an interrupt arrives that wakes up a real-time process while the kernel is already executing a system call on behalf of another process, the real-time process will just have to wait until the currently running system call completes or blocks.

6.8 ■ Summary

CPU scheduling is the task of selecting a waiting process from the ready queue and allocating the CPU to it. The CPU is allocated to the selected process by the dispatcher.

First-come, first-served (FCFS) scheduling is the simplest scheduling algorithm, but it can cause short processes to wait for very long processes. Shortest-job-first (SJF) scheduling is provably optimal, providing the shortest average waiting time. Implementing SJF scheduling is difficult because predicting the length of the next CPU burst is difficult. The SJF algorithm is a special case of the general priority-scheduling algorithm, which simply allocates the CPU to the highest-priority process. Both priority and SJF scheduling may suffer from starvation. Aging is a technique to prevent starvation.

Round-robin (RR) scheduling is more appropriate for a time-shared (interactive) system. RR scheduling allocates the CPU to the first process in the ready queue for q time units, where q is the time quantum. After q time units, if the process has not relinquished the CPU, it is preempted and the process is put at the tail of the ready queue. The major problem is the selection of the time quantum. If the quantum is too large, RR scheduling degenerates to FCFS scheduling; if the quantum is too small, scheduling overhead in the form of context-switch time becomes excessive.

The FCFS algorithm is nonpreemptive; the RR algorithm is preemptive. The SJF and priority algorithms may be either preemptive or nonpreemptive.

Multilevel queue algorithms allow different algorithms to be used for various classes of processes. The most common is a foreground interactive queue, which uses RR scheduling, and a background batch queue, which uses FCFS scheduling. Multilevel feedback queues allow processes to move from one queue to another.

Because such a wide variety of scheduling algorithms are available, we need methods to select among them. Analytic methods use mathematical analysis to determine the performance of an algorithm. Simulation methods determine performance by imitating the scheduling algorithm on a “representative” sample of processes, and computing the resulting performance.

Operating systems supporting threads at the kernel level must schedule threads—not processes—for execution. This is the case with Solaris 2 and Windows 2000 where both systems schedule threads using preemptive, priority-based scheduling algorithms including support for real-time threads. The Linux process scheduler also uses a priority-based algorithm with real-time

support as well. The scheduling algorithms for these three operating systems typically favor interactive over batch and CPU-bound processes.

■ Exercises

- 6.1 A CPU-scheduling algorithm determines an order for the execution of its scheduled processes. Given n processes to be scheduled on one processor, how many different schedules are possible? Give a formula in terms of n .
- 6.2 Define the difference between preemptive and nonpreemptive scheduling. State why strict nonpreemptive scheduling is unlikely to be used in a computer center.
- 6.3 Consider the following set of processes, with the length of the CPU-burst time given in milliseconds:

Process	Burst Time	Priority
P_1	10	3
P_2	1	1
P_3	2	3
P_4	1	4
P_5	5	2

The processes are assumed to have arrived in the order P_1, P_2, P_3, P_4, P_5 , all at time 0.

- a. Draw four Gantt charts illustrating the execution of these processes using FCFS, SJF, a nonpreemptive priority (a smaller priority number implies a higher priority), and RR (quantum = 1) scheduling.
- b. What is the turnaround time of each process for each of the scheduling algorithms in part a?
- c. What is the waiting time of each process for each of the scheduling algorithms in part a?
- d. Which of the schedules in part a results in the minimal average waiting time (over all processes)?
- 6.4 Suppose that the following processes arrive for execution at the times indicated. Each process will run the listed amount of time. In answering the questions, use nonpreemptive scheduling and base all decisions on the information you have at the time the decision must be made.

Process	Arrival Time	Burst Time
P_1	0.0	8
P_2	0.4	4
P_3	1.0	1

- a. What is the average turnaround time for these processes with the FCFS scheduling algorithm?
- b. What is the average turnaround time for these processes with the SJF scheduling algorithm?
- c. The SJF algorithm is supposed to improve performance, but notice that we chose to run process P_1 at time 0 because we did not know that two shorter processes would arrive soon. Compute what the average turnaround time will be if the CPU is left idle for the first 1 unit and then SJF scheduling is used. Remember that processes P_1 and P_2 are waiting during this idle time, so their waiting time may increase. This algorithm could be known as future-knowledge scheduling.
- 6.5 Consider a variant of the RR scheduling algorithm where the entries in the ready queue are pointers to the PCBs.
- What would be the effect of putting two pointers to the same process in the ready queue?
 - What would be the major advantages and disadvantages of this scheme?
 - How would you modify the basic RR algorithm to achieve the same effect without the duplicate pointers?
- 6.6 What advantage is there in having different time-quantum sizes on different levels of a multilevel queueing system?
- 6.7 Consider the following preemptive priority-scheduling algorithm based on dynamically changing priorities. Larger priority numbers imply higher priority. When a process is waiting for the CPU (in the ready queue, but not running), its priority changes at a rate α ; when it is running, its priority changes at a rate β . All processes are given a priority of 0 when they enter the ready queue. The parameters α and β can be set to give many different scheduling algorithms.
- What is the algorithm that results from $\beta > \alpha > 0$?
 - What is the algorithm that results from $\alpha < \beta < 0$?
- 6.8 Many CPU-scheduling algorithms are parameterized. For example, the RR algorithm requires a parameter to indicate the time slice. Multilevel

feedback queues require parameters to define the number of queues, the scheduling algorithms for each queue, the criteria used to move processes between queues, and so on.

These algorithms are thus really sets of algorithms (for example, the set of RR algorithms for all time slices, and so on). One set of algorithms may include another (for example, the FCFS algorithm is the RR algorithm with an infinite time quantum). What (if any) relation holds between the following pairs of sets of algorithms?

- a. Priority and SJF
- b. Multilevel feedback queues and FCFS
- c. Priority and FCFS
- d. RR and SJF

- 6.9** Suppose that a scheduling algorithm (at the level of short-term CPU scheduling) favors those processes that have used the least processor time in the recent past. Why will this algorithm favor I/O-bound programs and yet not permanently starve CPU-bound programs?
- 6.10** Explain the differences in the degree to which the following scheduling algorithms discriminate in favor of short processes:
- a. FCFS
 - b. RR
 - c. Multilevel feedback queues

Bibliographical Notes

Lampson [1968] provided general discussions concerning scheduling. More formal treatments of scheduling theory were contained in Kleinrock [1975], Sauer and Chandy [1981], and Lazowska et al. [1984]. A unifying approach to scheduling was presented by Ruschizka and Fabry [1977]. Haldar and Subramanian [1991] discuss fairness in processor scheduling in time-sharing systems.

Feedback queues were originally implemented on the CTSS system described in Corbato et al. [1962]. This feedback queueing system was analyzed by Schrage [1967]; variations on multilevel feedback queues were studied by Coffman and Kleinrock [1968]. Additional studies were presented by Coffman and Denning [1973] and Svobodova [1976]. A data structure for manipulating priority queues was presented by Vuillemin [1978].

Anderson et al. [1989] discussed thread scheduling. Discussions concerning multiprocessor scheduling were presented by Jones and Schwarz [1980], Tucker

and Gupta [1989], Zahorjan and McCann [1990], Feitelson and Rudolph [1990], and Leutenegger and Vernon [1990].

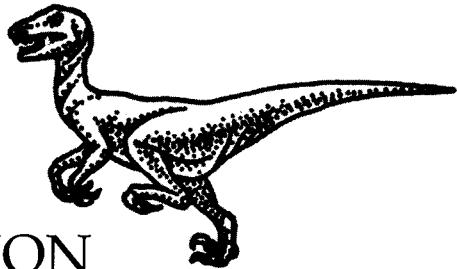
Discussions concerning scheduling in real-time systems were offered by Liu and Layland [1973], Abbot [1984], Jensen et al. [1985], Hong et al. [1989], and Khanna et al. [1992]. A special issue on real-time operating systems was edited by Zhao [1989]. Eykholt et al. [1992] described the real-time component of Solaris 2.

Fair-share schedulers were covered by Henry [1984], Woodside [1986], and Kay and Lauder [1988].

Details concerning scheduling in Solaris 2, Windows 2000, and Linux can be found in Mauro and McDougall [2001], Solomon and Russinovich [2000], and Bovet and Cesati [2001] respectively.

Chapter 7

PROCESS SYNCHRONIZATION



A cooperating process is one that can affect or be affected by other processes executing in the system. Cooperating processes may either directly share a logical address space (that is, both code and data), or be allowed to share data only through files. The former case is achieved through the use of lightweight processes or threads, which we discussed in Section 5. Concurrent access to shared data may result in data inconsistency. In this chapter, we discuss various mechanisms to ensure the orderly execution of cooperating processes that share a logical address space, so that data consistency is maintained.

7.1 ■ Background

In Chapter 4, we developed a model of a system consisting of a number of **cooperating sequential processes**, all running asynchronously and possibly sharing data. We illustrated this model with the bounded-buffer scheme, which is representative of operating systems.

Let us return to the shared-memory solution to the bounded-buffer problem that we presented in Section 4.4. As we pointed out, our solution allows at most `BUFFER_SIZE - 1` items in the buffer at the same time. Suppose that we want to modify the algorithm to remedy this deficiency. One possibility is to add an integer variable `counter`, initialized to 0. `counter` is incremented every time we add a new item to the buffer, and is decremented every time we remove one item from the buffer. The code for the producer process can be modified as follows:

```

while(1) {
    /* produce an item in nextProduced */
    while (counter == BUFFER_SIZE)
        ; /* do nothing */
    buffer[in] = nextProduced;
    in = (in + 1) % BUFFER_SIZE;
    counter++;
}

```

The code for the consumer process can be modified as follows:

```

while(1) {
    while (counter == 0)
        ; /* do nothing */
    nextConsumed = buffer[out];
    out = (out + 1) % BUFFER_SIZE;
    counter--;
    /* consume the item in nextConsumed */
}

```

Although both the producer and consumer routines are correct separately, they may not function correctly when executed concurrently. As an illustration, suppose that the value of the variable `counter` is currently 5, and that the producer and consumer processes execute the statements “`counter++`” and “`counter--`” concurrently. Following the execution of these two statements, the value of the variable `counter` may be 4, 5, or 6! The only correct result is `counter == 5`, which is generated correctly if the producer and consumer execute separately.

We can show that the value of `counter` may be incorrect as follows. Note that the statement “`counter++`” may be implemented in machine language (on a typical machine) as

```

register1 = counter
register1 = register1 + 1
counter = register1

```

where `register1` is a local CPU register. Similarly, the statement “`counter--`” is implemented as follows:

```

register2 = counter
register2 = register2 - 1
counter = register2

```

where again `register2` is a local CPU register. Even though `register1` and `register2` may be the same physical register (an accumulator, say), remember that the

contents of this register will be saved and restored by the interrupt handler (Section 2.1).

The concurrent execution of “`counter++`” and “`counter--`” is equivalent to a sequential execution where the lower-level statements presented previously are interleaved in some arbitrary order (but the order within each high-level statement is preserved). One such interleaving is

T_0 :	<i>producer</i>	execute	<code>register₁ = counter</code>	{ $register_1 = 5$ }
T_1 :	<i>producer</i>	execute	<code>register₁ = register₁ + 1</code>	{ $register_1 = 6$ }
T_2 :	<i>consumer</i>	execute	<code>register₂ = counter</code>	{ $register_2 = 5$ }
T_3 :	<i>consumer</i>	execute	<code>register₂ = register₂ - 1</code>	{ $register_2 = 4$ }
T_4 :	<i>producer</i>	execute	<code>counter = register₁</code>	{ $counter = 6$ }
T_5 :	<i>consumer</i>	execute	<code>counter = register₂</code>	{ $counter = 4$ }

Notice that we have arrived at the incorrect state “`counter == 4`” recording that there are four full buffers, when, in fact, there are five full buffers. If we reversed the order of the statements at T_4 and T_5 , we would arrive at the incorrect state “`counter == 6`”.

We would arrive at this incorrect state because we allowed both processes to manipulate the variable `counter` concurrently. A situation like this, where several processes access and manipulate the same data concurrently and the outcome of the execution depends on the particular order in which the access takes place, is called a **race condition**. To guard against the race condition above, we need to ensure that only one process at a time can be manipulating the variable `counter`. To make such a guarantee, we require some form of synchronization of the processes. Such situations occur frequently in operating systems as different parts of the system manipulate resources and we want the changes not to interfere with one another. A major portion of this chapter is concerned with the issue of **process synchronization** and **coordination**.

7.2 ■ The Critical-Section Problem

Consider a system consisting of n processes $\{P_0, P_1, \dots, P_{n-1}\}$. Each process has a segment of code, called a **critical section**, in which the process may be changing common variables, updating a table, writing a file, and so on. The important feature of the system is that, when one process is executing in its critical section, no other process is to be allowed to execute in its critical section. Thus, the execution of critical sections by the processes is *mutually exclusive* in time. The *critical-section* problem is to design a protocol that the processes can use to cooperate. Each process must request permission to enter its critical section. The section of code implementing this request is the **entry section**. The critical section may be followed by an **exit section**. The remaining code is the **remainder section**.

```

do {

    entry section

    critical section

    exit section

    remainder section

} while (1);

```

Figure 7.1 General structure of a typical process P_i .

A solution to the critical-section problem must satisfy the following three requirements:

1. **Mutual Exclusion:** If process P_i is executing in its critical section, then no other processes can be executing in their critical sections.
2. **Progress:** If no process is executing in its critical section and some processes wish to enter their critical sections, then only those processes that are not executing in their remainder section can participate in the decision on which will enter its critical section next, and this selection cannot be postponed indefinitely.
3. **Bounded Waiting:** There exists a bound on the number of times that other processes are allowed to enter their critical sections after a process has made a request to enter its critical section and before that request is granted.

We assume that each process is executing at a nonzero speed. However, we can make no assumption concerning the **relative speed** of the n processes.

In Sections 7.2.1 and 7.2.2, we work up to solutions to the critical-section problem that satisfy these three requirements. The solutions do not rely on any assumptions concerning the hardware instructions or the number of processors that the hardware supports. We do, however, assume that the basic machine-language instructions (the primitive instructions such as **load**, **store**, and **test**) are executed atomically. That is, if two such instructions are executed concurrently, the result is equivalent to their sequential execution in some unknown order. Thus, if a **load** and a **store** are executed concurrently, the **load** will get either the old value or the new value, but not some combination of the two.

When presenting an algorithm, we define only the variables used for synchronization purposes, and describe only a typical process P_i whose general structure is shown in Figure 7.1. The entry section and exit section are enclosed in boxes to highlight these important segments of code.

```

do {
    while (turn != i);
        critical section
        turn = j;
    remainder section
} while (1);

```

Figure 7.2 The structure of process P_i in algorithm 1.

7.2.1 Two-Process Solutions

In this section, we restrict our attention to algorithms that are applicable to only two processes at a time. The processes are numbered P_0 and P_1 . For convenience, when presenting P_i , we use P_j to denote the other process; that is, $j == 1 - i$.

7.2.1.1 Algorithm 1

Our first approach is to let the processes share a common integer variable `turn` initialized to 0 (or 1). If `turn == i`, then process P_i is allowed to execute in its critical section. The structure of process P_i is shown in Figure 7.2.

This solution ensures that only one process at a time can be in its critical section. However, it does not satisfy the progress requirement, since it requires strict alternation of processes in the execution of the critical section. For example, if `turn == 0` and P_1 is ready to enter its critical section, P_1 cannot do so, even though P_0 may be in its remainder section.

7.2.1.2 Algorithm 2

The problem with algorithm 1 is that it does not retain sufficient information about the state of each process; it remembers only which process is allowed to enter its critical section. To remedy this problem, we can replace the variable `turn` with the following array:

```
boolean flag[2];
```

The elements of the array are initialized to `false`. If `flag[i]` is `true`, this value indicates that P_i is *ready* to enter the critical section. The structure of process P_i is shown in Figure 7.3.

In this algorithm, process P_i first sets `flag[i]` to be `true`, signaling that it is ready to enter its critical section. Then, P_i checks to verify that process P_j is

```

do {
    flag[i] = true;
    while (flag[j]);
        critical section
    flag[i] = false;
        remainder section
} while (1);

```

Figure 7.3 The structure of process P_i in algorithm 2.

not also ready to enter its critical section. If P_j were ready, then P_i would wait until P_j had indicated that it no longer needed to be in the critical section (that is, until `flag[j]` was `false`). At this point, P_i would enter the critical section. On exiting the critical section, P_i would set `flag[i]` to be `false`, allowing the other process (if it is waiting) to enter its critical section.

In this solution, the mutual-exclusion requirement is satisfied. Unfortunately, the progress requirement is not met. To illustrate this problem, we consider the following execution sequence:

T_0 : P_0 sets `flag[0] = true`
 T_1 : P_1 sets `flag[1] = true`

Now P_0 and P_1 are looping forever in their respective `while` statements.

This algorithm is crucially dependent on the exact timing of the two processes. The sequence could have been derived in an environment where there are several processors executing concurrently, or where an interrupt (such as a timer interrupt) occurs immediately after step T_0 is executed, and the CPU is switched from one process to another.

Note that switching the order of the instructions for setting `flag[i]`, and testing the value of a `flag[j]`, will not solve our problem. Rather, we will have a situation where it is possible for both processes to be in the critical section at the same time, violating the mutual-exclusion requirement.

7.2.1.3 Algorithm 3

By combining the key ideas of algorithm 1 and algorithm 2, we obtain a correct solution to the critical-section problem, where all three requirements are met. The processes share two variables:

```

boolean flag[2];
int turn;

```

Initially $\text{flag}[0] = \text{flag}[1] = \text{false}$, and the value of turn is immaterial (but is either 0 or 1). The structure of process P_i is shown in Figure 7.4.

To enter the critical section, process P_i first sets $\text{flag}[i]$ to be true and then sets turn to the value j , thereby asserting that if the other process wishes to enter the critical section it can do so. If both processes try to enter at the same time, turn will be set to both i and j at roughly the same time. Only one of these assignments will last; the other will occur, but will be overwritten immediately. The eventual value of turn decides which of the two processes is allowed to enter its critical section first.

We now prove that this solution is correct. We need to show that:

1. Mutual exclusion is preserved,
2. The progress requirement is satisfied,
3. The bounded-waiting requirement is met.

To prove property 1, we note that each P_i enters its critical section only if either $\text{flag}[j] == \text{false}$ or $\text{turn} == i$. Also note that, if both processes can be executing in their critical sections at the same time, then $\text{flag}[0] == \text{flag}[1] == \text{true}$. These two observations imply that P_0 and P_1 could not have successfully executed their while statements at about the same time, since the value of turn can be either 0 or 1, but cannot be both. Hence, one of the processes—say P_j —must have successfully executed the while statement, whereas P_i had to execute at least one additional statement (“ $\text{turn} == j$ ”). However, since, at that time, $\text{flag}[j] == \text{true}$, and $\text{turn} == j$, and this condition will persist as long as P_j is in its critical section, the result follows: Mutual exclusion is preserved.

```

do {
    flag[i] = true;
    turn = j;
    while (flag[j] && turn == j);

    critical section

    flag[i] = false;

    remainder section
} while (1);

```

Figure 7.4 The structure of process P_i in algorithm 3.

To prove properties 2 and 3, we note that a process P_i can be prevented from entering the critical section only if it is stuck in the while loop with the condition $\text{flag}[j] == \text{true}$ and $\text{turn} == j$; this loop is the only one. If P_j is not ready to enter the critical section, then $\text{flag}[j] == \text{false}$ and P_i can enter its critical section. If P_j has set $\text{flag}[j]$ to `true` and is also executing in its `while` statement, then either $\text{turn} == i$ or $\text{turn} == j$. If $\text{turn} == i$, then P_i will enter the critical section. If $\text{turn} == j$, then P_j will enter the critical section. However, once P_j exits its critical section, it will reset $\text{flag}[j]$ to `false`, allowing P_i to enter its critical section. If P_j resets $\text{flag}[j]$ to `true`, it must also set turn to i . Thus, since P_i does not change the value of the variable turn while executing the `while` statement, P_i will enter the critical section (progress) after at most one entry by P_j (bounded waiting).

7.2.2 Multiple-Process Solutions

We have seen that algorithm 3 solves the critical-section problem for two processes. Now let us develop an algorithm for solving the critical-section problem for n processes. This algorithm is known as the *bakery algorithm*, and it is based on a scheduling algorithm commonly used in bakeries, ice-cream stores, deli counters, motor-vehicle registries, and other locations where order must be made out of chaos. This algorithm was developed for a distributed environment, but at this point we are concerned with only those aspects of the algorithm that pertain to a centralized environment.

On entering the store, each customer receives a number. The customer with the lowest number is served next. Unfortunately, the bakery algorithm cannot guarantee that two processes (customers) do not receive the same number. In the case of a tie, the process with the lowest name is served first. That is, if P_i and P_j receive the same number and if $i < j$, then P_i is served first. Since process names are unique and totally ordered, our algorithm is completely deterministic.

The common data structures are

```
boolean choosing[n];
int number[n];
```

Initially, these data structures are initialized to `false` and 0, respectively. For convenience, we define the following notation:

- $(a,b) < (c,d)$ if $a < c$ or if $a == c$ and $b < d$.
- $\max(a_0, \dots, a_{n-1})$ is a number, k , such that $k \geq a_i$ for $i = 0, \dots, n - 1$.

The structure of process P_i , used in the bakery algorithm, is shown in Figure 7.5.

To prove that the bakery algorithm is correct, we need first to show that, if P_i is in its critical section and P_k ($k != i$) has already chosen its `number` $k != 0$,

```

do {
    choosing[i] = true;
    number[i] = max(number[0], number[1], ..., number[n-1]) + 1;
    choosing[i] = false;
    for (j=0; j < n; j++) {
        while (choosing[j]);
        while ((number[j] != 0) && (number[j,j] < number[i,i]));
    }
}

critical section

    number[i] = 0;

remainder section

} while (1);

```

Figure 7.5 The structure of process P_i in the bakery algorithm.

then $(\text{number}[i], i) < (\text{number}[k], k)$. The proof of this algorithm is left to you in Exercise 7.3.

Given this result, it is now simple to show that mutual exclusion is observed. Indeed, consider P_i in its critical section and P_k trying to enter the P_k critical section. When process P_k executes the second while statement for $j == i$, it finds that

- $\text{number}[i] \neq 0$
- $(\text{number}[i], i) < (\text{number}[k], k)$.

Thus, it continues looping in the while statement until P_i leaves the P_i critical section.

If we wish to show that the progress and bounded-waiting requirements are preserved, and that the algorithm ensures fairness, it is sufficient to observe that the processes enter their critical section on a first-come, first-served basis.

7.3 ■ Synchronization Hardware

As with other aspects of software, hardware features can make the programming task easier and improve system efficiency. In this section, we present some simple hardware instructions that are available on many systems, and show how they can be used effectively in solving the critical-section problem.

```

boolean TestAndSet(boolean &target) {
    boolean rv = target;
    target = true;
    return rv;
}

```

Figure 7.6 The definition of the `TestAndSet` instruction.

The critical-section problem could be solved simply in a uniprocessor environment if we could forbid interrupts to occur while a shared variable is being modified. In this manner, we could be sure that the current sequence of instructions would be allowed to execute in order without preemption. No other instructions would be run, so no unexpected modifications could be made to the shared variable.

Unfortunately, this solution is not feasible in a multiprocessor environment. Disabling interrupts on a multiprocessor can be time-consuming, as the message is passed to all the processors. This message passing delays entry into each critical section, and system efficiency decreases. Also, consider the effect on a system's clock, if the clock is kept updated by interrupts.

Many machines therefore provide special hardware instructions that allow us either to test and modify the content of a word, or to swap the contents of two words, **atomically**—that is, as one uninterruptible unit. We can use these special instructions to solve the critical-section problem in a relatively simple manner. Rather than discussing one specific instruction for one specific machine, let us abstract the main concepts behind these types of instructions.

The `TestAndSet` instruction can be defined as shown in Figure 7.6. The important characteristic is that this instruction is executed atomically. Thus, if two `TestAndSet` instructions are executed simultaneously (each on a different CPU), they will be executed sequentially in some arbitrary order.

```

do {
    while (TestAndSet(lock));
    critical section
    lock = false;
    remainder section
} while (1);

```

Figure 7.7 Mutual-exclusion implementation with `TestAndSet`.

```
void Swap(boolean &a, boolean &b) {
    boolean temp = a;
    a = b;
    b = temp;
}
```

Figure 7.8 The definition of the Swap instruction.

If the machine supports the `TestAndSet` instruction, then we can implement mutual exclusion by declaring a Boolean variable `lock`, initialized to `false`. The structure of process P_i is shown in Figure 7.7.

The `Swap` instruction, defined as shown in Figure 7.8, operates on the contents of two words; like the `TestAndSet` instruction, it is executed atomically.

If the machine supports the `Swap` instruction, then mutual exclusion can be provided as follows. A global Boolean variable `lock` is declared and is initialized to `false`. In addition, each process also has a local Boolean variable `key`. The structure of process P_i is shown in Figure 7.9.

These algorithms do not satisfy the bounded-waiting requirement. We present an algorithm that uses the `TestAndSet` instruction in Figure 7.10. This algorithm satisfies all the critical-section requirements. The common data structures are

```
boolean waiting[n];
boolean lock;
```

These data structures are initialized to `false`. To prove that the mutual-exclusion requirement is met, we note that process P_i can enter its critical section

```
do {
    key = true;
    while (key == true)
        Swap(lock, key);

    critical section

    lock = false;

    remainder section

} while (1);
```

Figure 7.9 Mutual-exclusion implementation with the `Swap` instruction.

```

do {

    waiting[i] = true;
    key = true;
    while (waiting[i] && key)
        key = TestAndSet(lock);
    waiting[i] = false;

    critical section

    j = (i+1) % n;
    while ((j != i) && !waiting[j])
        j = (j+1) % n;
    if (j == i)
        lock = false;
    else
        waiting[j] = false;

    remainder section

} while (1);

```

Figure 7.10 Bounded-waiting mutual exclusion with `TestAndSet`.

only if either `waiting[i] == false` or `key == false`. The value of `key` can become `false` only if the `TestAndSet` is executed. The first process to execute the `TestAndSet` will find `key == false`; all others must wait. The variable `waiting[i]` can become `false` only if another process leaves its critical section; only one `waiting[i]` is set to `false`, maintaining the mutual-exclusion requirement.

To prove the progress requirement is met, we note that the arguments presented for mutual exclusion also apply here, since a process exiting the critical section either sets `lock` to `false`, or sets `waiting[j]` to `false`. Both allow a process that is waiting to enter its critical section to proceed.

To prove the bounded-waiting requirement is met, we note that, when a process leaves its critical section, it scans the array `waiting` in the cyclic ordering $(i + 1, i + 2, \dots, n - 1, 0, \dots, i - 1)$. It designates the first process in this ordering that is in the entry section (`waiting[j] == true`) as the next one to enter the critical section. Any process waiting to enter its critical section will thus do so within $n - 1$ turns. Unfortunately for hardware designers, implementing atomic `TestAndSet` instructions on multiprocessors is not a trivial task. Such implementations are discussed in books on computer architecture.

7.4 ■ Semaphores

The solutions to the critical-section problem presented in Section 7.3 are not easy to generalize to more complex problems. To overcome this difficulty, we can use a synchronization tool called a **semaphore**. A semaphore S is an integer variable that, apart from initialization, is accessed only through two standard atomic operations: **wait** and **signal**. These operations were originally termed P (for *wait*; from the Dutch *proberen*, to test) and V (for *signal*; from *verhogen*, to increment). The classical definition of **wait** in pseudocode is

```
wait(S) {
    while (S ≤ 0)
        ; // no-op
    S--;
}
```

The classical definitions of **signal** in pseudocode is

```
signal(S) {
    S++;
}
```

Modifications to the integer value of the semaphore in the **wait** and **signal** operations must be executed indivisibly. That is, when one process modifies the semaphore value, no other process can simultaneously modify that same semaphore value. In addition, in the case of the **wait(S)**, the testing of the integer value of S ($S \leq 0$), and its possible modification ($S--$), must also be executed without interruption. We shall see how these operations can be implemented in Section 7.4.2; first, let us see how semaphores can be used.

7.4.1 Usage

We can use semaphores to deal with the n -process critical-section problem. The n processes share a semaphore, **mutex** (standing for *mutual exclusion*), initialized to 1. Each process P_i is organized as shown in Figure 7.11.

We can also use semaphores to solve various synchronization problems. For example, consider two concurrently running processes: P_1 with a statement S_1 and P_2 with a statement S_2 . Suppose that we require that S_2 be executed only after S_1 has completed. We can implement this scheme readily by letting P_1 and P_2 share a common semaphore **synch**, initialized to 0, and by inserting the statements

```
 $S_1;$ 
signal(synch);
```

```

do {
    wait(mutex);
    critical section
    signal(mutex);
    remainder section
} while(1);

```

Figure 7.11 Mutual-exclusion implementation with semaphores.

in process P_1 , and the statements

```

wait(synch);
S2;

```

in process P_2 . Because `synch` is initialized to 0, P_2 will execute S_2 only after P_1 has invoked `signal(synch)`, which is after S_1 .

7.4.2 Implementation

The main disadvantage of the mutual-exclusion solutions of Section 7.2, and of the semaphore definition given here, is that they all require **busy waiting**. While a process is in its critical section, any other process that tries to enter its critical section must loop continuously in the entry code. This continual looping is clearly a problem in a real multiprogramming system, where a single CPU is shared among many processes. Busy waiting wastes CPU cycles that some other process might be able to use productively. This type of semaphore is also called a **spinlock** (because the process “spins” while waiting for the lock). Spinlocks are useful in multiprocessor systems. The advantage of a spinlock is that no context switch is required when a process must wait on a lock, and a context switch may take considerable time. Thus, when locks are expected to be held for short times, spinlocks are useful.

To overcome the need for busy waiting, we can modify the definition of the `wait` and `signal` semaphore operations. When a process executes the `wait` operation and finds that the semaphore value is not positive, it must wait. However, rather than busy waiting, the process can *block* itself. The `block` operation places a process into a waiting queue associated with the semaphore, and the state of the process is switched to the waiting state. Then, control is transferred to the CPU scheduler, which selects another process to execute.

A process that is blocked, waiting on a semaphore S , should be restarted when some other process executes a `signal` operation. The process is restarted

by a `wakeup` operation, which changes the process from the waiting state to the ready state. The process is then placed in the ready queue. (The CPU may or may not be switched from the running process to the newly ready process, depending on the CPU-scheduling algorithm.)

To implement semaphores under this definition, we define a semaphore as a “C” struct:

```
typedef struct {
    int value;
    struct process *L;
} semaphore;
```

Each semaphore has an integer value and a list of processes. When a process must wait on a semaphore, it is added to the list of processes. A `signal` operation removes one process from the list of waiting processes and awakens that process.

The `wait` semaphore operation can now be defined as

```
void wait(semaphore S) {
    S.value--;
    if (S.value < 0) {
        add this process to S.L;
        block();
    }
}
```

The `signal` semaphore operation can now be defined as

```
void signal(semaphore S) {
    S.value++;
    if (S.value <= 0) {
        remove a process P from S.L;
        wakeup(P);
    }
}
```

The `block` operation suspends the process that invokes it. The `wakeup(P)` operation resumes the execution of a blocked process *P*. These two operations are provided by the operating system as basic system calls.

Note that, although under the classical definition of semaphores with busy waiting the semaphore value is never negative, this implementation may have negative semaphore values. If the semaphore value is negative, its magnitude is the number of processes waiting on that semaphore. This fact is a result of

the switching of the order of the decrement and the test in the implementation of the `wait` operation. The list of waiting processes can be easily implemented by a link field in each process control block (PCB). Each semaphore contains an integer value and a pointer to a list of PCBs. One way to add and remove processes from the list, that ensures bounded waiting would be to use a FIFO queue, where the semaphore contains both head and tail pointers to the queue. In general, however, the list may use *any* queueing strategy. Correct usage of semaphores does not depend on a particular queueing strategy for the semaphore lists.

The critical aspect of semaphores is that they are executed atomically. We must guarantee that no two processes can execute `wait` and `signal` operations on the same semaphore at the same time. This situation is a critical-section problem, and can be solved in either of two ways.

In a uniprocessor environment (that is, where only one CPU exists), we can simply inhibit interrupts during the time the `wait` and `signal` operations are executing. This scheme works in a uniprocessor environment because, once interrupts are inhibited, instructions from different processes cannot be interleaved. Only the currently running process executes, until interrupts are reenabled and the scheduler can regain control.

In a multiprocessor environment, inhibiting interrupts does not work. Instructions from different processes (running on different processors) may be interleaved in some arbitrary way. If the hardware does not provide any special instructions, we can employ any of the correct software solutions for the critical-section problem (Section 7.2), where the critical sections consist of the `wait` and `signal` procedures.

It is important to admit that we have not completely eliminated busy waiting with this definition of the `wait` and `signal` operations. Rather, we have removed busy waiting from the entry to the critical sections of application programs. Furthermore, we have limited busy waiting to only the critical sections of the `wait` and `signal` operations, and these sections are short (if properly coded, they should be no more than about 10 instructions). Thus, the critical section is almost never occupied, and busy waiting occurs rarely, and then for only a short time. An entirely different situation exists with application programs whose critical sections may be long (minutes or even hours) or may be almost always occupied. In this case, busy waiting is extremely inefficient.

7.4.3 Deadlocks and Starvation

The implementation of a semaphore with a waiting queue may result in a situation where two or more processes are waiting indefinitely for an event that can be caused only by one of the waiting processes. The event in question is the execution of a `signal` operation. When such a state is reached, these processes are said to be **deadlocked**.

To illustrate this, we consider a system consisting of two processes, P_0 and P_1 , each accessing two semaphores, S and Q, set to the value 1:

P_0	P_1
<code>wait(S);</code>	<code>wait(Q);</code>
<code>wait(Q);</code>	<code>wait(S);</code>
.	.
.	.
.	.
<code>signal(S);</code>	<code>signal(Q);</code>
<code>signal(Q);</code>	<code>signal(S);</code>

Suppose that P_0 executes `wait(S)`, and then P_1 executes `wait(Q)`. When P_0 executes `wait(Q)`, it must wait until P_1 executes `signal(Q)`. Similarly, when P_1 executes `wait(S)`, it must wait until P_0 executes `signal(S)`. Since these `signal` operations cannot be executed, P_0 and P_1 are deadlocked.

We say that a set of processes is in a deadlock state when every process in the set is waiting for an event that can be caused only by another process in the set. The events with which we are mainly concerned here are *resource acquisition and release*. However, other types of events may result in deadlocks, as we shall show in Chapter 8. In that chapter, we shall describe various mechanisms for dealing with the deadlock problem.

Another problem related to deadlocks is **indefinite blocking** or **starvation**, a situation where processes wait indefinitely within the semaphore. Indefinite blocking may occur if we add and remove processes from the list associated with a semaphore in LIFO order.

7.4.4 Binary Semaphores

The semaphore construct described in the previous sections is commonly known as a **counting semaphore**, since its integer value can range over an unrestricted domain. A **binary semaphore** is a semaphore with an integer value that can range only between 0 and 1. A binary semaphore can be simpler to implement than a counting semaphore, depending on the underlying hardware architecture. We will now show how a counting semaphore can be implemented using binary semaphores.

Let S be a counting semaphore. To implement it in terms of binary semaphores we need the following data structures:

```
binary-semaphore S1, S2;
int C;
```

Initially $S1 = 1$, $S2 = 0$, and the value of integer C is set to the initial value of the counting semaphore S.

The `wait` operation on the counting semaphore `S` can be implemented as follows:

```
wait(S1);
C--;
if (C < 0) {
    signal(S1);
    wait(S2);
}
signal(S1);
```

The `signal` operation on the counting semaphore `S` can be implemented as follows:

```
wait(S1);
C++;
if (C <= 0)
    signal(S2);
else
    signal(S1);
```

7.5 ■ Classic Problems of Synchronization

In this section, we present a number of different synchronization problems as examples for a large class of concurrency-control problems. These problems are used for testing nearly every newly proposed synchronization scheme. Semaphores are used for synchronization in our solutions.

7.5.1 The Bounded-Buffer Problem

The *bounded-buffer* problem was introduced in Section 7.1; it is commonly used to illustrate the power of synchronization primitives. We present here a general structure of this scheme, without committing ourselves to any particular implementation. We assume that the pool consists of n buffers, each capable of holding one item. The `mutex` semaphore provides mutual exclusion for accesses to the buffer pool and is initialized to the value 1. The `empty` and `full` semaphores count the number of empty and full buffers, respectively. The semaphore `empty` is initialized to the value n ; the semaphore `full` is initialized to the value 0.

The code for the producer process is shown in Figure 7.12; the code for the consumer process is shown in Figure 7.13. Note the symmetry between the producer and the consumer. We can interpret this code as the producer producing full buffers for the consumer, or as the consumer producing empty buffers for the producer.

```

do {
    ...
    produce an item in nextp
    ...
    wait(empty);
    wait(mutex);
    ...
    add nextp to buffer
    ...
    signal(mutex);
    signal(full);
} while(1);

```

Figure 7.12 The structure of the producer process.

7.5.2 The Readers–Writers Problem

A data object (such as a file or record) is to be shared among several concurrent processes. Some of these processes may want only to read the content of the shared object, whereas others may want to update (that is, to read and write) the shared object. We distinguish between these two types of processes by referring to those processes that are interested in only reading as **readers**, and to the rest as **writers**. Obviously, if two readers access the shared data object simultaneously, no adverse effects will result. However, if a writer and some other process (either a reader or a writer) access the shared object simultaneously, chaos may ensue.

To ensure that these difficulties do not arise, we require that the writers have exclusive access to the shared object. This synchronization problem is referred to as the *readers–writers* problem. Since it was originally stated, it has

```

do {
    wait(full);
    wait(mutex);
    ...
    remove an item from buffer to nextc
    ...
    signal(mutex);
    signal(empty);
    ...
    consume the item in nextc
    ...
} while(1);

```

Figure 7.13 The structure of the consumer process.

```

    wait(wrt);
    ...
        writing is performed
    ...
    signal(wrt);

```

Figure 7.14 The structure of a writer process.

been used to test nearly every new synchronization primitive. The readers-writers problem has several variations, all involving priorities. The simplest one, referred to as the *first* readers-writers problem, requires that no reader will be kept waiting unless a writer has already obtained permission to use the shared object. In other words, no reader should wait for other readers to finish simply because a writer is waiting. The *second* readers-writers problem requires that, once a writer is ready, that writer performs its write as soon as possible. In other words, if a writer is waiting to access the object, no new readers may start reading.

A solution to either problem may result in starvation. In the first case, writers may starve; in the second case, readers may starve. For this reason, other variants of the problem have been proposed. In this section, we present a solution to the first readers-writers problem. Refer to the Bibliographical Notes for relevant references on starvation-free solutions to the readers-writers problem.

In the solution to the first readers-writers problem, the reader processes share the following data structures:

```

semaphore mutex, wrt;
int readcount;

```

The semaphores `mutex` and `wrt` are initialized to 1; `readcount` is initialized to 0. The semaphore `wrt` is common to both the reader and writer processes. The `mutex` semaphore is used to ensure mutual exclusion when the variable `readcount` is updated. The `readcount` variable keeps track of how many processes are currently reading the object. The semaphore `wrt` functions as a mutual-exclusion semaphore for the writers. It is also used by the first or last reader that enters or exits the critical section. It is not used by readers who enter or exit while other readers are in their critical sections.

The code for a writer process is shown in Figure 7.14; the code for a reader process is shown in Figure 7.15. Note that, if a writer is in the critical section and n readers are waiting, then one reader is queued on `wrt`, and $n - 1$ readers are queued on `mutex`. Also observe that, when a writer executes `signal(wrt)`, we may resume the execution of either the waiting readers or a single waiting writer. The selection is made by the scheduler.

```

wait(mutex);
readcount++;
if (readcount == 1)
    wait(wrt);
signal(mutex);
...
reading is performed
...
wait(mutex);
readcount--;
if (readcount == 0)
    signal(wrt);
signal(mutex);

```

Figure 7.15 The structure of a reader process.

7.5.3 The Dining-Philosophers Problem

Consider five philosophers who spend their lives thinking and eating. The philosophers share a common circular table surrounded by five chairs, each belonging to one philosopher. In the center of the table is a bowl of rice, and the table is laid with five single chopsticks (Figure 7.16). When a philosopher thinks, she does not interact with her colleagues. From time to time, a philosopher gets hungry and tries to pick up the two chopsticks that are closest to her (the chopsticks that are between her and her left and right neighbors). A philosopher may pick up only one chopstick at a time. Obviously, she cannot pick up a chopstick that is already in the hand of a neighbor. When a hungry philosopher has both her chopsticks at the same time, she eats without releas-

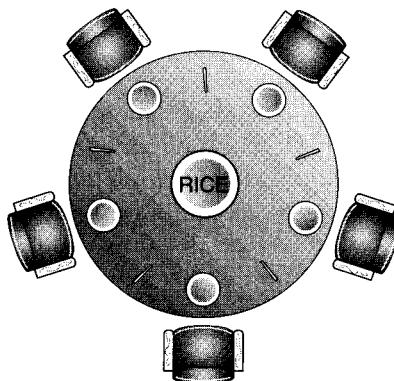


Figure 7.16 The situation of the dining philosophers.

```

do {
    wait(chopstick[i]);
    wait(chopstick[(i+1) % 5]);
    ...
    eat
    ...
    signal(chopstick[i]);
    signal(chopstick[(i+1) % 5]);
    ...
    think
    ...
} while (1);

```

Figure 7.17 The structure of philosopher *i*.

ing her chopsticks. When she is finished eating, she puts down both of her chopsticks and starts thinking again.

The *dining-philosophers* problem is considered a classic synchronization problem, neither because of its practical importance nor because computer scientists dislike philosophers, but because it is an example of a large class of concurrency-control problems. It is a simple representation of the need to allocate several resources among several processes in a deadlock- and starvation-free manner.

One simple solution is to represent each chopstick by a semaphore. A philosopher tries to grab the chopstick by executing a `wait` operation on that semaphore; she releases her chopsticks by executing the `signal` operation on the appropriate semaphores. Thus, the shared data are

```
semaphore chopstick[5];
```

where all the elements of `chopstick` are initialized to 1. The structure of philosopher *i* is shown in Figure 7.17.

Although this solution guarantees that no two neighbors are eating simultaneously, it nevertheless must be rejected because it has the possibility of creating a deadlock. Suppose that all five philosophers become hungry simultaneously, and each grabs her left chopstick. All the elements of `chopstick` will now be equal to 0. When each philosopher tries to grab her right chopstick, she will be delayed forever.

Several possible remedies to the deadlock problem are listed next. In Section 7.7, we present a solution to the *dining-philosophers* problem that ensures freedom from deadlocks.

- Allow at most four philosophers to be sitting simultaneously at the table.
- Allow a philosopher to pick up her chopsticks only if both chopsticks are available (to do this she must pick them up in a critical section).

- Use an asymmetric solution; that is, an odd philosopher picks up first her left chopstick and then her right chopstick, whereas an even philosopher picks up her right chopstick and then her left chopstick.

Finally, any satisfactory solution to the dining-philosophers problem must guard against the possibility that one of the philosophers will starve to death. A deadlock-free solution does not necessarily eliminate the possibility of starvation.

7.6 ■ Critical Regions

Although semaphores provide a convenient and effective mechanism for process synchronization, their incorrect use can still result in timing errors that are difficult to detect, since these errors happen only if some particular execution sequences take place, and these sequences do not always occur.

We have seen an example of such types of errors in the use of counters in our solution to the *producer-consumer* problem (Section 7.1). In that example, the timing problem happened only rarely, and even then the counter value appeared to be a reasonable value—off by only 1. Nevertheless, this solution is obviously not an acceptable one. It is for this reason that semaphores were introduced in the first place.

Unfortunately, such timing errors can still occur with the use of semaphores. To illustrate how, let us review the solution to the critical-section problem using semaphores. All processes share a semaphore variable `mutex`, which is initialized to 1. Each process must execute `wait(mutex)` before entering the critical section, and `signal(mutex)` afterward. If this sequence is not observed, two processes may be in their critical sections simultaneously.

Let us examine the various difficulties that may result. Note that these difficulties will arise if even a *single* process is not well behaved. This situation may be the result of an honest programming error or of an uncooperative programmer.

- Suppose that a process interchanges the order in which the `wait` and `signal` operations on the semaphore `mutex` are executed, resulting in the following execution:

```
signal(mutex);
...
critical section
...
wait(mutex);
```

In this situation, several processes may be executing in their critical section simultaneously, violating the mutual-exclusion requirement. This error

may be discovered only if several processes are simultaneously active in their critical sections. Note that this situation may not always be reproducible.

- Suppose that a process replaces `signal(mutex)` with `wait(mutex)`. That is, it executes

```
wait(mutex);
...
critical section
...
wait(mutex);
```

In this case, a deadlock will occur.

- Suppose that a process omits the `wait(mutex)`, or the `signal(mutex)`, or both. In this case, either mutual exclusion is violated or a deadlock will occur.

These examples illustrate that various types of errors can be generated easily when semaphores are used incorrectly to solve the critical-section problem. Similar problems may arise in the other synchronization models we discussed in Section 7.5.

To deal with the type of errors we have outlined, a number of high-level language constructs have been introduced. In this section, we describe one fundamental high-level synchronization construct—the **critical region** (sometimes referred to as **conditional critical region**). In Section 7.7, we present another fundamental synchronization construct—the **monitor**. In our presentation of these two constructs, we assume that a process consists of some local data, and a sequential program that can operate on the data. The local data can be accessed by only the sequential program that is encapsulated within the same process. That is, one process cannot directly access the local data of another process. Processes can, however, share global data.

The critical-region high-level language synchronization construct requires that a variable `v` of type `T`, which is to be shared among many processes, be declared as

```
v: shared T;
```

The variable `v` can be accessed only inside a `region` statement of the following form:

```
region v when B do S;
```

This construct means that, while statement `S` is being executed, no other process can access the variable `v`. The expression `B` is a Boolean expression

that governs the access to the critical region. When a process tries to enter the critical-section region, the Boolean expression B is evaluated. If the expression is **true**, statement S is executed. If it is **false**, the process relinquishes the mutual exclusion and is delayed until B becomes **true** and no other process is in the region associated with v. Thus, if the two statements,

```
region v when (true) S1;
region v when (true) S2;
```

are executed concurrently in distinct sequential processes, the result will be equivalent to the sequential execution “S1 followed by S2” or “S2 followed by S1.”

The critical-region construct guards against certain simple errors associated with the semaphore solution to the critical-section problem that may be made by a programmer. Note that it does not necessarily eliminate all synchronization errors; rather, it reduces their number. If errors occur in the logic of the program, reproducing a particular sequence of events may not be simple.

The critical-region construct can be effectively used to solve certain general synchronization problems. To illustrate, let us code the bounded-buffer scheme. The buffer space and its pointers are encapsulated in

```
struct buffer {
    item pool[n];
    int count, in, out;
};
```

The producer process inserts a new item **nextp** into the shared buffer by executing

```
region buffer when (count < n) {
    pool[in] = nextp;
    in = (in+1) % n;
    count++;
}
```

The consumer process removes an item from the shared buffer and puts it in **nextc** by executing

```
region buffer when (count > 0) {
    nextc = pool[out];
    out = (out+1) % n;
    count--;
}
```

Let us illustrate how the conditional critical region could be implemented by a compiler. With each shared variable, the following variables are associated:

```
semaphore mutex, first_delay, second_delay;
int first_count, second_count;
```

The semaphore `mutex` is initialized to 1; the semaphores `first_delay` and `second_delay` are initialized to 0. The integers `first_count` and `second_count` are initialized to 0.

Mutually exclusive access to the critical section is provided by `mutex`. If a process cannot enter the critical section because the Boolean condition B is `false`, it initially waits on the `first_delay` semaphore. A process waiting on the `first_delay` semaphore is eventually moved to the `second_delay` semaphore before it is allowed to reevaluate its Boolean condition B. We keep track of the number of processes waiting on `first_delay` and `second_delay`, with `first_count` and `second_count` respectively.

```
wait(mutex);
while (!B) {
    first_count++;
    if (second_count > 0)
        signal(second_delay);
    else
        signal(mutex);
    wait(first_delay);
    first_count--;
    second_count++;
    if (first_count > 0)
        signal(first_delay);
    else
        signal(second_delay);
    wait(second_delay);
    second_count--;
}
S;
if (first_count > 0)
    signal(first_delay);
else if (second_count > 0)
    signal(second_delay);
else
    signal(mutex);
```

Figure 7.18 Implementation of the conditional-region construct.

When a process leaves the critical section, it may have changed the value of some Boolean condition B that prevented another process from entering the critical section. Accordingly, we must trace through the queue of processes waiting on `first_delay` and `second_delay` (in that order) allowing each process to test its Boolean condition. When a process tests its Boolean condition (during this trace), it may discover that the latter now evaluates to the value `true`. In this case, the process enters its critical section. Otherwise, the process must wait again on the `first_delay` and `second_delay` semaphores, as described previously. Accordingly, for a shared variable `x`, the statement

```
region x when (B) S;
```

can be implemented as shown in Figure 7.18. Note that this implementation requires the reevaluation of the expression B for any waiting processes every time a process leaves the critical region. If several processes are delayed waiting for their respective Boolean expressions to become `true`, this reevaluation overhead may result in inefficient code. We can use various optimization methods to reduce this overhead. Refer to the Bibliographical Notes for relevant references.

```
monitor monitor-name
{
    shared variable declarations

    procedure body P1 ( ... ) {
        ...
    }
    procedure body P2 ( ... ) {
        ...
    }
    .
    .
    .
    procedure body Pn ( ... ) {
        ...
    }
    {
        initialization code
    }
}
```

Figure 7.19 Syntax of a monitor.

7.7 ■ Monitors

Another high-level synchronization construct is the monitor type. A **monitor** is characterized by a set of programmer-defined operators. The representation of a monitor type consists of declarations of variables whose values define the state of an instance of the type, as well as the bodies of procedures or functions that implement operations on the type. The syntax of a monitor is shown in Figure 7.19.

The representation of a monitor type cannot be used directly by the various processes. Thus, a procedure defined within a monitor can access only those variables declared locally within the monitor and its formal parameters. Similarly, the local variables of a monitor can be accessed by only the local procedures.

The monitor construct ensures that only one process at a time can be active within the monitor. Consequently, the programmer does not need to code this synchronization constraint explicitly (Figure 7.20). However, the monitor construct, as defined so far, is not sufficiently powerful for modeling some synchronization schemes. For this purpose, we need to define additional synchronization mechanisms. These mechanisms are provided by the condition

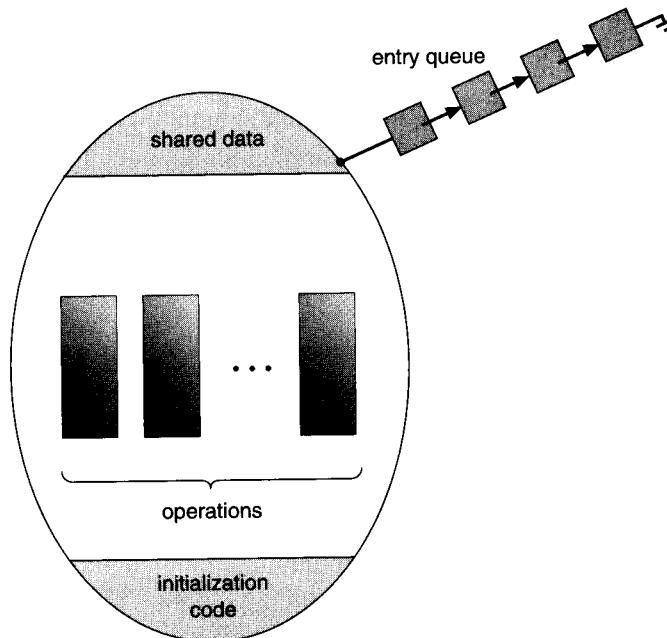


Figure 7.20 Schematic view of a monitor.

construct. A programmer who needs to write her own tailor-made synchronization scheme can define one or more variables of type *condition*:

```
condition x, y;
```

The only operations that can be invoked on a condition variable are **wait** and **signal**. The operation

```
x.wait();
```

means that the process invoking this operation is suspended until another process invokes

```
x.signal();
```

The **x.signal()** operation resumes exactly one suspended process. If no process is suspended, then the **signal** operation has no effect; that is, the state of **x** is as though the operation were never executed (Figure 7.21). Contrast this operation with the **signal** operation associated with semaphores, which always affects the state of the semaphore.

Now suppose that, when the **x.signal()** operation is invoked by a process P, there is a suspended process Q associated with condition **x**. Clearly, if the

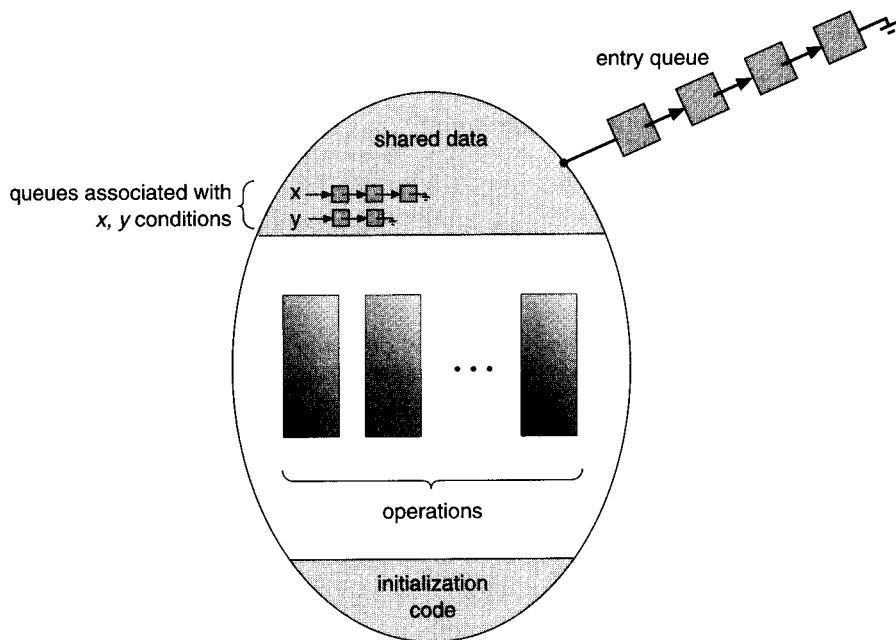


Figure 7.21 Monitor with condition variables.

suspended process Q is allowed to resume its execution, the signaling process P must wait. Otherwise, both P and Q will be active simultaneously within the monitor. Note, however, that both processes can conceptually continue with their execution. Two possibilities exist:

1. P either waits until Q leaves the monitor, or waits for another condition.
2. Q either waits until P leaves the monitor, or waits for another condition.

There are reasonable arguments in favor of adopting either option 1 or option 2. Since P was already executing in the monitor, choice 2 seems more reasonable. However, if we allow process P to continue, the “logical” condition for which Q was waiting may no longer hold by the time Q is resumed.

Choice 1 was advocated by Hoare, mainly because the preceding argument in favor of it translates directly to simpler and more elegant proof rules. A compromise between these two choices was adopted in the language Concurrent C. When process P executes the `signal` operation, process Q is immediately resumed. This model is less powerful than Hoare’s, because a process cannot signal more than once during a single procedure call.

Let us illustrate these concepts by presenting a deadlock-free solution to the dining-philosophers problem. Recall that a philosopher is allowed to pick up her chopsticks only if both of them are available. To code this solution, we need to distinguish among three states in which a philosopher may be. For this purpose, we introduce the following data structure:

```
enum {thinking, hungry, eating} state[5];
```

Philosopher i can set the variable `state[i] = eating` only if her two neighbors are not eating: $(\text{state}[(i+4) \% 5] \neq \text{eating})$ and $(\text{state}[(i+1) \% 5] \neq \text{eating})$.

We also need to declare

```
condition self[5];
```

where philosopher i can delay herself when she is hungry, but is unable to obtain the chopsticks she needs.

We are now in a position to describe our solution to the dining-philosopher problem. The distribution of the chopsticks is controlled by the monitor `dp`, whose definition is shown in Figure 7.22. Each philosopher, before starting to eat, must invoke the operation `pickup`. This may result in the suspension of the philosopher process. After the successful completion of the operation, the philosopher may eat. Following this, the philosopher invokes the `putdown` operation, and may start to think. Thus, philosopher i must invoke the operations `pickup` and `putdown` in the following sequence:

```

monitor dp
{
    enum {thinking, hungry, eating} state[5];
    condition self[5];

    void pickup(int i) {
        state[i] = hungry;
        test(i);
        if (state[i] != eating)
            self[i].wait();
    }

    void putdown(int i) {
        state[i] = thinking;
        test((i + 4) % 5);
        test((i + 1) % 5);
    }

    void test(int i) {
        if ((state[(i + 4) % 5] != eating) &&
            (state[i] == hungry) &&
            (state[(i + 1) % 5] != eating)) {
            state[i] = eating;
            self[i].signal();
        }
    }

    void init() {
        for (int i = 0; i < 5; i++)
            state[i] = thinking;
    }
}

```

Figure 7.22 A monitor solution to the dining-philosopher problem.

```

dp.pickup(i);
...
eat
...
dp.putdown(i);

```

It is easy to show that this solution ensures that no two neighbors are eating simultaneously, and that no deadlocks will occur. We note, however, that it is possible for a philosopher to starve to death. We shall not present a

solution to this problem, but rather shall leave it as an exercise for you. We shall now consider a possible implementation of the monitor mechanism using semaphores. For each monitor, a semaphore `mutex` (initialized to 1) is provided. A process must execute `wait(mutex)` before entering the monitor, and must execute `signal(mutex)` after leaving the monitor.

Since a signaling process must wait until the resumed process either leaves or waits, an additional semaphore, `next`, is introduced, initialized to 0, on which the signaling processes may suspend themselves. An integer variable `next_count` will also be provided to count the number of processes suspended on `next`. Thus, each external procedure `F` will be replaced by

```

wait(mutex);
...
body of F
...
if (next_count > 0)
    signal(next);
else
    signal(mutex);

```

Mutual exclusion within a monitor is ensured.

We can now describe how condition variables are implemented. For each condition `x`, we introduce a semaphore `x.sem` and an integer variable `x.count`, both initialized to 0. The operation `x.wait` can now be implemented as

```

x_count++;
if (next_count > 0)
    signal(next);
else
    signal(mutex);
wait(x.sem);
x_count--;

```

The operation `x.signal()` can be implemented as

```

if (x_count > 0) {
    next_count++;
    signal(x.sem);
    wait(next);
    next_count--;
}

```

This implementation is applicable to the definitions of monitors given by both Hoare and Brinch-Hansen. In some cases, however, the generality of the

implementation is unnecessary, and a significant improvement in efficiency is possible. We leave this problem to you in Exercise 7.13.

We turn now to the subject of process-resumption order within a monitor. If several processes are suspended on condition *x*, and an *x.signal* operation is executed by some process, then how do we determine which of the suspended processes should be resumed next? One simple solution is to use a FCFS ordering, so that the process waiting the longest is resumed first. In many circumstances, however, such a simple scheduling scheme is not adequate. For this purpose, the **conditional-wait** construct can be used; it has the form

```
x.wait(c);
```

where *c* is an integer expression that is evaluated when the *wait* operation is executed. The value of *c*, which is called a **priority number**, is then stored with the name of the process that is suspended. When *x.signal* is executed, the process with the smallest associated priority number is resumed next.

To illustrate this new mechanism, we consider the monitor shown in Figure 7.23, which controls the allocation of a single resource among competing processes. Each process, when requesting an allocation of its resources, specifies the maximum time it plans to use the resource. The monitor allocates the resource to that process that has the shortest time-allocation request.

A process that needs to access the resource in question must observe the following sequence:

```
monitor ResourceAllocation
{
    boolean busy;
    condition x;

    void acquire(int time) {
        if (busy)
            x.wait(time);
        busy = true;
    }

    void release() {
        busy = false;
        x.signal();
    }
    void init() {
        busy = false;
    }
}
```

Figure 7.23 A monitor to allocate a single resource.

```
R.acquire(t);
...
access the resource;
...
R.release();
```

where R is an instance of type `ResourceAllocation`.

Unfortunately, the monitor concept cannot guarantee that the preceding access sequences will be observed. In particular,

- A process might access the resource without first gaining access permission to that resource.
- A process might never release the resource once it has been granted access to that resource.
- A process might attempt to release a resource that it never requested.
- A process might request the same resource twice (without first releasing that resource).

The same difficulties are encountered with the critical section construct, and these difficulties are similar in nature to those that encouraged us to develop the critical-region and monitor constructs in the first place. Previously, we had to worry about the correct use of semaphores. Now, we have to worry about the correct use of higher-level programmer-defined operations, with which the compiler can no longer assist us.

One possible solution to the above problem is to include the resource-access operations within the `ResourceAllocation` monitor. However, this solution will result in scheduling being done according to the built-in monitor-scheduling algorithm, rather than by the one we have coded.

To ensure that the processes observe the appropriate sequences, we must inspect all the programs that make use of the `ResourceAllocation` monitor and its managed resource. There are two conditions that we must check to establish the correctness of this system. First, user processes must always make their calls on the monitor in a correct sequence. Second, we must be sure that an uncooperative process does not simply ignore the mutual-exclusion gateway provided by the monitor and try to access the shared resource directly, without using the access protocols. Only if these two conditions can be ensured can we guarantee that no time-dependent errors will occur, and that the scheduling algorithm will not be defeated.

Although this inspection may be possible for a small, static system, it is not reasonable for a large system or for a dynamic system. This access-control problem can be solved only by additional mechanisms that will be elaborated in Chapter 18.

7.8 ■ OS Synchronization

We next describe the synchronization mechanisms provided by the Solaris and Windows 2000 operating systems.

7.8.1 Synchronization in Solaris 2

The Solaris 2 operating system was designed to provide real-time computing, be multithreaded, and support multiple processors. To control access to critical sections, Solaris 2 provides adaptive mutexes, condition variables, semaphores, reader-writer locks, and turnstiles. Solaris 2 implements semaphores and condition variables in the same way as they have been fundamentally presented in Sections 7.4 and 7.7. In this section we describe the adaptive mutexes, reader-writer locks, and turnstiles.

An **adaptive mutex** protects access to every critical data item. On a multiprocessor system, an adaptive mutex starts as a standard semaphore implemented as a spinlock. If the data are locked and therefore already in use, the adaptive mutex does one of two things. If the lock is held by a thread that is currently running on another CPU, the thread spins while waiting for the lock to become available because the thread holding the lock is likely to be done soon. If the thread holding the lock is not currently in run state, the thread blocks, going to sleep until it is awakened by the lock being released. It is put to sleep so that it will avoid spinning when the lock will not be freed reasonably quickly. A lock held by a sleeping thread is likely to be in this category. On a uniprocessor system, the thread holding the lock is never running if the lock is being tested by another thread, because only one thread can run at a time. Therefore, on a uniprocessor system, threads always sleep rather than spin if they encounter a lock. We use the adaptive-mutex method to protect only those data that are accessed by short code segments. That is, a mutex is used if a lock will be held for less than a few hundred instructions. If the code segment is longer than that, spin waiting will be exceedingly inefficient. For longer code segments, condition variables and semaphores are used. If the desired lock is already held, the thread issues a wait and sleeps. When a thread frees the lock, it issues a signal to the next sleeping thread in the queue. The extra cost of putting a thread to sleep and waking it, and of the associated context switches, is less than the cost of wasting several hundred instructions waiting in a spinlock.

The readers-writers locks are used to protect data that are accessed frequently, but usually only in a read-only manner. In these circumstances, readers-writers locks are more efficient than semaphores, because multiple threads may be reading data concurrently, whereas semaphores would always serialize access to the data. Readers-writers locks are relatively expensive to implement, so again they are used on only long sections of code.

Solaris 2 uses turnstiles to order the list of threads waiting to acquire either an adaptive mutex or a reader-writer lock. A **turnstile** is a queue structure

containing threads blocked on a lock. For example, if one thread currently owns the lock for a synchronized object, all other threads trying to acquire the lock will block and enter the turnstile for that lock. When the lock is released, the kernel selects a thread from the turnstile as the next owner of the lock. Each synchronized object with at least one thread blocked on the object's lock requires a separate turnstile. However, rather than associating a turnstile with each synchronized object, Solaris 2 gives each kernel thread its own turnstile. The turnstile for the first thread to block on a synchronized object becomes the turnstile for the object itself. Subsequent threads blocking on the lock will be added to this turnstile. When the initial thread ultimately releases the lock, it gains a new turnstile from a list of free turnstiles the kernel maintains. To prevent a **priority inversion**, turnstiles are organized according to a **priority inheritance protocol** (Section 6.5). This means that if a lower-priority thread currently holds a lock that a higher-priority thread is blocked on, the thread with the lower priority will temporarily inherit the priority of the higher-priority thread. Upon releasing the lock, the thread will revert to its original priority.

Note that the locking mechanisms used by the kernel are implemented for user-level threads as well, so the same types of locks are available inside and outside the kernel. A crucial implementation difference is the priority-inheritance protocol. Kernel-locking routines adhere to the kernel priority-inheritance methods used by the scheduler, as described in Section 6.5; user-level thread-locking mechanisms do not provide this functionality.

Because locks are used frequently, and typically are used for crucial kernel functions, fine-tuning their implementation and use can provide great performance gains. To optimize Solaris 2 performance, developers continually refine the locking methods.

7.8.2 Synchronization in Windows 2000

The Windows 2000 operating system is a multithreaded kernel that also provides support for real-time applications and multiple processors. When the Windows 2000 kernel accesses a global resource on a uniprocessor system, it temporarily masks interrupts for all interrupt handlers that may also access the global resource. On a multiprocessor system, Windows 2000 protects access to global resources using spinlocks. Just as in Solaris 2, the kernel only uses spinlocks only to protect short code segments. Furthermore, for reasons of efficiency, the kernel ensures that a thread will never be preempted while holding a spinlock. For thread synchronization outside of the kernel, Windows 2000 provides **dispatcher objects**. Using a dispatcher object, a thread can synchronize according to several different mechanisms including mutexes, semaphores, and events. Shared data can be protected by requiring a thread to gain ownership of a mutex to access the data and to release ownership when it is finished. Events are a synchronization mechanism that may be used much as are condition

variables; that is, they may notify a waiting thread when a desired condition occurs.

Dispatcher objects may be in either a **signaled** or **nonsignaled** state. A signaled state indicates that an object is available and a thread will not block when acquiring the object. A nonsignaled state indicates that an object is not available and that a thread will block when attempting to acquire the object. There is a relationship between the state of a dispatcher object and the state of a thread. When a thread blocks on a nonsignaled dispatcher object, its state changes from ready to waiting and the thread is placed in a waiting queue for that object. When the state for the dispatcher object moves to signaled, the kernel checks if there are any threads waiting on the object. If so, the kernel moves one—or possibly more—threads from the waiting state to the ready state where they can resume executing. The number of threads the kernel selects from the waiting queue depends upon the type of dispatcher object they are waiting upon. The kernel will select only one thread from the waiting queue for a mutex, since a mutex object may be “owned” by only a single thread. For an event object, the kernel will select all threads that are waiting for the event.

Let us use a mutex lock as an illustrating example of dispatcher objects and thread states. If a thread tries to acquire a mutex dispatcher object that is in a nonsignaled state, that thread will be suspended and placed in a waiting queue for the mutex object. When the mutex moves to the signaled state (the result of another thread releasing the lock on the mutex), the thread waiting on the mutex will:

1. Be moved from the wait to the ready state,
2. Acquire the mutex lock.

7.9 ■ Atomic Transactions

The mutual exclusion of critical sections ensures that the critical sections are executed atomically. That is, if two critical sections are executed concurrently, the result is equivalent to their sequential execution in some unknown order. Although this property is useful in many application domains, in many cases we would like to make sure that a critical section forms a single logical unit of work that either is performed in its entirety or is not performed at all. An example is funds transfer, in which one account is debited and another is credited. Clearly, it is essential for data consistency that either both the credit and debit occur, or that neither occur.

The remainder of this section is related to the field of database systems. **Databases** are concerned with the storage and retrieval of data, and with the consistency of the data. Recently, there has been an upsurge of interest in using database-systems techniques in operating systems. Operating systems can be viewed as manipulators of data; as such, they can benefit from the advanced

techniques and models available from database research. For instance, many of the ad hoc techniques used in operating systems to manage files could be more flexible and powerful if more formal database methods were used in their place. In Sections 7.9.2 to 7.9.4, we describe what these database techniques are and how they can be used by operating systems.

7.9.1 System Model

A collection of instructions (or operations) that performs a single logical function is called a **transaction**. A major issue in processing transactions is the preservation of atomicity despite the possibility of failures within the computer system. In this section, we describe various mechanisms for ensuring transaction atomicity. We do so by first considering an environment where only one transaction can be executing at a time. Then, we consider the case where multiple transactions are active simultaneously. A transaction is a program unit that accesses and possibly updates various data items that may reside on the disk within some files. From our point of view, a transaction is simply a sequence of **read** and **write** operations, terminated by either a **commit** operation or an **abort** operation. A **commit** operation signifies that the transaction has terminated its execution successfully, whereas an **abort** operation signifies that the transaction had to cease its normal execution due to some logical error. A terminated transaction that has completed its execution successfully is **committed**; otherwise, it is **aborted**. The effect of a committed transaction cannot be undone by abortion of the transaction.

A transaction may also cease its normal execution due to a system failure. In either case, since an aborted transaction may have already modified the various data that it has accessed, the state of these data may not be the same as it would be had the transaction executed atomically. So that the atomicity property is ensured, an aborted transaction must have no effect on the state of the data that it has already modified. Thus, the state of the data accessed by an aborted transaction must be restored to what it was just before the transaction started executing. We say that such a transaction has been **rolled back**. It is part of the responsibility of the system to ensure this property.

To determine how the system should ensure atomicity, we need first to identify the properties of devices used for storing the various data accessed by the transactions. Various types of storage media are distinguished by their relative speed, capacity, and resilience to failure.

- **Volatile Storage:** Information residing in volatile storage does not usually survive system crashes. Examples of such storage are main and cache memory. Access to volatile storage is extremely fast, both because of the speed of the memory access itself and because it is possible to access directly any data item in volatile storage.

- **Nonvolatile Storage:** Information residing in nonvolatile storage usually survives system crashes. Examples of media for such storage are disks and magnetic tapes. Disks are more reliable than is main memory, but are less reliable than are magnetic tapes. Both disks and tapes, however, are subject to failure, which may result in loss of information. Currently, nonvolatile storage is slower than volatile storage by several orders of magnitude, because disk and tape devices are electromechanical and require physical motion to access data.
- **Stable Storage:** Information residing in stable storage is *never* lost (*never* should be taken with a grain of salt, since theoretically such absolutes cannot be guaranteed). To implement an approximation of such storage, we need to replicate information in several nonvolatile storage caches (usually disk) with independent failure modes, and to update the information in a controlled manner (Section 14.7).

Here, we are concerned only with ensuring transaction atomicity in an environment where failures result in the loss of information on volatile storage.

7.9.2 Log-Based Recovery

One way to ensure atomicity is to record, on stable storage, information describing all the modifications made by the transaction to the various data it accessed. The most widely used method for achieving this form of recording is **write-ahead logging**. The system maintains, on stable storage, a data structure called the **log**. Each log record describes a single operation of a transaction write, and has the following fields:

- **Transaction Name:** The unique name of the transaction that performed the `write` operation
- **Data Item Name:** The unique name of the data item written
- **Old Value:** The value of the data item prior to the `write` operation
- **New Value:** The value that the data item will have after the `write`

Other special log records exist to record significant events during transaction processing, such as the start of a transaction and the commit or abort of a transaction.

Before a transaction T_i starts its execution, the record $\langle T_i \text{ starts} \rangle$ is written to the log. During its execution, any `write` operation by T_i is preceded by the writing of the appropriate new record to the log. When T_i commits, the record $\langle T_i \text{ commits} \rangle$ is written to the log.

Because the information in the log is used in reconstructing the state of the data items accessed by the various transactions, we cannot allow the actual update to a data item to take place before the corresponding log record is

written out to stable storage. We therefore require that, prior to a `write(X)` operation being executed, the log records corresponding to X be written onto stable storage.

Note the performance penalty inherent in this system. Two physical writes are required for every logical write requested. Also, more storage is needed: for the data themselves and for the log of the changes. In cases where the data are extremely important and fast failure recovery is necessary, the price is worth the functionality.

Using the log, the system can handle any failure that does not result in the loss of information on nonvolatile storage. The recovery algorithm uses two procedures:

- `undo(T_i)`, which restores the value of all data updated by transaction T_i to the old values
- `redo(T_i)`, which sets the value of all data updated by transaction T_i to the new values

The set of data updated by T_i and their respective old and new values can be found in the log.

The `undo` and `redo` operations must be idempotent (that is, multiple executions of an operation have the same result as does one execution) to guarantee correct behavior, even if a failure occurs during the recovery process.

If a transaction T_i aborts, then we can restore the state of the data that it has updated by simply executing `undo(T_i)`. If a system failure occurs, we restore the state of all updated data by consulting the log to determine which transactions need to be redone and which need to be undone. This classification of transactions is accomplished as follows:

- Transaction T_i needs to be undone if the log contains the `< T_i starts >` record, but does not contain the `< T_i commits >` record.
- Transaction T_i needs to be redone if the log contains both the `< T_i starts >` and the `< T_i commits >` records.

7.9.3 Checkpoints

When a system failure occurs, we must consult the log to determine those transactions that need to be redone and those that need to be undone. In principle, we need to search the entire log to make these determinations. There are two major drawbacks to this approach:

1. The searching process is time-consuming.
2. Most of the transactions that, according to our algorithm, need to be redone have already actually updated the data that the log says they need to

modify. Although redoing the data modifications will cause no harm (due to idempotency), it will nevertheless cause recovery to take longer.

To reduce these types of overhead, we introduce the concept of **checkpoints**. During execution, the system maintains the write-ahead log. In addition, the system periodically performs checkpoints that require the following sequence of actions to take place:

1. Output all log records currently residing in volatile storage (usually main memory) onto stable storage.
2. Output all modified data residing in volatile storage to the stable storage.
3. Output a log record `<checkpoint>` onto stable storage.

The presence of a `<checkpoint>` record in the log allows the system to streamline its recovery procedure. Consider a transaction T_i that committed prior to the checkpoint. The `<Ti commits>` record appears in the log before the `<checkpoint>` record. Any modifications made by T_i must have been written to stable storage either prior to the checkpoint, or as part of the checkpoint itself. Thus, at recovery time, there is no need to perform a `redo` operation on T_i .

This observation allows us to refine our previous recovery algorithm. After a failure has occurred, the recovery routine examines the log to determine the most recent transaction T_i that started executing before the most recent checkpoint took place. It finds such a transaction by searching the log backward to find the first `<checkpoint>` record, and then finding the subsequent `<Ti start>` record.

Once transaction T_i has been identified, the `redo` and `undo` operations need to be applied to only transaction T_i and all transactions T_j that started executing after transaction T_i . Let us denote these transactions by the set T . The remainder of the log can thus be ignored. The recovery operations that are required are as follows:

- For all transactions T_k in T such that the record `<Tk commits>` appears in the log, execute `redo(Tk)`.
- For all transactions T_k in T that have no `<Tk commits>` record in the log, execute `undo(Tk)`.

7.9.4 Concurrent Atomic Transactions

Because each transaction is atomic, the concurrent execution of transactions must be equivalent to the case where these transactions executed serially in some arbitrary order. This property, called **serializability**, can be maintained by simply executing each transaction within a critical section. That is, all

transactions share a common semaphore *mutex*, which is initialized to 1. When a transaction starts executing, its first action is to execute `wait(mutex)`. After the transaction either commits or aborts, it executes `signal(mutex)`.

Although this scheme ensures the atomicity of all concurrently executing transactions, it is nevertheless too restrictive. As we shall see, in many cases we can allow transactions to overlap their execution, while maintaining serializability. A number of different **concurrency-control** algorithms ensure serializability. These are described below.

7.9.4.1 Serializability

Consider a system with two data items *A* and *B* that are both read and written by two transactions T_0 and T_1 . Suppose that these transactions are executed atomically in the order T_0 followed by T_1 . This execution sequence, which is called a **schedule**, is represented in Figure 7.24. In schedule 1 of Figure 7.24, the sequence of instruction steps is in chronological order from top to bottom, with instructions of T_0 appearing in the left column and instructions of T_1 appearing in the right column.

A schedule where each transaction is executed atomically is called a **serial schedule**. Each serial schedule consists of a sequence of instructions from various transactions where the instructions belonging to one single transaction appear together in that schedule. Thus, for a set of n transactions, there exist $n!$ different valid serial schedules. Each serial schedule is correct, because it is equivalent to the atomic execution of the various participating transactions, in some arbitrary order.

If we allow the two transactions to overlap their execution, then the resulting schedule is no longer serial. A **nonserial schedule** does not necessarily imply that the resulting execution is incorrect (that is, is not equivalent to a serial schedule). To see that this is the case, we need to define the notion of **conflicting operations**. Consider a schedule S in which there are two consecutive operations O_i and O_j of transactions T_i and T_j , respectively. We say that O_i and

T_0	T_1
read(<i>A</i>)	
write(<i>A</i>)	
read(<i>B</i>)	
write(<i>B</i>)	
	read(<i>A</i>)
	write(<i>A</i>)
	read(<i>B</i>)
	write(<i>B</i>)

Figure 7.24 Schedule 1: A serial schedule in which T_0 is followed by T_1 .

T_0	T_1
read(A)	
write(A)	
	read(A)
	write(A)
read(B)	
write(B)	
	read(B)
	write(B)

Figure 7.25 Schedule 2: A concurrent serializable schedule.

O_i conflict if they access the same data item and at least one of these operations is a write operation. To illustrate the concept of conflicting operations, we consider the nonserial schedule 2 of Figure 7.25. The write(A) operation of T_0 conflicts with the read(A) operation of T_1 . However, the write(A) operation of T_1 does not conflict with the read(B) operation of T_0 , because the two operations access different data items.

Let O_i and O_j be consecutive operations of a schedule S . If O_i and O_j are operations of different transactions and O_i and O_j do not conflict, then we can swap the order of O_i and O_j to produce a new schedule S' . We expect S to be equivalent to S' , as all operations appear in the same order in both schedules, except for O_i and O_j , whose order does not matter.

Let us illustrate the swapping idea by considering again schedule 2 of Figure 7.25. As the write(A) operation of T_1 does not conflict with the read(B) operation of T_0 , we can swap these operations to generate an equivalent schedule. Regardless of the initial system state, both schedules produce the same final system state. Continuing with this procedure of swapping nonconflicting operations, we get:

- Swap the read(B) operation of T_0 with the read(A) operation of T_1 .
- Swap the write(B) operation of T_0 with the write(A) operation of T_1 .
- Swap the write(B) operation of T_0 with the read(A) operation of T_1 .

The final result of these swaps is schedule 1 in Figure 7.24, which is a serial schedule. Thus, we have shown that schedule 2 is equivalent to a serial schedule. This result implies that, regardless of the initial system state, schedule 2 will produce the same final state as will some serial schedule.

If a schedule S can be transformed into a serial schedule S' by a series of swaps of nonconflicting operations, we say that a schedule S is **conflict serializable**. Thus, schedule 2 is conflict serializable, because it can be transformed into the serial schedule 1.

7.9.4.2 Locking Protocol

One way to ensure serializability is to associate with each data item a lock, and to require that each transaction follow a **locking protocol** that governs how locks are acquired and released. There are various modes in which a data item can be locked. In this section, we restrict our attention to two modes:

- **Shared:** If a transaction T_i has obtained a shared-mode lock (denoted by S) on data item Q , then T_i can read this item, but cannot write Q .
- **Exclusive:** If a transaction T_i has obtained an exclusive-mode lock (denoted by X) on data item Q , then T_i can both read and write Q .

We require that every transaction request a lock in an appropriate mode on data item Q , depending on the type of operations it will perform on Q .

To access a data item Q , transaction T_i must first lock Q in the appropriate mode. If Q is not currently locked, then the lock is granted, and T_i can now access it. However, if the data item Q is currently locked by some other transaction, then T_i may have to wait. More specifically, suppose that T_i requests an exclusive lock on Q . In this case, T_i must wait until the lock on Q is released. If T_i requests a shared lock on Q , then T_i must wait if Q is locked in exclusive mode. Otherwise, it can obtain the lock and access Q . Notice that this scheme is quite similar to the readers-writers algorithm discussed in Section 7.5.2.

A transaction may unlock a data item that it had locked at an earlier point. It must, however, hold a lock on a data item as long as it accesses that item. Moreover, it is not always desirable for a transaction to unlock a data item immediately after its last access of that data item, because serializability may not be ensured.

One protocol that ensures serializability is the **two-phase locking protocol**. This protocol requires that each transaction issue lock and unlock requests in two phases:

- **Growing Phase:** A transaction may obtain locks, but may not release any lock.
- **Shrinking Phase:** A transaction may release locks, but may not obtain any new locks.

Initially, a transaction is in the growing phase. The transaction acquires locks as needed. Once the transaction releases a lock, it enters the shrinking phase, and no more lock requests can be issued.

The two-phase locking protocol ensures conflict serializability (Exercise 7.23). It does not, however, ensure freedom from deadlock. We note that it

is possible that, for a set of transactions, there are conflict-serializable schedules that cannot be obtained through the two-phase locking protocol. However, to improve performance over two-phase locking, we need either to have additional information about the transactions or to impose some structure or ordering on the set of data.

7.9.4.3 Timestamp-Based Protocols

In the locking protocols described above, the order between every pair of conflicting transactions is determined at execution time by the first lock that they both request and that involves incompatible modes. Another method for determining the serializability order is to select an ordering among transactions in advance. The most common method for doing so is to use a **timestamp-ordering** scheme.

With each transaction T_i in the system, we associate a unique fixed timestamp, denoted by $TS(T_i)$. This timestamp is assigned by the system before the transaction T_i starts execution. If a transaction T_i has been assigned timestamp $TS(T_i)$, and later on a new transaction T_j enters the system, then $TS(T_i) < TS(T_j)$. There are two simple methods for implementing this scheme:

- Use the value of the system clock as the timestamp; that is, a transaction's timestamp is equal to the value of the clock when the transaction enters the system. This method will not work for transactions that occur on separate systems or for processors that do not share a clock.
- Use a logical counter as the timestamp; that is, a transaction's timestamp is equal to the value of the counter when the transaction enters the system. The counter is incremented after a new timestamp is assigned.

The timestamps of the transactions determine the serializability order. Thus, if $TS(T_i) < TS(T_j)$, then the system must ensure that the produced schedule is equivalent to a serial schedule in which transaction T_i appears before transaction T_j .

To implement this scheme, we associate with each data item Q two timestamp values:

- **W-timestamp(Q)**, which denotes the largest timestamp of any transaction that executed `write(Q)` successfully
- **R-timestamp(Q)**, which denotes the largest timestamp of any transaction that executed `read(Q)` successfully

These timestamps are updated whenever a new `read(Q)` or `write(Q)` instruction is executed.

The timestamp-ordering protocol ensures that any conflicting `read` and `write` operations are executed in timestamp order. This protocol operates as follows:

- Suppose that transaction T_i issues `read(Q)`:
 - If $\text{TS}(T_i) < \text{W-timestamp}()$, then this state implies that T_i needs to read a value of Q that was already overwritten. Hence, the `read` operation is rejected, and T_i is rolled back.
 - If $\text{TS}(T_i) \geq \text{W-timestamp}(Q)$, then the `read` operation is executed, and $\text{R-timestamp}(Q)$ is set to the maximum of $\text{R-timestamp}(Q)$ and $\text{TS}(T_i)$.
- Suppose that transaction T_i issues `write(Q)`:
 - If $\text{TS}(T_i) < \text{R-timestamp}(Q)$, then this state implies that the value of Q that T_i is producing was needed previously and T_i assumed that this value would never be produced. Hence, the `write` operation is rejected, and T_i is rolled back.
 - If $\text{TS}(T_i) < \text{W-timestamp}(Q)$, then this state implies that T_i is attempting to write an obsolete value of Q . Hence, this `write` operation is rejected, and T_i is rolled back.
 - Otherwise, the `write` operation is executed.

A transaction T_i , that is rolled back by the concurrency-control scheme as a result of the issuing of either a `read` or `write` operation is assigned a new timestamp and is restarted.

To illustrate this protocol, we consider schedule 3 of Figure 7.26 with transactions T_2 and T_3 . We assume that a transaction is assigned a timestamp immediately before its first instruction. Thus, in schedule 3, $\text{TS}(T_2) < \text{TS}(T_3)$, and the schedule is possible under the timestamp protocol.

This execution can also be produced by the two-phase locking protocol. However, some schedules are possible under the two-phase locking protocol but not under the timestamp protocol, and vice versa (Exercise 7.24).

The timestamp-ordering protocol ensures conflict serializability. This capability follows from the fact that conflicting operations are processed in time-

T_2	T_3
<code>read(B)</code>	<code>read(B)</code>
<code>read(A)</code>	<code>write(B)</code> <code>read(A)</code> <code>write(A)</code>

Figure 7.26 Schedule 3: A schedule possible under the timestamp protocol.

stamp order. The protocol ensures freedom from deadlock, because no transaction ever waits.

7.10 ■ Summary

Given a collection of cooperating sequential processes that share data, mutual exclusion must be provided. One solution is to ensure that a critical section of code is in use by only one process or thread at a time. Different algorithms exist for solving the critical-section problem, with the assumption that only storage interlock is available.

The main disadvantage of these user-coded solutions is that they all require busy waiting. Semaphores overcome this difficulty. Semaphores can be used to solve various synchronization problems and can be implemented efficiently, especially if hardware support for atomic operations is available.

Various different synchronization problems (such as the bounded-buffer problem, the readers-writers problem, and the dining-philosophers problem) are important mainly because they are examples of a large class of concurrency-control problems. These problems are used to test nearly every newly proposed synchronization scheme.

The operating system must provide the means to guard against timing errors. Several language constructs have been proposed to deal with these problems. Critical regions can be used to implement mutual-exclusion and arbitrary-synchronization problems safely and efficiently. Monitors provide the synchronization mechanism for sharing abstract data types. A condition variable provides a method for a monitor procedure to block its execution until it is signaled to continue.

Solaris 2 is an example of a modern operating system that implements a variety of locks to support multitasking, multithreading (including real-time threads), and multiprocessing. It uses adaptive mutexes for efficiency when protecting data from short code segments. Condition variables and readers-writers locks are used when longer sections of code need access to data. Solaris uses turnstiles to order the list of threads waiting to acquire an adaptive mutex or readers-writers lock.

Windows 2000 supports realtime processes and multiprocessing. When the kernel attempts to access global resources on uniprocessor systems, interrupts that may also access the global resource are masked. On multiprocessor systems, global resources are protected using spinlocks. Outside of the kernel, synchronization is provided using dispatcher objects. A dispatcher object may be used as a mutex, semaphore, or event. An event is a type of dispatcher object that behaves similar to condition variables.

A transaction is a program unit that must be executed atomically; that is, either all the operations associated with it are executed to completion, or none are performed. To ensure atomicity despite system failure, we can use a write-ahead log. All updates are recorded on the log, which is kept in stable storage.

If a system crash occurs, the information in the log is used in restoring the state of the updated data items, which is accomplished with the use of the undo and redo operations. To reduce the overhead in searching the log after a system failure has occurred, we can use a checkpoint scheme.

When several transactions overlap their execution, the resulting execution may no longer be equivalent to an execution where these transactions executed atomically. To ensure correct execution, we must use a concurrency-control scheme to guarantee serializability. There are various different concurrency-control schemes that ensure serializability by either delaying an operation or aborting the transaction that issued the operation. The most common ones are locking protocols and timestamp-ordering schemes.

■ Exercises

- 7.1 What is the meaning of the term *busy waiting*? What other kinds of waiting are there in an operating system? Can busy waiting be avoided altogether? Explain your answer.
- 7.2 Explain why spinlocks are not appropriate for uniprocessor systems yet may be suitable for multiprocessor systems.
- 7.3 Prove that, in the bakery algorithm (Section 7.2.2), the following property holds: If P_i is in its critical section and P_k ($k \neq i$) has already chosen its number $[k] \neq 0$, then $(\text{number}[i], i) < (\text{number}[k], k)$.
- 7.4 The first known correct software solution to the critical-section problem for two processes was developed by Dekker. The two processes, P_0 and P_1 , share the following variables:

```
boolean flag[2]; /* initially false */
int turn;
```

The structure of process P_i ($i == 0$ or 1), with P_j ($j == 1$ or 0) being the other process, is shown in Figure 7.27.

Prove that the algorithm satisfies all three requirements for the critical-section problem.

- 7.5 The first known correct software solution to the critical-section problem for n processes with a lower bound on waiting of $n - 1$ turns was presented by Eisenberg and McGuire. The processes share the following variables:

```
enum pstate {idle, want_in, in_cs};
pstate flag[n];
int turn;
```

```

do {
    flag[i] = true;
    while (flag[j]) {
        if (turn == j) {
            flag[i] = false;
            while (turn == j);
            flag[i] = true;
        }
    }
}

critical section

turn = j;
flag[i] = false;

remainder section

} while (1);

```

Figure 7.27 The structure of process P_i in Dekker's algorithm.

All the elements of `flag` are initially `idle`; the initial value of `turn` is immaterial (between 0 and $n-1$). The structure of process P_i is shown in Figure 7.28.

Prove that the algorithm satisfies all three requirements for the critical-section problem.

- 7.6 In Section 7.3, we mentioned that disabling interrupts frequently can affect the system's clock. Explain why it can, and how such effects can be minimized.
- 7.7 Show that, if the `wait` and `signal` operations are not executed atomically, then mutual exclusion may be violated.
- 7.8 **The Sleeping-Barber Problem.** A barbershop consists of a waiting room with n chairs and the barber room containing the barber chair. If there are no customers to be served, the barber goes to sleep. If a customer enters the barbershop and all chairs are occupied, then the customer leaves the shop. If the barber is busy but chairs are available, then the customer sits in one of the free chairs. If the barber is asleep, the customer wakes up the barber. Write a program to coordinate the barber and the customers.
- 7.9 **The Cigarette-Smokers Problem.** Consider a system with three *smoker* processes and one *agent* process. Each smoker continuously rolls a cigarette and then smokes it. But to roll and smoke a cigarette, the smoker

```

do {
    while(1) {
        flag[i] = want_in;
        j = turn;
        while (j != i) {
            if (flag[j] != idle)
                j = turn;
            else
                j = (j+1) % n;
        }
        flag[i] = in_cs;
        j = 0;
        while ((j < n) && (j == i || flag[j] != in_cs))
            j++;
        if ((j >= n) && (turn == i || flag[turn] == idle)) break;
    }
    turn = i;
}

```

critical section

```

j = (turn+1) % n;
while (flag[j] == idle)
    j = (j+1) % n;
turn = j;
flag[i] = idle;

```

remainder section

```

} while (1);

```

Figure 7.28 The structure of process P_i in Eisenberg and McGuire's algorithm.

needs three ingredients: tobacco, paper, and matches. One of the smoker processes has paper, another has tobacco, and the third has matches. The agent has an infinite supply of all three materials. The agent places two of the ingredients on the table. The smoker who has the remaining ingredient then makes and smokes a cigarette, signaling the agent on completion. The agent then puts out another two of the three ingredients, and the cycle repeats. Write a program to synchronize the agent and the smokers.

- 7.10** Demonstrate that monitors, conditional critical regions, and semaphores are all equivalent, insofar as the same types of synchronization problems can be implemented with them.

- 7.11 Write a bounded-buffer monitor in which the buffers (portions) are embedded within the monitor itself.
- 7.12 The strict mutual exclusion within a monitor makes the bounded-buffer monitor of Exercise 7.11 mainly suitable for small portions.
- Explain why this assertion is *true*.
 - Design a new scheme that is suitable for larger portions.
- 7.13 Suppose that the `signal` statement can appear as only the last statement in a monitor procedure. Suggest how the implementation described in Section 7.7 can be simplified.
- 7.14 Consider a system consisting of processes P_1, P_2, \dots, P_n , each of which has a unique priority number. Write a monitor that allocates three identical line printers to these processes, using the priority numbers for deciding the order of allocation.
- 7.15 A file is to be shared among different processes, each of which has a unique number. The file can be accessed simultaneously by several processes, subject to the following constraint: The sum of all unique numbers associated with all the processes currently accessing the file must be less than n . Write a monitor to coordinate access to the file.
- 7.16 Suppose that we replace the `wait` and `signal` operations of monitors with a single construct `await(B)`, where B is a general Boolean expression that causes the process executing it to wait until B becomes `true`.
- Write a monitor using this scheme to implement the readers-writers problem.
 - Explain why, in general, this construct cannot be implemented efficiently.
 - What restrictions need to be put on the `await` statement so that it can be implemented efficiently? (Hint: Restrict the generality of B ; see Kessels [1977].)
- 7.17 Write a monitor that implements an *alarm clock* that enables a calling program to delay itself for a specified number of time units (*ticks*). You may assume the existence of a real hardware clock that invokes a procedure `tick` in your monitor at regular intervals.
- 7.18 Why does Solaris 2 implement multiple locking mechanisms? Under what circumstances does it use spinlocks, semaphores, adaptive mutexes, conditional variables, and readers-writers locks? Why does it use each mechanism? What is the purpose of turnstiles?

- 7.19 Why do Solaris 2 and Windows 2000 use spinlocks as a synchronization mechanism on only multiprocessor systems and not on uniprocessor systems?
- 7.20 Explain the differences, in terms of cost, among the three storage types: volatile, nonvolatile, and stable.
- 7.21 Explain the purpose of the checkpoint mechanism. How often should checkpoints be performed? How does the frequency of checkpoints affect:
- System performance when no failure occurs?
 - The time it takes to recover from a system crash?
 - The time it takes to recover from a disk crash?
- 7.22 Explain the concept of transaction atomicity.
- 7.23 Show that the two-phase locking protocol ensures conflict serializability.
- 7.24 Show that some schedules are possible under the two-phase locking protocol but not possible under the timestamp protocol, and vice versa.

Bibliographical Notes

The mutual-exclusion algorithms 1 and 2 for two processes were first discussed in the classic paper by Dijkstra [1965a]. Dekker's algorithm (Exercise 6.3)—the first correct software solution to the two-process mutual-exclusion problem—was developed by the Dutch mathematician T. Dekker. This algorithm also was discussed by Dijkstra [1965a]. A simpler solution to the two-process mutual-exclusion problem has since been presented by Peterson [1981] (algorithm 3).

Dijkstra [1965b] presented the first solution to the mutual-exclusion problem for n processes. This solution, however does not have an upper bound on the amount of time a process must wait before it is allowed to enter the critical section. Knuth [1966] presented the first algorithm with a bound; his bound was 2^n turns. A refinement of Knuth's algorithm by deBruijn [1967] reduced the waiting time to n^2 turns, after which Eisenberg and McGuire [1972] (Exercise 6.4) succeeded in reducing the time to the lower bound of $n-1$ turns. The bakery algorithm (algorithm 5) was developed by Lamport [1974]; it also requires $n-1$ turns, but it is easier to program and to understand. Burns [1978] developed the hardware-solution algorithm that satisfies the bounded waiting requirement.

General discussions concerning the mutual-exclusion problem were offered by Lamport [1986] and Lamport [1991]. A collection of algorithms for mutual exclusion was given by Raynal [1986].

The semaphore concept was suggested by Dijkstra [1965a]. Patil [1971] examined the question of whether semaphores can solve all possible synchronization problems. Parnas [1975] discussed some of the flaws in Patil's arguments. Kosaraju [1973] followed up on Patil's work to produce a problem that cannot be solved by `wait` and `signal` operations. Lipton [1974] discussed the limitations of various synchronization primitives.

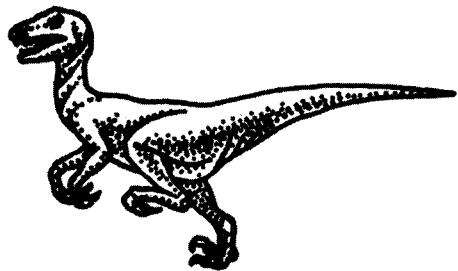
The classic process-coordination problems that we have described are paradigms for a large class of concurrency-control problems. The bounded-buffer problem, the dining-philosophers problem, and the sleeping-barber problem (Exercise 6.7) were suggested by Dijkstra [1965a] and Dijkstra [1971]. The cigarette-smokers problem (Exercise 6.8) was developed by Patil [1971]. The readers-writers problem was suggested by Courtois et al. [1971]. The issue of concurrent reading and writing was discussed by Lamport [1977]. The problem of synchronization of independent processes was discussed by Lamport [1976].

The critical-region concept was suggested by Hoare [1972] and by Brinch-Hansen [1972]. The monitor concept was developed by Brinch-Hansen [1973]. A complete description of the monitor was given by Hoare [1974]. Kessels [1977] proposed an extension to the monitor to allow automatic signaling. General discussions concerning concurrent programming were offered by Ben-Ari [1990].

Some details of the locking mechanisms used in Solaris 2 are presented in Mauro and McDougall [2001]. Note that the locking mechanisms used by the kernel are implemented for user-level threads as well, so the same types of locks are available inside and outside of the kernel. Details of the Windows 2000 synchronization can be found in Solomon and Russinovich [2000].

The write-ahead log scheme was first introduced in System R by Gray et al. [1981]. The concept of serializability was formulated by Eswaran et al. [1976] in connection with their work on concurrency control for System R. The two-phase locking protocol was introduced by Eswaran et al. [1976]. The timestamp-based concurrency-control scheme was provided by Reed [1983]. An exposition of various timestamp-based concurrency-control algorithms was presented by Bernstein and Goodman [1980].

Chapter 8



DEADLOCKS

In a multiprogramming environment, several processes may compete for a finite number of resources. A process requests resources; if the resources are not available at that time, the process enters a wait state. Waiting processes may never again change state, because the resources they have requested are held by other waiting processes. This situation is called a **deadlock**. We have already discussed this issue briefly in Chapter 7, in connection with semaphores.

Perhaps the best illustration of a deadlock can be drawn from a law passed by the Kansas legislature early in the 20th century. It said, in part: "When two trains approach each other at a crossing, both shall come to a full stop and neither shall start up again until the other has gone."

In this chapter, we describe methods that an operating system can use to prevent or deal with deadlocks. Most current operating systems do not provide deadlock-prevention facilities, but such features will probably be added soon. Deadlock problems can only become more common, given current trends, including larger numbers of processes, multithreaded programs, many more resources within a system, and the emphasis on long-lived file and database servers rather than batch systems.

8.1 ■ System Model

A system consists of a finite number of resources to be distributed among a number of competing processes. The resources are partitioned into several types, each of which consists of some number of identical instances. Memory

space, CPU cycles, files, and I/O devices (such as printers and tape drives) are examples of resource types. If a system has two CPUs, then the resource type *CPU* has two instances. Similarly, the resource type *printer* may have five instances.

If a process requests an instance of a resource type, the allocation of *any* instance of the type will satisfy the request. If it will not, then the instances are not identical, and the resource type classes have not been defined properly. For example, a system may have two printers. These two printers may be defined to be in the same resource class if no one cares which printer prints which output. However, if one printer is on the ninth floor and the other is in the basement, then people on the ninth floor may not see both printers as equivalent, and separate resource classes may need to be defined for each printer.

A process must request a resource before using it, and must release the resource after using it. A process may request as many resources as it requires to carry out its designated task. Obviously, the number of resources requested may not exceed the total number of resources available in the system. In other words, a process cannot request three printers if the system has only two.

Under the normal mode of operation, a process may utilize a resource in only the following sequence:

1. **Request:** If the request cannot be granted immediately (for example, the resource is being used by another process), then the requesting process must wait until it can acquire the resource.
2. **Use:** The process can operate on the resource (for example, if the resource is a printer, the process can print on the printer).
3. **Release:** The process releases the resource.

The request and release of resources are system calls, as explained in Chapter 3. Examples are the `request` and `release` device, open and close file, and `allocate` and `free` memory system calls. Request and release of other resources can be accomplished through the *wait* and *signal* operations on semaphores. Therefore, for each use, the operating system checks to make sure that the using process has requested and been allocated the resource. A system table records whether each resource is free or allocated, and, if a resource is allocated, to which process. If a process requests a resource that is currently allocated to another process, it can be added to a queue of processes waiting for this resource.

A set of processes is in a deadlock state when every process in the set is waiting for an event that can be caused only by another process in the set. The events with which we are mainly concerned here are resource acquisition and release. The resources may be either physical resources (for example, printers, tape drives, memory space, and CPU cycles) or logical resources (for example,

files, semaphores, and monitors). However, other types of events may result in deadlocks (for example, the IPC facilities discussed in Chapter 4).

To illustrate a deadlock state, we consider a system with three tape drives. Suppose each of three processes holds one of these tape drives. If each process now requests another tape drive, the three processes will be in a deadlock state. Each is waiting for the event “tape drive is released,” which can be caused only by one of the other waiting processes. This example illustrates a deadlock involving the same resource type.

Deadlocks may also involve different resource types. For example, consider a system with one printer and one tape drive. Suppose that process P_i is holding the tape drive and process P_j is holding the printer. If P_i requests the printer and P_j requests the tape drive, a deadlock occurs.

A programmer who is developing multithreaded applications must pay particular attention to this problem: Multithreaded programs are good candidates for deadlock because multiple threads can compete for shared resources.

8.2 ■ Deadlock Characterization

In a deadlock, processes never finish executing and system resources are tied up, preventing other jobs from starting. Before we discuss the various methods for dealing with the deadlock problem, we shall describe features that characterize deadlocks.

8.2.1 Necessary Conditions

A deadlock situation can arise if the following four conditions hold simultaneously in a system:

1. **Mutual exclusion:** At least one resource must be held in a nonshareable mode; that is, only one process at a time can use the resource. If another process requests that resource, the requesting process must be delayed until the resource has been released.
2. **Hold and wait:** A process must be holding at least one resource and waiting to acquire additional resources that are currently being held by other processes.
3. **No preemption:** Resources cannot be preempted; that is, a resource can be released only voluntarily by the process holding it, after that process has completed its task.
4. **Circular wait:** A set $\{P_0, P_1, \dots, P_n\}$ of waiting processes must exist such that P_0 is waiting for a resource that is held by P_1 , P_1 is waiting for a resource that is held by P_2 , ..., P_{n-1} is waiting for a resource that is held by P_n , and P_n is waiting for a resource that is held by P_0 .

We emphasize that all four conditions must hold for a deadlock to occur. The circular-wait condition implies the hold-and-wait condition, so the four conditions are not completely independent. We shall see in Section 8.4, however, that it is useful to consider each condition separately.

8.2.2 Resource-Allocation Graph

Deadlocks can be described more precisely in terms of a directed graph called a **system resource-allocation graph**. This graph consists of a set of vertices V and a set of edges E . The set of vertices V is partitioned into two different types of nodes $P = \{P_1, P_2, \dots, P_n\}$, the set consisting of all the active processes in the system, and $R = \{R_1, R_2, \dots, R_m\}$, the set consisting of all resource types in the system.

A directed edge from process P_i to resource type R_j is denoted by $P_i \rightarrow R_j$; it signifies that process P_i requested an instance of resource type R_j and is currently waiting for that resource. A directed edge from resource type R_j to process P_i is denoted by $R_j \rightarrow P_i$; it signifies that an instance of resource type R_j has been allocated to process P_i . A directed edge $P_i \rightarrow R_j$ is called a **request edge**; a directed edge $R_j \rightarrow P_i$ is called an **assignment edge**.

Pictorially, we represent each process P_i as a circle, and each resource type R_j as a square. Since resource type R_j may have more than one instance, we represent each such instance as a dot within the square. Note that a request edge points to only the square R_j , whereas an assignment edge must also designate one of the dots in the square.

When process P_i requests an instance of resource type R_j , a request edge is inserted in the resource-allocation graph. When this request can be fulfilled, the request edge is *instantaneously* transformed to an assignment edge. When the process no longer needs access to the resource it releases the resource, and as a result the assignment edge is deleted.

The resource-allocation graph shown in Figure 8.1 depicts the following situation.

- The sets P , R , and E :
 - $P = \{P_1, P_2, P_3\}$
 - $R = \{R_1, R_2, R_3, R_4\}$
 - $E = \{P_1 \rightarrow R_1, P_2 \rightarrow R_3, R_1 \rightarrow P_2, R_2 \rightarrow P_2, R_2 \rightarrow P_1, R_3 \rightarrow P_3\}$
- Resource instances:
 - One instance of resource type R_1
 - Two instances of resource type R_2
 - One instance of resource type R_3
 - Three instances of resource type R_4

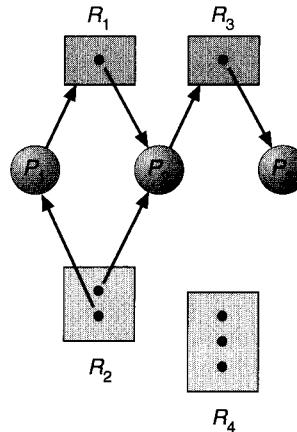


Figure 8.1 Resource-allocation graph.

- Process states:

- Process P_1 is holding an instance of resource type R_2 , and is waiting for an instance of resource type R_1 .
- Process P_2 is holding instances of R_1 and R_2 , and is waiting for an instance of resource type R_3 .
- Process P_3 is holding an instance of R_3 .

Given the definition of a resource-allocation graph, it can be shown that, if the graph contains no cycles, then no process in the system is deadlocked. If the graph does contain a cycle, then a deadlock may exist.

If each resource type has exactly one instance, then a cycle implies that a deadlock has occurred. If the cycle involves only a set of resource types, each of which has only a single instance, then a deadlock has occurred. Each process involved in the cycle is deadlocked. In this case, a cycle in the graph is both a necessary and a sufficient condition for the existence of deadlock.

If each resource type has several instances, then a cycle does not necessarily imply that a deadlock has occurred. In this case, a cycle in the graph is a necessary but not a sufficient condition for the existence of deadlock.

To illustrate this concept, let us return to the resource-allocation graph depicted in Figure 8.1. Suppose that process P_3 requests an instance of resource type R_2 . Since no resource instance is currently available, a request edge $P_3 \rightarrow R_2$ is added to the graph (Figure 8.2). At this point, two minimal cycles exist in the system:

$$\begin{aligned} P_1 &\rightarrow R_1 \rightarrow P_2 \rightarrow R_3 \rightarrow P_3 \rightarrow R_2 \rightarrow P_1 \\ P_2 &\rightarrow R_3 \rightarrow P_3 \rightarrow R_2 \rightarrow P_2 \end{aligned}$$

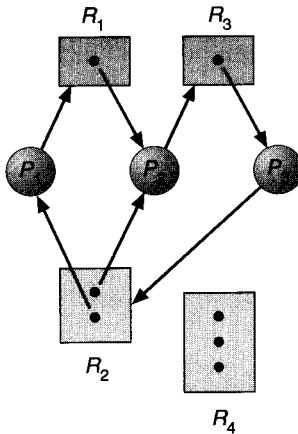


Figure 8.2 Resource-allocation graph with a deadlock.

Processes P_1 , P_2 , and P_3 are deadlocked. Process P_2 is waiting for the resource R_3 , which is held by process P_3 . Process P_3 , on the other hand, is waiting for either process P_1 or process P_2 to release resource R_2 . In addition, process P_1 is waiting for process P_2 to release resource R_1 .

Now consider the resource-allocation graph in Figure 8.3. In this example, we also have a cycle

$$P_1 \rightarrow R_1 \rightarrow P_3 \rightarrow R_2 \rightarrow P_1$$

However, there is no deadlock. Observe that process P_4 may release its instance of resource type R_2 . That resource can then be allocated to P_3 , breaking the cycle.

In summary, if a resource-allocation graph does not have a cycle, then the system is *not* in a deadlock state. On the other hand, if there is a cycle, then the system may or may not be in a deadlock state. This observation is important when we deal with the deadlock problem.

8.3 ■ Methods for Handling Deadlocks

Principally, we can deal with the deadlock problem in one of three ways:

- We can use a protocol to prevent or avoid deadlocks, ensuring that the system will *never* enter a deadlock state.
- We can allow the system to enter a deadlock state, detect it, and recover.

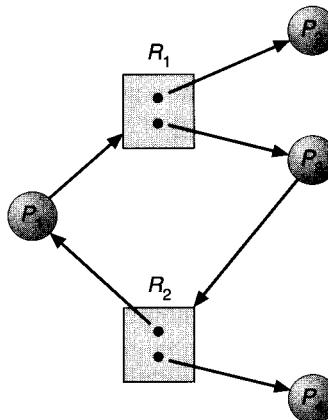


Figure 8.3 Resource-allocation graph with a cycle but no deadlock.

- We can ignore the problem altogether, and pretend that deadlocks never occur in the system. This solution is used by most operating systems, including UNIX.

We shall elaborate briefly on each method. Then, in Sections 8.4 to 8.7, we shall present detailed algorithms.

To ensure that deadlocks never occur, the system can use either a deadlock-prevention or a deadlock-avoidance scheme. **Deadlock prevention** is a set of methods for ensuring that at least one of the necessary conditions (Section 8.2.1) cannot hold. These methods prevent deadlocks by constraining how requests for resources can be made. We discuss these methods in Section 8.4.

Deadlock avoidance, on the other hand, requires that the operating system be given in advance additional information concerning which resources a process will request and use during its lifetime. With this additional knowledge, we can decide for each request whether or not the process should wait. To decide whether the current request can be satisfied or must be delayed, the system must consider the resources currently available, the resources currently allocated to each process, and the future requests and releases of each process. We discuss these schemes in Section 8.5.

If a system does not employ either a deadlock-prevention or a deadlock-avoidance algorithm, then a deadlock situation may occur. In this environment, the system can provide an algorithm that examines the state of the system to determine whether a deadlock has occurred, and an algorithm to recover from the deadlock (if a deadlock has indeed occurred). We discuss these issues in Section 8.6 and Section 8.7.

If a system does not ensure that a deadlock will never occur, and also does not provide a mechanism for deadlock detection and recovery, then we may arrive at a situation where the system is in a deadlock state yet has no way

of recognizing what has happened. In this case, the undetected deadlock will result in the deterioration of the system performance, because resources are being held by processes that cannot run, and because more and more processes, as they make requests for resources, enter a deadlock state. Eventually, the system will stop functioning and will need to be restarted manually.

Although this method does not seem to be a viable approach to the deadlock problem, it is nevertheless used in some operating systems. In many systems, deadlocks occur infrequently (say, once per year); thus, this method is cheaper than the costly deadlock-prevention, deadlock-avoidance, or deadlock-detection and recovery methods that must be used constantly. Also, in some circumstances, the system is in a frozen state but not in a deadlock state. As an example, consider a real-time process running at the highest priority (or any process running on a non-preemptive scheduler) and never returning control to the operating system. Thus, systems must have manual recovery methods for non-deadlock conditions, and may simply use those techniques for deadlock recovery.

8.4 ■ Deadlock Prevention

As we noted in Section 8.2.1, for a deadlock to occur, each of the four necessary conditions must hold. By ensuring that at least one of these conditions cannot hold, we can *prevent* the occurrence of a deadlock. Let us elaborate on this approach by examining each of the four necessary conditions separately.

8.4.1 Mutual Exclusion

The mutual-exclusion condition must hold for nonsharable resources. For example, a printer cannot be simultaneously shared by several processes. Sharable resources, on the other hand, do not require mutually exclusive access, and thus cannot be involved in a deadlock. Read-only files are a good example of a sharable resource. If several processes attempt to open a read-only file at the same time, they can be granted simultaneous access to the file. A process never needs to wait for a sharable resource. In general, however, we cannot prevent deadlocks by denying the mutual-exclusion condition: Some resources are intrinsically nonsharable.

8.4.2 Hold and Wait

To ensure that the hold-and-wait condition never occurs in the system, we must guarantee that, whenever a process requests a resource, it does not hold any other resources. One protocol that can be used requires each process to request and be allocated all its resources before it begins execution. We can implement

this provision by requiring that system calls requesting resources for a process precede all other system calls.

An alternative protocol allows a process to request resources only when the process has none. A process may request some resources and use them. Before it can request any additional resources, however, it must release all the resources that it is currently allocated.

To illustrate the difference between these two protocols, we consider a process that copies data from a tape drive to a disk file, sorts the disk file, and then prints the results to a printer. If all resources must be requested at the beginning of the process, then the process must initially request the tape drive, disk file, and printer. It will hold the printer for its entire execution, even though it needs the printer only at the end.

The second method allows the process to request initially only the tape drive and disk file. It copies from the tape drive to the disk, then releases both the tape drive and the disk file. The process must then again request the disk file and the printer. After copying the disk file to the printer, it releases these two resources and terminates.

These protocols have two main disadvantages. First, **resource utilization** may be low, since many of the resources may be allocated but unused for a long period. In the example given, for instance, we can release the tape drive and disk file, and then again request the disk file and printer, only if we can be sure that our data will remain on the disk file. If we cannot be assured that they will, then we must request all resources at the beginning for both protocols.

Second, starvation is possible. A process that needs several popular resources may have to wait indefinitely, because at least one of the resources that it needs is always allocated to some other process.

8.4.3 No Preemption

The third necessary condition is that there be no preemption of resources that have already been allocated. To ensure that this condition does not hold, we can use the following protocol. If a process is holding some resources and requests another resource that cannot be immediately allocated to it (that is, the process must wait), then all resources currently being held are preempted. In other words, these resources are implicitly released. The preempted resources are added to the list of resources for which the process is waiting. The process will be restarted only when it can regain its old resources, as well as the new ones that it is requesting.

Alternatively, if a process requests some resources, we first check whether they are available. If they are, we allocate them. If they are not available, we check whether they are allocated to some other process that is waiting for additional resources. If so, we preempt the desired resources from the waiting process and allocate them to the requesting process. If the resources are not either available or held by a waiting process, the requesting process must wait.

While it is waiting, some of its resources may be preempted, but only if another process requests them. A process can be restarted only when it is allocated the new resources it is requesting and recovers any resources that were preempted while it was waiting.

This protocol is often applied to resources whose state can be easily saved and restored later, such as CPU registers and memory space. It cannot generally be applied to such resources as printers and tape drives.

8.4.4 Circular Wait

The fourth and final condition for deadlocks is the circular-wait condition. One way to ensure that this condition never holds is to impose a total ordering of all resource types, and to require that each process requests resources in an increasing order of enumeration.

Let $R = \{R_1, R_2, \dots, R_m\}$ be the set of resource types. We assign to each resource type a unique integer number, which allows us to compare two resources and to determine whether one precedes another in our ordering. Formally, we define a one-to-one function $F: R \rightarrow N$, where N is the set of natural numbers. For example, if the set of resource types R includes tape drives, disk drives, and printers, then the function F might be defined as follows:

$$\begin{aligned} F(\text{tape drive}) &= 1, \\ F(\text{disk drive}) &= 5, \\ F(\text{printer}) &= 12. \end{aligned}$$

We can now consider the following protocol to prevent deadlocks: Each process can request resources only in an increasing order of enumeration. That is, a process can initially request any number of instances of a resource type, say R_i . After that, the process can request instances of resource type R_j if and only if $F(R_j) > F(R_i)$. If several instances of the same resource type are needed, a *single* request for all of them must be issued. For example, using the function defined previously, a process that wants to use the tape drive and printer at the same time must first request the tape drive and then request the printer.

Alternatively, we can require that, whenever a process requests an instance of resource type R_j , it has released any resources R_i such that $F(R_i) \geq F(R_j)$.

If these two protocols are used, then the circular-wait condition cannot hold. We can demonstrate this fact by assuming that a circular wait exists (proof by contradiction). Let the set of processes involved in the circular wait be $\{P_0, P_1, \dots, P_n\}$, where P_i is waiting for a resource R_i , which is held by process P_{i+1} . (Modulo arithmetic is used on the indexes, so that P_n is waiting for a resource R_n held by P_0 .) Then, since process P_{i+1} is holding resource R_i while requesting resource R_{i+1} , we must have $F(R_i) < F(R_{i+1})$, for all i . But this condition means that $F(R_0) < F(R_1) < \dots < F(R_n) < F(R_0)$. By transitivity, $F(R_0) < F(R_0)$, which is impossible. Therefore, there can be no circular wait.

Note that the function F should be defined according to the normal order of usage of the resources in a system. For example, since the tape drive is usually needed before the printer, it would be reasonable to define $F(\text{tape drive}) < F(\text{printer})$.

8.5 ■ Deadlock Avoidance

Deadlock-prevention algorithms, as discussed in Section 8.4, prevent deadlocks by restraining how requests can be made. The restraints ensure that at least one of the necessary conditions for deadlock cannot occur, and, hence, that deadlocks cannot hold. Possible side effects of preventing deadlocks by this method, however, are low device utilization and reduced system throughput.

An alternative method for avoiding deadlocks is to require additional information about how resources are to be requested. For example, in a system with one tape drive and one printer, we might be told that process P will request first the tape drive, and later the printer, before releasing both resources. Process Q , on the other hand, will request first the printer, and then the tape drive. With this knowledge of the complete sequence of requests and releases for each process, we can decide for each request whether or not the process should wait. Each request requires that the system consider the resources currently available, the resources currently allocated to each process, and the future requests and releases of each process, to decide whether the current request can be satisfied or must wait to avoid a possible future deadlock.

The various algorithms differ in the amount and type of information required. The simplest and most useful model requires that each process declare the *maximum number* of resources of each type that it may need. Given a priori information about the maximum number of resources of each type that may be requested for each process, it is possible to construct an algorithm that ensures that the system will never enter a deadlock state. This algorithm defines the **deadlock-avoidance** approach. A deadlock-avoidance algorithm dynamically examines the resource-allocation state to ensure that a circular-wait condition can never exist. The resource-allocation *state* is defined by the number of available and allocated resources, and the maximum demands of the processes.

8.5.1 Safe State

A state is *safe* if the system can allocate resources to each process (up to its maximum) in some order and still avoid a deadlock. More formally, a system is in a safe state only if there exists a **safe sequence**. A sequence of processes $\langle P_1, P_2, \dots, P_n \rangle$ is a safe sequence for the current allocation state if, for each P_i , the resources that P_i can still request can be satisfied by the currently available resources plus the resources held by all the P_j , with $j < i$. In this situation, if

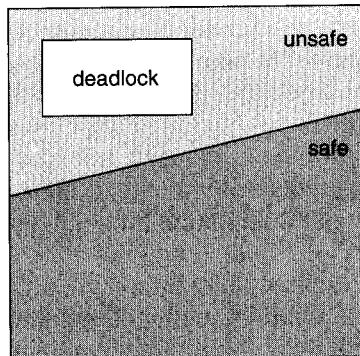


Figure 8.4 Safe, unsafe, and deadlock state spaces.

the resources that process P_i needs are not immediately available, then P_i can wait until all P_j have finished. When they have finished, P_i can obtain all of its needed resources, complete its designated task, return its allocated resources, and terminate. When P_i terminates, P_{i+1} can obtain its needed resources, and so on. If no such sequence exists, then the system state is said to be *unsafe*.

A safe state is not a deadlock state. Conversely, a deadlock state is an unsafe state. Not all unsafe states are deadlocks, however (Figure 8.4). An unsafe state *may* lead to a deadlock. As long as the state is safe, the operating system can avoid unsafe (and deadlock) states. In an unsafe state, the operating system cannot prevent processes from requesting resources such that a deadlock occurs: The behavior of the processes controls unsafe states.

To illustrate, we consider a system with 12 magnetic tape drives and 3 processes: P_0 , P_1 , and P_2 . Process P_0 requires 10 tape drives, process P_1 may need as many as 4, and process P_2 may need up to 9 tape drives. Suppose that, at time t_0 , process P_0 is holding 5 tape drives, process P_1 is holding 2, and process P_2 is holding 2 tape drives. (Thus, there are 3 free tape drives.)

	Maximum Needs	Current Needs
P_0	10	5
P_1	4	2
P_2	9	2

At time t_0 , the system is in a safe state. The sequence $\langle P_1, P_0, P_2 \rangle$ satisfies the safety condition, since process P_1 can immediately be allocated all its tape drives and then return them (the system will then have 5 available tape drives), then process P_0 can get all its tape drives and return them (the system will then have 10 available tape drives), and finally process P_2 could get all its tape drives and return them (the system will then have all 12 tape drives available).

A system may go from a safe state to an unsafe state. Suppose that, at time t_1 , process P_2 requests and is allocated 1 more tape drive. The system is no longer in a safe state. At this point, only process P_1 can be allocated all its tape drives. When it returns them, the system will have only 4 available tape drives. Since process P_0 is allocated 5 tape drives, but has a maximum of 10, it may then request 5 more tape drives. Since they are unavailable, process P_0 must wait. Similarly, process P_2 may request an additional 6 tape drives and have to wait, resulting in a deadlock.

Our mistake was in granting the request from process P_2 for 1 more tape drive. If we had made P_2 wait until either of the other processes had finished and released its resources, then we could have avoided the deadlock.

Given the concept of a safe state, we can define avoidance algorithms that ensure that the system will never deadlock. The idea is simply to ensure that the system will always remain in a safe state. Initially, the system is in a safe state. Whenever a process requests a resource that is currently available, the system must decide whether the resource can be allocated immediately or whether the process must wait. The request is granted only if the allocation leaves the system in a safe state.

In this scheme, if a process requests a resource that is currently available, it may still have to wait. Thus, resource utilization may be lower than it would be without a deadlock-avoidance algorithm.

8.5.2 Resource-Allocation Graph Algorithm

If we have a resource-allocation system with only one instance of each resource type, a variant of the resource-allocation graph defined in Section 8.2.2 can be used for deadlock avoidance.

In addition to the request and assignment edges, we introduce a new type of edge, called a **claim edge**. A claim edge $P_i \rightarrow R_j$ indicates that process P_i may request resource R_j at some time in the future. This edge resembles a request edge in direction, but is represented by a dashed line. When process P_i requests resource R_j , the claim edge $P_i \rightarrow R_j$ is converted to a request edge. Similarly, when a resource R_j is released by P_i , the assignment edge $R_j \rightarrow P_i$ is reconverted to a claim edge $P_i \rightarrow R_j$. We note that the resources must be claimed a priori in the system. That is, before process P_i starts executing, all its claim edges must already appear in the resource-allocation graph. We can relax this condition by allowing a claim edge $P_i \rightarrow R_j$ to be added to the graph only if all the edges associated with process P_i are claim edges.

Suppose that process P_i requests resource R_j . The request can be granted only if converting the request edge $P_i \rightarrow R_j$ to an assignment edge $R_j \rightarrow P_i$ does not result in the formation of a cycle in the resource-allocation graph. Note that we check for safety by using a cycle-detection algorithm. An algorithm for detecting a cycle in this graph requires an order of n^2 operations, where n is the number of processes in the system.

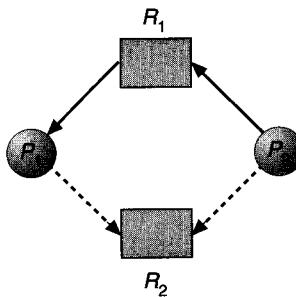


Figure 8.5 Resource-allocation graph for deadlock avoidance.

If no cycle exists, then the allocation of the resource will leave the system in a safe state. If a cycle is found, then the allocation will put the system in an unsafe state. Therefore, process P_i will have to wait for its requests to be satisfied.

To illustrate this algorithm, we consider the resource-allocation graph of Figure 8.5. Suppose that P_2 requests R_2 . Although R_2 is currently free, we cannot allocate it to P_2 , since this action will create a cycle in the graph (Figure 8.6). A cycle indicates that the system is in an unsafe state. If P_1 requests R_2 , and P_2 requests R_1 , then a deadlock will occur.

8.5.3 Banker's Algorithm

The resource-allocation graph algorithm is not applicable to a resource-allocation system with multiple instances of each resource type. The deadlock-avoidance algorithm that we describe next is applicable to such a system, but is less efficient than the resource-allocation graph scheme. This algorithm is commonly known as the *banker's algorithm*. The name was chosen because this algorithm could be used in a banking system to ensure that the

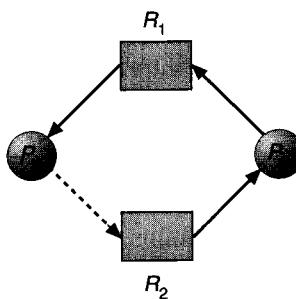


Figure 8.6 An unsafe state in a resource-allocation graph.

bank never allocates its available cash such that it can no longer satisfy the needs of all its customers.

When a new process enters the system, it must declare the maximum number of instances of each resource type that it may need. This number may not exceed the total number of resources in the system. When a user requests a set of resources, the system must determine whether the allocation of these resources will leave the system in a safe state. If it will, the resources are allocated; otherwise, the process must wait until some other process releases enough resources.

Several data structures must be maintained to implement the banker's algorithm. These data structures encode the state of the resource-allocation system. Let n be the number of processes in the system and m be the number of resource types. We need the following data structures:

- **Available:** A vector of length m indicates the number of available resources of each type. If $\text{Available}[j] = k$, there are k instances of resource type R_j available.
- **Max:** An $n \times m$ matrix defines the maximum demand of each process. If $\text{Max}[i,j] = k$, then process P_i may request at most k instances of resource type R_j .
- **Allocation:** An $n \times m$ matrix defines the number of resources of each type currently allocated to each process. If $\text{Allocation}[i,j] = k$, then process P_i is currently allocated k instances of resource type R_j .
- **Need:** An $n \times m$ matrix indicates the remaining resource need of each process. If $\text{Need}[i,j] = k$, then process P_i may need k more instances of resource type R_j to complete its task. Note that $\text{Need}[i,j] = \text{Max}[i,j] - \text{Allocation}[i,j]$.

These data structures vary over time in both size and value.

To simplify the presentation of the banker's algorithm, let us establish some notation. Let X and Y be vectors of length n . We say that $X \leq Y$ if and only if $X[i] \leq Y[i]$ for all $i = 1, 2, \dots, n$. For example, if $X = (1,7,3,2)$ and $Y = (0,3,2,1)$, then $Y \leq X$. $Y < X$ if $Y \leq X$ and $Y \neq X$.

We can treat each row in the matrices *Allocation* and *Need* as vectors and refer to them as Allocation_i and Need_i , respectively. The vector Allocation_i specifies the resources currently allocated to process P_i ; the vector Need_i specifies the additional resources that process P_i may still request to complete its task.

8.5.3.1 Safety Algorithm

The algorithm for finding out whether or not a system is in a safe state can be described as follows:

1. Let $Work$ and $Finish$ be vectors of length m and n , respectively. Initialize $Work := Available$ and $Finish[i] := false$ for $i = 1, 2, \dots, n$.
2. Find an i such that both
 - a. $Finish[i] = false$
 - b. $Need_i \leq Work$.

If no such i exists, go to step 4.

3. $Work := Work + Allocation_i$
 $Finish[i] := true$
 go to step 2.
4. If $Finish[i] = true$ for all i , then the system is in a safe state.

This algorithm may require an order of $m \times n^2$ operations to decide whether a state is safe.

8.5.3.2 Resource-Request Algorithm

Let $Request_i$ be the request vector for process P_i . If $Request_i[j] = k$, then process P_i wants k instances of resource type R_j . When a request for resources is made by process P_i , the following actions are taken:

1. If $Request_i \leq Need_i$, go to step 2. Otherwise, raise an error condition, since the process has exceeded its maximum claim.
2. If $Request_i \leq Available$, go to step 3. Otherwise, P_i must wait, since the resources are not available.
3. Have the system pretend to have allocated the requested resources to process P_i by modifying the state as follows:

```

 $Available := Available - Request_i;$ 
 $Allocation_i := Allocation_i + Request_i;$ 
 $Need_i := Need_i - Request_i;$ 

```

If the resulting resource-allocation state is safe, the transaction is completed and process P_i is allocated its resources. However, if the new state is unsafe, then P_i must wait for $Request_i$ and the old resource-allocation state is restored.

8.5.3.3 An Illustrative Example

Consider a system with five processes P_0 through P_4 and three resource types A, B, C . Resource type A has 10 instances, resource type B has 5 instances, and resource type C has 7 instances. Suppose that, at time T_0 , the following snapshot of the system has been taken:

	<u>Allocation</u>	<u>Max</u>	<u>Available</u>
	<u>$A\ B\ C$</u>	<u>$A\ B\ C$</u>	<u>$A\ B\ C$</u>
P_0	0 1 0	7 5 3	3 3 2
P_1	2 0 0	3 2 2	
P_2	3 0 2	9 0 2	
P_3	2 1 1	2 2 2	
P_4	0 0 2	4 3 3	

The content of the matrix $Need$ is defined to be $Max - Allocation$ and is

	<u>Need</u>
	<u>$A\ B\ C$</u>
P_0	7 4 3
P_1	1 2 2
P_2	6 0 0
P_3	0 1 1
P_4	4 3 1

We claim that the system is currently in a safe state. Indeed, the sequence $\langle P_1, P_3, P_4, P_2, P_0 \rangle$ satisfies the safety criteria. Suppose now that process P_1 requests one additional instance of resource type A and two instances of resource type C , so $Request_1 = (1,0,2)$. To decide whether this request can be immediately granted, we first check that $Request_1 \leq Available$ (that is, $(1,0,2) \leq (3,3,2)$), which is true. We then pretend that this request has been fulfilled, and we arrive at the following new state:

	<u>Allocation</u>	<u>Need</u>	<u>Available</u>
	<u>$A\ B\ C$</u>	<u>$A\ B\ C$</u>	<u>$A\ B\ C$</u>
P_0	0 1 0	7 4 3	2 3 0
P_1	3 0 2	0 2 0	
P_2	3 0 2	6 0 0	
P_3	2 1 1	0 1 1	
P_4	0 0 2	4 3 1	

We must determine whether this new system state is safe. To do so, we execute our safety algorithm and find that the sequence $\langle P_1, P_3, P_4, P_0, P_2 \rangle$

satisfies our safety requirement. Hence, we can immediately grant the request of process P_1 .

You should be able to see, however, that when the system is in this state, a request for $(3,3,0)$ by P_4 cannot be granted, since the resources are not available. A request for $(0,2,0)$ by P_0 cannot be granted, even though the resources are available, since the resulting state is unsafe.

8.6 ■ Deadlock Detection

If a system does not employ either a deadlock-prevention or a deadlock-avoidance algorithm, then a deadlock situation may occur. In this environment, the system must provide:

- An algorithm that examines the state of the system to determine whether a deadlock has occurred
- An algorithm to recover from the deadlock

In the following discussion, we elaborate on these two requirements as they pertain to systems with only a single instance of each resource type, as well as to systems with several instances of each resource type. At this point, however, let us note that a detection-and-recovery scheme requires overhead that includes not only the run-time costs of maintaining the necessary information and executing the detection algorithm, but also the potential losses inherent in recovering from a deadlock.

8.6.1 Single Instance of Each Resource Type

If all resources have only a single instance, then we can define a deadlock-detection algorithm that uses a variant of the resource-allocation graph, called a *wait-for* graph. We obtain this graph from the resource-allocation graph by removing the nodes of type resource and collapsing the appropriate edges.

More precisely, an edge from P_i to P_j in a wait-for graph implies that process P_i is waiting for process P_j to release a resource that P_i needs. An edge $P_i \rightarrow P_j$ exists in a wait-for graph if and only if the corresponding resource-allocation graph contains two edges $P_i \rightarrow R_q$ and $R_q \rightarrow P_j$ for some resource R_q . For example, in Figure 8.7, we present a resource-allocation graph and the corresponding wait-for graph.

As before, a deadlock exists in the system if and only if the wait-for graph contains a cycle. To detect deadlocks, the system needs to *maintain* the wait-for graph and periodically to *invoke an algorithm* that searches for a cycle in the graph.

An algorithm to detect a cycle in a graph requires an order of n^2 operations, where n is the number of vertices in the graph.

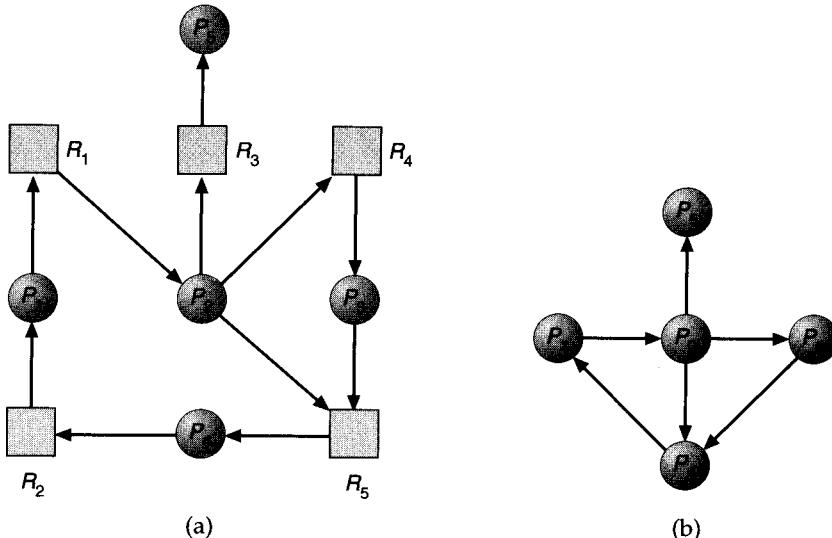


Figure 8.7 (a) Resource-allocation graph. (b) Corresponding wait-for graph.

8.6.2 Several Instances of a Resource Type

The wait-for graph scheme is not applicable to a resource-allocation system with multiple instances of each resource type. The deadlock-detection algorithm that we describe next is applicable to such a system. The algorithm employs several time-varying data structures that are similar to those used in the banker's algorithm (Section 8.5.3):

- **Available:** A vector of length m indicates the number of available resources of each type.
- **Allocation:** An $n \times m$ matrix defines the number of resources of each type currently allocated to each process.
- **Request:** An $n \times m$ matrix indicates the current request of each process. If $\text{Request}[i,j] = k$, then process P_i is requesting k more instances of resource type R_j .

The \leq relation between two vectors is defined as in Section 8.5.3. To simplify notation, we shall again treat the rows in the matrices *Allocation* and *Request* as vectors, and shall refer to them as Allocation_i and Request_i , respectively. The detection algorithm described here simply investigates every possible allocation sequence for the processes that remain to be completed. Compare this algorithm with the banker's algorithm of Section 8.5.3.

1. Let $Work$ and $Finish$ be vectors of length m and n , respectively. Initialize $Work := Available$. For $i = 1, 2, \dots, n$, if $Allocation_i \neq 0$, then $Finish[i] := false$; otherwise, $Finish[i] := true$.
2. Find an index i such that both
 - a. $Finish[i] = false$.
 - b. $Request_i \leq Work$.

If no such i exists, go to step 4.

3. $Work := Work + Allocation_i$
 $Finish[i] := true$
 go to step 2.
4. If $Finish[i] = false$, for some i , $1 \leq i \leq n$, then the system is in a deadlock state. Moreover, if $Finish[i] = false$, then process P_i is deadlocked.

This algorithm requires an order of $m \times n^2$ operations to detect whether the system is in a deadlocked state.

You may wonder why we reclaim the resources of process P_i (in step 3) as soon as we determine that $Request_i \leq Work$ (in step 2b). We know that P_i is currently *not* involved in a deadlock (since $Request_i \leq Work$). Thus, we take an optimistic attitude, and assume that P_i will require no more resources to complete its task; it will thus soon return all currently allocated resources to the system. If our assumption is incorrect, a deadlock may occur later. That deadlock will be detected the next time that the deadlock-detection algorithm is invoked.

To illustrate this algorithm, we consider a system with five processes P_0 through P_4 and three resource types A, B, C . Resource type A has 7 instances, resource type B has 2 instances, and resource type C has 6 instances. Suppose that, at time T_0 , we have the following resource-allocation state:

	<u>Allocation</u>	<u>Request</u>	<u>Available</u>
	<u>$A\ B\ C$</u>	<u>$A\ B\ C$</u>	<u>$A\ B\ C$</u>
P_0	0 1 0	0 0 0	0 0 0
P_1	2 0 0	2 0 2	
P_2	3 0 3	0 0 0	
P_3	2 1 1	1 0 0	
P_4	0 0 2	0 0 2	

We claim that the system is not in a deadlocked state. Indeed, if we execute our algorithm, we will find that the sequence $\langle P_0, P_2, P_3, P_1, P_4 \rangle$ will result in $Finish[i] = true$ for all i .

Suppose now that process P_2 makes one additional request for an instance of type C. The *Request* matrix is modified as follows:

	<i>Request</i>		
	A	B	C
P_0	0	0	0
P_1	2	0	2
P_2	0	0	1
P_3	1	0	0
P_4	0	0	2

We claim that the system is now deadlocked. Although we can reclaim the resources held by process P_0 , the number of available resources is not sufficient to fulfill the requests of the other processes. Thus, a deadlock exists, consisting of processes P_1 , P_2 , P_3 , and P_4 .

8.6.3 Detection-Algorithm Usage

When should we invoke the detection algorithm? The answer depends on two factors:

1. How *often* is a deadlock likely to occur?
2. How *many* processes will be affected by deadlock when it happens?

If deadlocks occur frequently, then the detection algorithm should be invoked frequently. Resources allocated to deadlocked processes will be idle until the deadlock can be broken. In addition, the number of processes involved in the deadlock cycle may grow.

Deadlocks occur only when some process makes a request that cannot be granted immediately. This request may be the final request that completes a chain of waiting processes. In the extreme, we could invoke the deadlock-detection algorithm every time a request for allocation cannot be granted immediately. In this case, we can identify not only the set of processes that is deadlocked, but also the specific process that “caused” the deadlock. (In reality, each of the deadlocked processes is a link in the cycle in the resource graph, so all of them, jointly, caused the deadlock.) If there are many different resource types, one request may cause many cycles in the resource graph, each cycle completed by the most recent request and “caused” by the one identifiable process.

Of course, invoking the deadlock-detection algorithm for every request may incur a considerable overhead in computation time. A less expensive alternative is simply to invoke the algorithm at less frequent intervals—for example, once per hour, or whenever CPU utilization drops below 40 percent. (A deadlock eventually cripples system throughput and will cause CPU utilization to

drop.) If the detection algorithm is invoked at arbitrary points in time, there may be many cycles in the resource graph. We would generally not be able to tell which of the many deadlocked processes “caused” the deadlock.

8.7 ■ Recovery from Deadlock

When a detection algorithm determines that a deadlock exists, several alternatives exist. One possibility is to inform the operator that a deadlock has occurred, and to let the operator deal with the deadlock manually. The other possibility is to let the system *recover* from the deadlock automatically. There are two options for breaking a deadlock. One solution is simply to abort one or more processes to break the circular wait. The second option is to preempt some resources from one or more of the deadlocked processes.

8.7.1 Process Termination

To eliminate deadlocks by aborting a process, we use one of two methods. In both methods, the system reclaims all resources allocated to the terminated processes.

- **Abort all deadlocked processes:** This method clearly will break the deadlock cycle, but at a great expense; these processes may have computed for a long time, and the results of these partial computations must be discarded and probably recomputed later.
- **Abort one process at a time until the deadlock cycle is eliminated:** This method incurs considerable overhead, since, after each process is aborted, a deadlock-detection algorithm must be invoked to determine whether any processes are still deadlocked.

Aborting a process may not be easy. If the process was in the midst of updating a file, terminating it will leave that file in an incorrect state. Similarly, if the process was in the midst of printing data on the printer, the system must reset the printer to a correct state before printing the next job.

If the partial termination method is used, then, given a set of deadlocked processes, we must determine which process (or processes) should be terminated in an attempt to break the deadlock. This determination is a policy decision, similar to CPU-scheduling problems. The question is basically an economic one; we should abort those processes the termination of which will incur the minimum cost. Unfortunately, the term *minimum cost* is not a precise one. Many factors may determine which process is chosen, including:

1. What the priority of the process is

2. How long the process has computed, and how much longer the process will compute before completing its designated task
3. How many and what type of resources the process has used (for example, whether the resources are simple to preempt)
4. How many more resources the process needs in order to complete
5. How many processes will need to be terminated
6. Whether the process is interactive or batch

8.7.2 Resource Preemption

To eliminate deadlocks using resource preemption, we successively preempt some resources from processes and give these resources to other processes until the deadlock cycle is broken.

If preemption is required to deal with deadlocks, then three issues need to be addressed:

1. **Selecting a victim:** Which resources and which processes are to be preempted? As in process termination, we must determine the order of preemption to minimize cost. Cost factors may include such parameters as the number of resources a deadlock process is holding, and the amount of time a deadlocked process has thus far consumed during its execution.
2. **Rollback:** If we preempt a resource from a process, what should be done with that process? Clearly, it cannot continue with its normal execution; it is missing some needed resource. We must roll back the process to some safe state, and restart it from that state.

Since, in general, it is difficult to determine what a safe state is, the simplest solution is a total rollback: Abort the process and then restart it. However, it is more effective to roll back the process only as far as necessary to break the deadlock. On the other hand, this method requires the system to keep more information about the state of all the running processes.

3. **Starvation:** How do we ensure that starvation will not occur? That is, how can we guarantee that resources will not always be preempted from the same process?

In a system where victim selection is based primarily on cost factors, it may happen that the same process is always picked as a victim. As a result, this process never completes its designated task, a *starvation* situation that needs to be dealt with in any practical system. Clearly, we must ensure that a process can be picked as a victim only a (small) finite number of times. The most common solution is to include the number of rollbacks in the cost factor.

8.8 ■ Summary

A deadlock state occurs when two or more processes are waiting indefinitely for an event that can be caused only by one of the waiting processes. Principally, there are three methods for dealing with deadlocks:

- Use some protocol to prevent or avoid deadlocks, ensuring that the system will never enter a deadlock state.
- Allow the system to enter deadlock state, detect it, and then recover.
- Ignore the problem all together, and pretend that deadlocks never occur in the system. This solution is the one used by most operating systems, including UNIX.

A deadlock situation may occur if and only if four necessary conditions hold simultaneously in the system: mutual exclusion, hold and wait, no pre-emption, and circular wait. To prevent deadlocks, we ensure that at least one of the necessary conditions never holds.

Another method for avoiding deadlocks that is less stringent than the prevention algorithms is to have a priori information on how each process will be utilizing the resources. The banker's algorithm, for example, needs to know the maximum number of each resource class that may be requested by each process. Using this information, we can define a deadlock-avoidance algorithm.

If a system does not employ a protocol to ensure that deadlocks will never occur, then a detection-and-recovery scheme must be employed. A deadlock-detection algorithm must be invoked to determine whether a deadlock has occurred. If a deadlock is detected, the system must recover either by terminating some of the deadlocked processes, or by preempting resources from some of the deadlocked processes.

In a system that selects victims for rollback primarily on the basis of cost factors, starvation may occur. As a result, the selected process never completes its designated task.

■ Exercises

- 8.1 List three examples of deadlocks that are not related to a computer-system environment.
- 8.2 Is it possible to have a deadlock involving only one process? Explain your answer.
- 8.3 People have said that proper spooling would eliminate deadlocks. Certainly, it eliminates contention card readers, plotters, printers, and so on. It is even possible to spool tapes (called *staging* them), which would

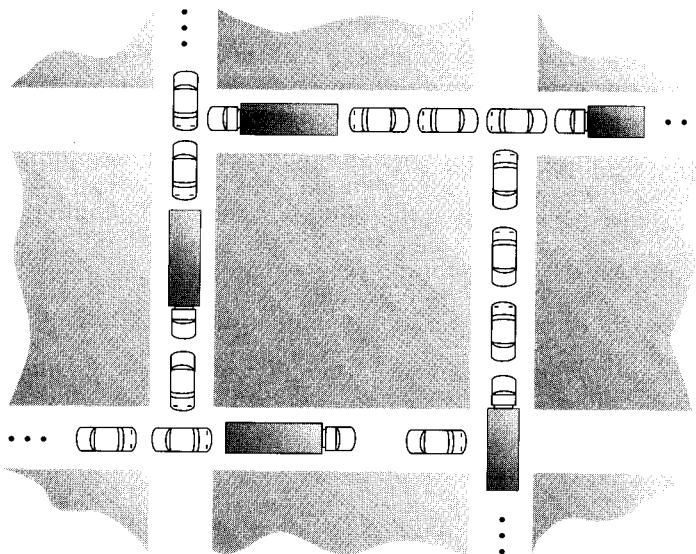


Figure 8.8 Traffic deadlock for Exercise 8.4.

leave the resources of CPU time, memory, and disk space. Is it possible to have a deadlock involving these resources? If it is, how could such a deadlock occur? If it is not, why not? What deadlock scheme would seem best to eliminate these deadlocks (if any are possible), or what condition is violated (if they are not possible)?

- 8.4 Consider the traffic deadlock depicted in Figure 8.8.
- Show that the four necessary conditions for deadlock indeed hold in this example.
 - State a simple rule that will avoid deadlocks in this system.
- 8.5 Suppose that a system is in an unsafe state. Show that it is possible for the processes to complete their execution without entering a deadlock state.
- 8.6 In a real computer system, neither the resources available nor the demands of processes for resources are consistent over long periods (months). Resources break or are replaced, new processes come and go, new resources are bought and added to the system. If deadlock is controlled by the banker's algorithm, which of the following changes can be made safely (without introducing the possibility of deadlock), and under what circumstances?
- Increase *Available* (new resources added)

- b. Decrease *Available* (resource permanently removed from system)
 - c. Increase *Max* for one process (the process needs more resources than allowed, it may want more)
 - d. Decrease *Max* for one process (the process decides it does not need that many resources)
 - e. Increase the number of processes
 - f. Decrease the number of processes
- 8.7 Prove that the safety algorithm presented in Section 8.5.3 requires an order of $m \times n^2$ operations.
- 8.8 Consider a system consisting of four resources of the same type that are shared by three processes, each of which needs at most two resources. Show that the system is deadlock-free.
- 8.9 Consider a system consisting of m resources of the same type, being shared by n processes. Resources can be requested and released by processes only one at a time. Show that the system is deadlock-free if the following two conditions hold:
- a. The maximum need of each process is between 1 and m resources
 - b. The sum of all maximum needs is less than $m + n$
- 8.10 Consider a computer system that runs 5,000 jobs per month with no deadlock-prevention or deadlock-avoidance scheme. Deadlocks occur about twice per month, and the operator must terminate and rerun about 10 jobs per deadlock. Each job is worth about \$2 (in CPU time), and the jobs terminated tend to be about half-done when they are aborted.
A systems programmer has estimated that a deadlock-avoidance algorithm (like the banker's algorithm) could be installed in the system with an increase in the average execution time per job of about 10 percent. Since the machine currently has 30-percent idle time, all 5,000 jobs per month could still be run, although turnaround time would increase by about 20 percent on average.
- a. What are the arguments for installing the deadlock-avoidance algorithm?
 - b. What are the arguments against installing the deadlock-avoidance algorithm?
- 8.11 We can obtain the banker's algorithm for a single resource type from the general banker's algorithm simply by reducing the dimensionality of the various arrays by 1. Show through an example that the multiple-resource-

type banker's scheme cannot be implemented by individual application of the single-resource-type scheme to each resource type.

- 8.12 Can a system detect that some of its processes are starving? If you answer "yes," explain how it can. If you answer "no," explain how the system can deal with the starvation problem.
- 8.13 Consider the following snapshot of a system:

	<u>Allocation</u>	<u>Max</u>	<u>Available</u>
	A B C D	A B C D	A B C D
P_0	0 0 1 2	0 0 1 2	1 5 2 0
P_1	1 0 0 0	1 7 5 0	
P_2	1 3 5 4	2 3 5 6	
P_3	0 6 3 2	0 6 5 2	
P_4	0 0 1 4	0 6 5 6	

Answer the following questions using the banker's algorithm:

- What is the content of the matrix *Need*?
- Is the system in a safe state?
- If a request from process P_1 arrives for $(0,4,2,0)$, can the request be granted immediately?

- 8.14 Consider the following resource-allocation policy. Requests and releases for resources are allowed at any time. If a request for resources cannot be satisfied because the resources are not available, then we check any processes that are blocked, waiting for resources. If they have the desired resources, then these resources are taken away from them and are given to the requesting process. The vector of resources for which the waiting process is waiting is increased to include the resources that were taken away.

For example, consider a system with three resource types and the vector *Available* initialized to $(4,2,2)$. If process P_0 asks for $(2,2,1)$, it gets them. If P_1 asks for $(1,0,1)$, it gets them. Then, if P_0 asks for $(0,0,1)$, it is blocked (resource not available). If P_2 now asks for $(2,0,0)$, it gets the available one $(1,0,0)$ and one that was allocated to P_0 (since P_0 is blocked). P_0 's *Allocation* vector goes down to $(1,2,1)$, and its *Need* vector goes up to $(1,0,1)$.

- Can deadlock occur? If so, give an example. If not, which necessary condition cannot occur?
- Can indefinite blocking occur?

- 8.15 Suppose that you have coded the deadlock-avoidance safety algorithm and now have been asked to implement the deadlock-detection algorithm. Can you do so by simply using the safety algorithm code and redefining $Max_i = Waiting_i + Allocation_i$, where $Waiting_i$ is a vector specifying the resources process i is waiting for, and $Allocation_i$ is as defined in Section 8.5? Explain your answer.

Bibliographical Notes

Dijkstra [1965a] was one of the first and most influential contributors in the deadlock area. Holt [1972] was the first person to formalize the notion of deadlocks in terms of a graph-theoretical model similar to the one presented in this chapter. Starvation was covered by Holt [1972]. Hyman [1985] provided the deadlock example from the Kansas legislature.

The various prevention algorithms were suggested by Havender [1968], who has devised the resource-ordering scheme for the IBM OS/360 system.

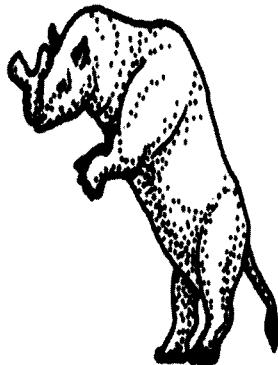
The banker's algorithm for avoiding deadlocks was developed for a single resource type by Dijkstra [1965a], and was extended to multiple resource types by Habermann [1969]. Exercises 8.8 and 8.9 are from Holt [1971].

The deadlock-detection algorithm for multiple instances of a resource type, which was described in Section 8.6.2, was presented by Coffman et al. [1971].

Bach [1987] describes how many of the algorithms in the traditional UNIX kernel handle deadlock.

Part Three

STORAGE MANAGEMENT

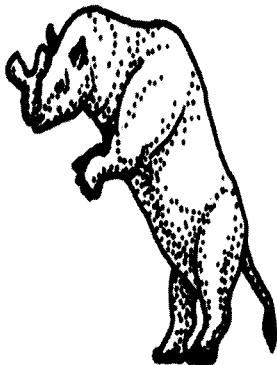


The main purpose of a computer system is to execute programs. These programs, together with the data they access, must be in main memory (at least partially) during execution.

To improve both the utilization of the CPU and the speed of its response to users, the computer must keep several processes in memory. Many memory-management schemes exist, reflecting various approaches, and the effectiveness of each algorithm depends on the situation. Selection of a memory-management scheme for a system depends on many factors, especially on the *hardware* design of the system. Each algorithm requires its own hardware support.

Since main memory is usually too small to accommodate all the data and programs permanently, the computer system must provide secondary storage to back up main memory. Modern computer systems use disks as the primary on-line storage medium for information (both programs and data). The file system provides the mechanism for on-line storage of and access to both data and programs residing on the disks. A file is a collection of related information defined by its creator. The files are mapped by the operating system onto physical devices. Files are normally organized into directories to ease their use.

Chapter 9



MEMORY MANAGEMENT

In Chapter 6, we showed how the CPU can be shared by a set of processes. As a result of CPU scheduling, we can improve both the utilization of the CPU and the speed of the computer's response to its users. To realize this increase in performance, however, we must keep several processes in memory; that is, we must *share* memory.

In this chapter, we discuss various ways to manage memory. The memory-management algorithms vary from a primitive bare-machine approach to paging and segmentation strategies. Each approach has its own advantages and disadvantages. Selection of a memory-management method for a specific system depends on many factors, especially on the *hardware* design of the system. As we shall see, many algorithms require hardware support, although recent designs have closely integrated the hardware and operating system.

9.1 ■ Background

As we saw in Chapter 1, memory is central to the operation of a modern computer system. Memory consists of a large array of words or bytes, each with its own address. The CPU fetches instructions from memory according to the value of the program counter. These instructions may cause additional loading from and storing to specific memory addresses.

A typical instruction-execution cycle, for example, first fetches an instruction from memory. The instruction is then decoded and may cause operands to be fetched from memory. After the instruction has been executed on the

operands, results may be stored back in memory. The memory unit sees only a stream of memory addresses; it does not know how they are generated (by the instruction counter, indexing, indirection, literal addresses, and so on) or what they are for (instructions or data). Accordingly, we can ignore *how* a memory address is generated by a program. We are interested in only the sequence of memory addresses generated by the running program.

9.1.1 Address Binding

Usually, a program resides on a disk as a binary executable file. The program must be brought into memory and placed within a process for it to be executed. Depending on the memory management in use, the process may be moved between disk and memory during its execution. The collection of processes on the disk that is waiting to be brought into memory for execution forms the **input queue**.

The normal procedure is to select one of the processes in the input queue and to load that process into memory. As the process is executed, it accesses instructions and data from memory. Eventually, the process terminates, and its memory space is declared available.

Most systems allow a user process to reside in any part of the physical memory. Thus, although the address space of the computer starts at 00000, the first address of the user process does not need to be 00000. This arrangement affects the addresses that the user program can use. In most cases, a user program will go through several steps—some of which may be optional—before being executed (Figure 9.1). Addresses may be represented in different ways during these steps. Addresses in the source program are generally symbolic (such as *count*). A compiler will typically **bind** these symbolic addresses to relocatable addresses (such as “14 bytes from the beginning of this module”). The linkage editor or loader will in turn bind these relocatable addresses to absolute addresses (such as 74014). Each binding is a mapping from one address space to another.

Classically, the binding of instructions and data to memory addresses can be done at any step along the way:

- **Compile time:** If you know at compile time where the process will reside in memory, then **absolute code** can be generated. For example, if you know *a priori* that a user process resides starting at location R, then the generated compiler code will start at that location and extend up from there. If, at some later time, the starting location changes, then it will be necessary to recompile this code. The MS-DOS .COM-format programs are absolute code bound at compile time.
- **Load time:** If it is not known at compile time where the process will reside in memory, then the compiler must generate **relocatable code**. In this case,

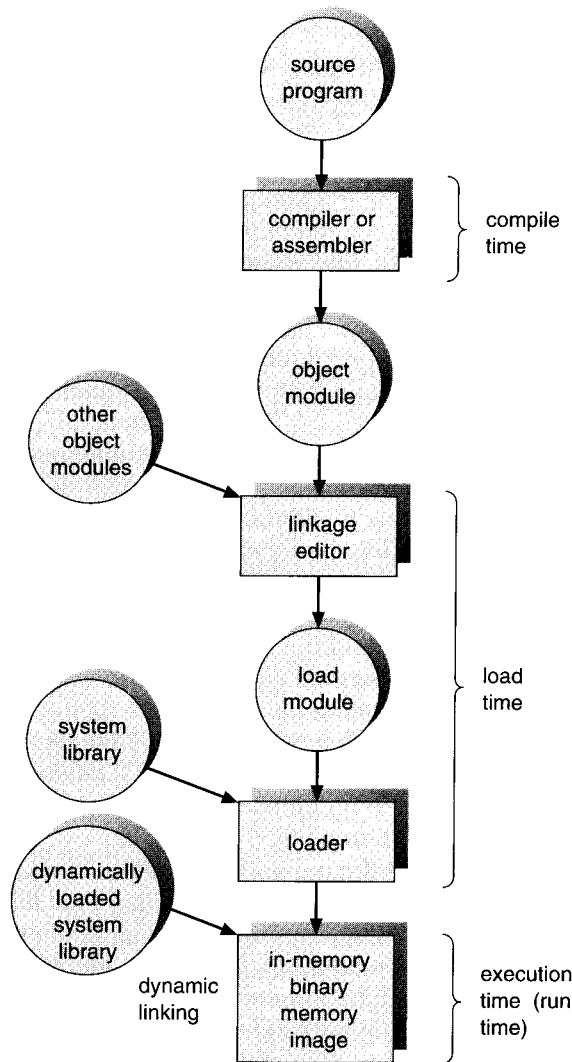


Figure 9.1 Multistep processing of a user program.

final binding is delayed until load time. If the starting address changes, we need only to reload the user code to incorporate this changed value.

- **Execution time:** If the process can be moved during its execution from one memory segment to another, then binding must be delayed until run time. Special hardware must be available for this scheme to work, as will be discussed in Section 9.1.2. Most general-purpose operating systems use this method.

A major portion of this chapter is devoted to showing how these various bindings can be implemented effectively in a computer system and to discussing appropriate hardware support.

9.1.2 Logical- Versus Physical-Address Space

An address generated by the CPU is commonly referred to as a **logical address**, whereas an address seen by the memory unit—that is, the one loaded into the **memory-address register** of the memory—is commonly referred to as a **physical address**.

The compile-time and load-time address-binding methods generate identical logical and physical addresses. However, the execution-time address-binding scheme results in differing logical and physical addresses. In this case, we usually refer to the logical address as a **virtual address**. We use *logical address* and *virtual address* interchangeably in this text. The set of all logical addresses generated by a program is a **logical-address space**; the set of all physical addresses corresponding to these logical addresses is a **physical-address space**. Thus, in the execution-time address-binding scheme, the logical- and physical-address spaces differ.

The run-time mapping from virtual to physical addresses is done by a hardware device called the **memory-management unit (MMU)**. We can choose from among many different methods to accomplish such a mapping, as we discuss in Sections 9.3, 9.4, 9.5, and 9.6. For the time being, we illustrate this mapping with a simple MMU scheme, which is a generalization of the base-register scheme described in Section 2.5.3.

As illustrated in Figure 9.2, this method requires hardware support slightly different from the hardware configuration discussed in Section 2.4. The base register is now called a **relocation register**. The value in the relocation register is *added* to every address generated by a user process at the time it is sent to memory. For example, if the base is at 14000, then an attempt by the user to address location 0 is dynamically relocated to location 14000; an access to location 346 is mapped to location 14346. The MS-DOS operating system running on the Intel 80x86 family of processors uses four relocation registers when loading and running processes.

The user program never sees the *real* physical addresses. The program can create a pointer to location 346, store it in memory, manipulate it, compare it to other addresses—all as the number 346. Only when it is used as a memory address (in an indirect load or store, perhaps) is it relocated relative to the base register. The user program deals with *logical* addresses. The memory-mapping hardware converts logical addresses into physical addresses. This form of execution-time binding was discussed in Section 9.1.1. The final location of a referenced memory address is not determined until the reference is made.

We now have two different types of addresses: logical addresses (in the range 0 to *max*) and physical addresses (in the range $R + 0$ to $R + max$ for a base

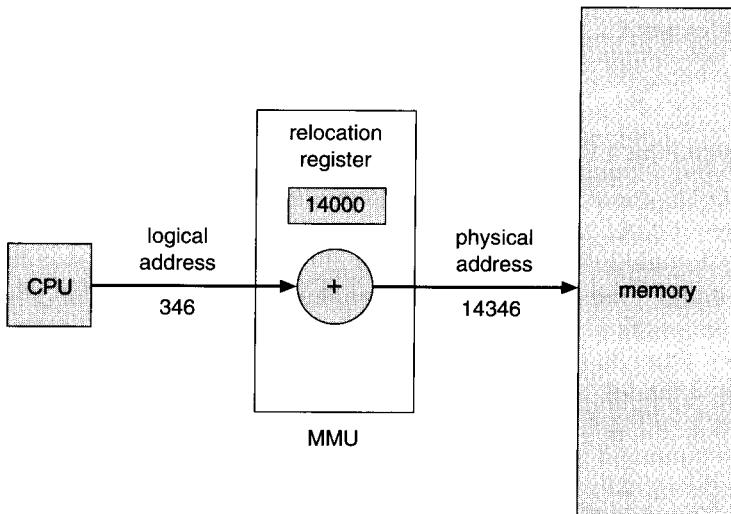


Figure 9.2 Dynamic relocation using a relocation register.

value R). The user generates only logical addresses and thinks that the process runs in locations 0 to max . The user program supplies logical addresses; these logical addresses must be mapped to physical addresses before they are used.

The concept of a *logical-address space* that is bound to a separate *physical-address space* is central to proper memory management.

9.1.3 Dynamic Loading

In our discussion so far, the entire program and data of a process must be in physical memory for the process to execute. The size of a process is limited to the size of physical memory. To obtain better memory-space utilization, we can use **dynamic loading**. With dynamic loading, a routine is not loaded until it is called. All routines are kept on disk in a relocatable load format. The main program is loaded into memory and is executed. When a routine needs to call another routine, the calling routine first checks to see whether the other routine has been loaded. If not, the relocatable linking loader is called to load the desired routine into memory and to update the program's address tables to reflect this change. Then, control is passed to the newly loaded routine.

The advantage of dynamic loading is that an unused routine is never loaded. This method is particularly useful when large amounts of code are needed to handle infrequently occurring cases, such as error routines. In this case, although the total program size may be large, the portion that is used (and hence loaded) may be much smaller.

Dynamic loading does not require special support from the operating system. It is the responsibility of the users to design their programs to take

advantage of such a method. Operating systems may help the programmer, however, by providing library routines to implement dynamic loading.

9.1.4 Dynamic Linking and Shared Libraries

Figure 9.1 also shows **dynamically linked libraries**. Some operating systems support only **static linking**, in which system language libraries are treated like any other object module and are combined by the loader into the binary program image. The concept of dynamic linking is similar to that of dynamic loading. Rather than loading being postponed until execution time, linking is postponed. This feature is usually used with system libraries, such as language subroutine libraries. Without this facility, all programs on a system need to have a copy of their language library (or at least the routines referenced by the program) included in the executable image. This requirement wastes both disk space and main memory. With dynamic linking, a stub is included in the image for each library-routine reference. This *stub* is a small piece of code that indicates how to locate the appropriate memory-resident library routine, or how to load the library if the routine is not already present.

When this stub is executed, it checks to see whether the needed routine is already in memory. If not, the program loads the routine into memory. Either way, the stub replaces itself with the address of the routine, and executes the routine. Thus, the next time that that code segment is reached, the library routine is executed directly, incurring no cost for dynamic linking. Under this scheme, all processes that use a language library execute only one copy of the library code.

This feature can be extended to library updates (such as bug fixes). A library may be replaced by a new version, and all programs that reference the library will automatically use the new version. Without dynamic linking, all such programs would need to be relinked to gain access to the new library. So that programs will not accidentally execute new, incompatible versions of libraries, version information is included in both the program and the library. More than one version of a library may be loaded into memory, and each program uses its version information to decide which copy of the library to use. Minor changes retain the same version number, whereas major changes increment the version number. Thus, only programs that are compiled with the new library version are affected by the incompatible changes incorporated in it. Other programs linked before the new library was installed will continue using the older library. This system is also known as **shared libraries**.

Unlike dynamic loading, dynamic linking generally requires help from the operating system. If the processes in memory are protected from one another (Section 9.3), then the operating system is the only entity that can check to see whether the needed routine is in another process' memory space, or that can allow multiple processes to access the same memory addresses. We elaborate on this concept when we discuss paging in Section 9.4.5.

9.1.5 Overlays

To enable a process to be larger than the amount of memory allocated to it, we can use **overlays**. The idea of overlays is to keep in memory only those instructions and data that are needed at any given time. When other instructions are needed, they are loaded into space occupied previously by instructions that are no longer needed.

As an example, consider a two-pass assembler. During pass 1, it constructs a symbol table; then, during pass 2, it generates machine-language code. We may be able to partition such an assembler into pass 1 code, pass 2 code, the symbol table, and common support routines used by both pass 1 and pass 2. Assume that the sizes of these components are as follows:

Pass 1	70 KB
Pass 2	80 KB
Symbol table	20 KB
Common routines	30 KB

To load everything at once, we would require 200 KB of memory. If only 150 KB is available, we cannot run our process. However, notice that pass 1 and pass 2 do not need to be in memory at the same time. We thus define two overlays: Overlay A is the symbol table, common routines, and pass 1, and overlay B is the symbol table, common routines, and pass 2.

We add an overlay driver (10 KB) and start with overlay A in memory. When we finish pass 1, we jump to the overlay driver, which reads overlay B into memory, overwriting overlay A, and then transfers control to pass 2. Overlay A needs only 120 KB, whereas overlay B needs 130 KB (Figure 9.3). We can now run our assembler in the 150 KB of memory. It will load somewhat faster because fewer data need to be transferred before execution starts. However, it will run somewhat slower, due to the extra I/O to read the code for overlay B over the code for overlay A.

The code for overlay A and the code for overlay B are kept on disk as absolute memory images, and are read by the overlay driver as needed. Special relocation and linking algorithms are needed to construct the overlays.

As in dynamic loading, overlays do not require any special support from the operating system. They can be implemented completely by the user with simple file structures, reading from the files into memory and then jumping to that memory and executing the newly read instructions. The operating system notices only that there is more I/O than usual.

The programmer, on the other hand, must design and program the overlay structure properly. This task can be a major undertaking, requiring complete knowledge of the structure of the program, its code, and its data structures. Because the program is, by definition, large—small programs do not need to be overlaid—obtaining a sufficient understanding of the program may be difficult. For these reasons, the use of overlays is currently limited to microcomputer

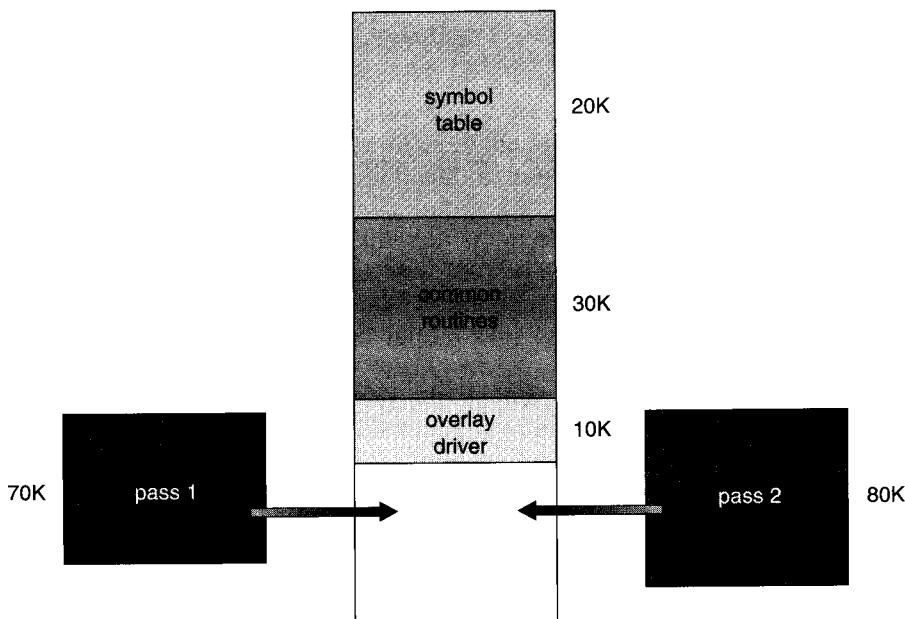


Figure 9.3 Overlays for a two-pass assembler.

and other systems that have limited amounts of physical memory and that lack hardware support for more advanced techniques. Some microcomputer compilers provide the programmer with support for overlays to make the task easier. Automatic techniques to run large programs in limited amounts of physical memory are certainly preferable.

9.2 ■ Swapping

A process needs to be in memory to be executed. A process, however, can be **swapped** temporarily out of memory to a **backing store**, and then brought back into memory for continued execution. For example, assume a multiprogramming environment with a round-robin CPU-scheduling algorithm. When a quantum expires, the memory manager will start to swap out the process that just finished, and to swap in another process to the memory space that has been freed (Figure 9.4). In the meantime, the CPU scheduler will allocate a time slice to some other process in memory. When each process finishes its quantum, it will be swapped with another process. Ideally, the memory manager can swap processes fast enough that some processes will be in memory, ready to execute, when the CPU scheduler wants to reschedule the CPU. The quantum must also be sufficiently large that reasonable amounts of computing are done between swaps.

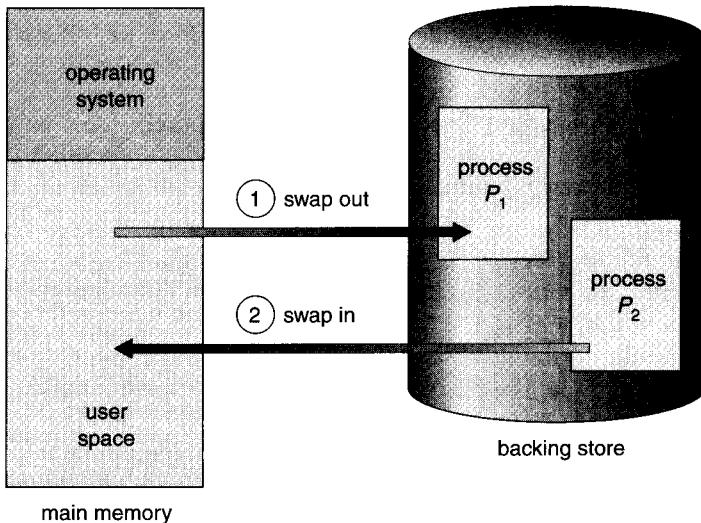


Figure 9.4 Swapping of two processes using a disk as a backing store.

A variant of this swapping policy is used for priority-based scheduling algorithms. If a higher-priority process arrives and wants service, the memory manager can swap out the lower-priority process so that it can load and execute the higher-priority process. When the higher-priority process finishes, the lower-priority process can be swapped back in and continued. This variant of swapping is sometimes called **roll out, roll in**.

Normally a process that is swapped out will be swapped back into the same memory space that it occupied previously. This restriction is dictated by the method of address binding. If binding is done at assembly or load time, then the process cannot be moved to different locations. If execution-time binding is being used, then a process can be swapped into a different memory space, because the physical addresses are computed during execution time.

Swapping requires a backing store. The backing store is commonly a fast disk. It must be large enough to accommodate copies of all memory images for all users, and it must provide direct access to these memory images. The system maintains a **ready queue** consisting of all processes whose memory images are on the backing store or in memory and are ready to run. Whenever the CPU scheduler decides to execute a process, it calls the dispatcher. The dispatcher checks to see whether the next process in the queue is in memory. If not, and there is no free memory region, the dispatcher swaps out a process currently in memory and swaps in the desired process. It then reloads registers as normal and transfers control to the selected process.

The context-switch time in such a swapping system is fairly high. To get an idea of the context-switch time, let us assume that the user process is of size 1

MB and the backing store is a standard hard disk with a transfer rate of 5 MB per second. The actual transfer of the 1 MB process to or from memory takes

$$\begin{aligned} 1000 \text{ KB} / 5000 \text{ KB per second} &= 1/5 \text{ second} \\ &= 200 \text{ milliseconds.} \end{aligned}$$

Assuming that no head seeks are necessary and an average latency of 8 milliseconds, the swap time takes 208 milliseconds. Since we must both swap out and swap in, the total swap time is then about 416 milliseconds.

For efficient CPU utilization, we want our execution time for each process to be long relative to the swap time. Thus, in a round-robin CPU-scheduling algorithm, for example, the time quantum should be substantially larger than 0.416 seconds.

Notice that the major part of the swap time is transfer time. The total transfer time is directly proportional to the *amount* of memory swapped. If we have a computer system with 128 MB of main memory and a resident operating system taking 5 MB, the maximum size of the user process is 123 MB. However, many user processes may be much smaller than this size—say, 1 MB. A 1 MB process could be swapped out in 208 milliseconds, compared to the 24.6 seconds for swapping 123 MB. Therefore, it would be useful to know exactly how much memory a user process *is* using, not simply how much it *might be* using. Then, we would need to swap only what is actually used, reducing swap time. For this method to be effective, the user must keep the system informed of any changes in memory requirements. Thus, a process with dynamic memory requirements will need to issue system calls (`request_memory` and `release_memory`) to inform the operating system of its changing memory needs.

Swapping is constrained by other factors as well. If we want to swap a process, we must be sure that it is completely idle. Of particular concern is any pending I/O. A process may be waiting for an I/O operation when we want to swap that process to free up its memory. However, if the I/O is asynchronously accessing the user memory for I/O buffers, then the process cannot be swapped. Assume that the I/O operation was queued because the device was busy. Then, if we were to swap out process P_1 and swap in process P_2 , the I/O operation might then attempt to use memory that now belongs to process P_2 . The two main solutions to this problem are never to swap a process with pending I/O, or to execute I/O operations only into operating-system buffers. Transfers between operating-system buffers and process memory then occur only when the process is swapped in.

The assumption that swapping requires few, if any, head seeks needs further explanation. We postpone discussing this issue until Chapter 14, where secondary-storage structure is covered. Generally, swap space is allocated as a chunk of disk, separate from the file system, so that its use is as fast as possible.

Currently, standard swapping is used in few systems. It requires too much swapping time and provides too little execution time to be a reasonable

memory-management solution. Modified versions of swapping, however, are found on many systems.

A modification of swapping is used in many versions of UNIX. Swapping was normally disabled, but would start if many processes were running and were using a threshold amount of memory. Swapping would again be halted if the load on the system were reduced. Memory management in UNIX is described fully in Section A.6.

Early PCs lacked sophisticated hardware (or operating systems that take advantage of the hardware) to implement more advanced memory-management methods, but they were used to run multiple large processes by a modified version of swapping. A prime example is the Microsoft Windows 3.1 operating system, which supports concurrent execution of processes in memory. If a new process is loaded and there is insufficient main memory, an old process is swapped to disk. This operating system, however, does not provide full swapping, because the user, rather than the scheduler, decides when it is time to preempt one process for another. Any swapped-out process remains swapped out (and not executing) until the user selects that process to run. Follow-on Microsoft operating systems, such as Windows NT, take advantage of advanced MMU features now found even on PCs. In Section 9.6, we describe the memory-management hardware found on the Intel 386 family of processors used in many PCs. In that section, we also describe the memory management used on this CPU by another advanced operating system for PCs: IBM OS/2.

9.3 ■ Contiguous Memory Allocation

The main memory must accommodate both the operating system and the various user processes. We therefore need to allocate different parts of the main memory in the most efficient way possible. This section will explain one common method, contiguous memory allocation.

The memory is usually divided into two partitions: one for the resident operating system, and one for the user processes. We may place the operating system in either low memory or high memory. The major factor affecting this decision is the location of the interrupt vector. Since the interrupt vector is often in low memory, programmers usually place the operating system in low memory as well. Thus, in this text, we shall discuss only the situation where the operating system resides in low memory. The development of the other situation is similar.

We usually want several user processes to reside in memory at the same time. We therefore need to consider how to allocate available memory to the processes that are in the input queue waiting to be brought into memory. In this contiguous memory allocation, each process is contained in a single contiguous section of memory.

9.3.1 Memory Protection

Before discussing memory allocation, we must discuss the issue of memory protection—protecting the operating system from user processes, and protecting user processes from one another. We can provide this protection by using a relocation register, as discussed in Section 9.1.2, with a limit register, as discussed in Section 2.5.3. The relocation register contains the value of the smallest physical address; the limit register contains the range of logical addresses (for example, relocation = 100040 and limit = 74600). With relocation and limit registers, each logical address must be less than the limit register; the MMU maps the logical address *dynamically* by adding the value in the relocation register. This mapped address is sent to memory (Figure 9.5).

When the CPU scheduler selects a process for execution, the dispatcher loads the relocation and limit registers with the correct values as part of the context switch. Because every address generated by the CPU is checked against these registers, we can protect both the operating system and the other users' programs and data from being modified by this running process.

The relocation-register scheme provides an effective way to allow the operating-system size to change dynamically. This flexibility is desirable in many situations. For example, the operating system contains code and buffer space for device drivers. If a device driver (or other operating-system service) is not commonly used, we do not want to keep the code and data in memory, as we might be able to use that space for other purposes. Such code is sometimes called **transient** operating-system code; it comes and goes as needed. Thus, using this code changes the size of the operating system during program execution.

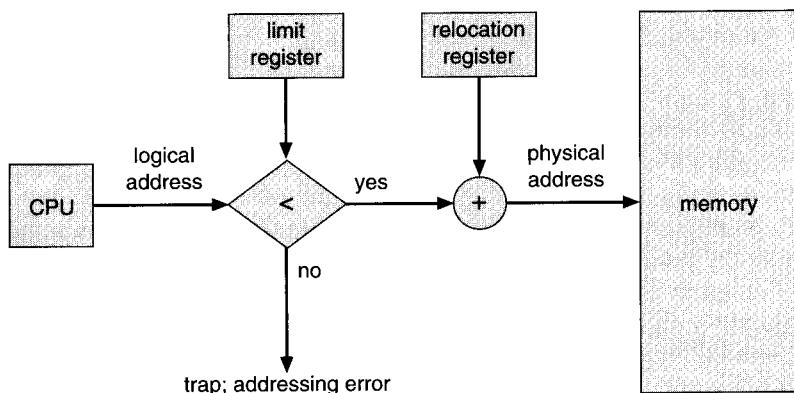


Figure 9.5 Hardware support for relocation and limit registers.

9.3.2 Memory Allocation

Now we are ready to turn to memory allocation. One of the simplest methods for memory allocation is to divide memory into several fixed-sized **partitions**. Each partition may contain exactly one process. Thus, the degree of multiprogramming is bound by the number of partitions. In this **multiple-partition method**, when a partition is free, a process is selected from the input queue and is loaded into the free partition. When the process terminates, the partition becomes available for another process. This method was originally used by the IBM OS/360 operating system (called MFT); it is no longer in use. The method described next is a generalization of the fixed-partition scheme (called MVT); it is used primarily in a batch environment. Many of the ideas presented here are also applicable to a time-sharing environment in which pure segmentation is used for memory management (Section 9.5).

The operating system keeps a table indicating which parts of memory are available and which are occupied. Initially, all memory is available for user processes, and is considered as one large block of available memory, a **hole**. When a process arrives and needs memory, we search for a hole large enough for this process. If we find one, we allocate only as much memory as is needed, keeping the rest available to satisfy future requests.

As processes enter the system, they are put into an input queue. The operating system takes into account the memory requirements of each process and the amount of available memory space in determining which processes are allocated memory. When a process is allocated space, it is loaded into memory and it can then compete for the CPU. When a process terminates, it releases its memory, which the operating system may then fill with another process from the input queue.

At any given time, we have a list of available block sizes and the input queue. The operating system can order the input queue according to a scheduling algorithm. Memory is allocated to processes until, finally, the memory requirements of the next process cannot be satisfied; no available block of memory (or hole) is large enough to hold that process. The operating system can then wait until a large enough block is available, or it can skip down the input queue to see whether the smaller memory requirements of some other process can be met.

In general, a *set* of holes, of various sizes, is scattered throughout memory at any given time. When a process arrives and needs memory, the system searches this set for a hole that is large enough for this process. If the hole is too large, it is split into two: One part is allocated to the arriving process; the other is returned to the set of holes. When a process terminates, it releases its block of memory, which is then placed back in the set of holes. If the new hole is adjacent to other holes, these adjacent holes are merged to form one larger hole. At this point, the system may need to check whether there are processes waiting for memory and whether this newly freed and recombined memory could satisfy the demands of any of these waiting processes.

This procedure is a particular instance of the general **dynamic storage-allocation problem**, which is how to satisfy a request of size n from a list of free holes. There are many solutions to this problem. The set of holes is searched to determine which hole is best to allocate. The **first-fit**, **best-fit**, and **worst-fit** strategies are the most common ones used to select a free hole from the set of available holes.

- *First fit:* Allocate the *first* hole that is big enough. Searching can start either at the beginning of the set of holes or where the previous first-fit search ended. We can stop searching as soon as we find a free hole that is large enough.
- *Best fit:* Allocate the *smallest* hole that is big enough. We must search the entire list, unless the list is kept ordered by size. This strategy produces the smallest leftover hole.
- *Worst fit:* Allocate the *largest* hole. Again, we must search the entire list, unless it is sorted by size. This strategy produces the largest leftover hole, which may be more useful than the smaller leftover hole from a best-fit approach.

Simulations have shown that both first fit and best fit are better than worst fit in terms of decreasing both time and storage utilization. Neither first fit nor best fit is clearly better in terms of storage utilization, but first fit is generally faster.

These algorithms, however, suffer from **external fragmentation**. As processes are loaded and removed from memory, the free memory space is broken into little pieces. External fragmentation exists when enough total memory space exists to satisfy a request, but it is not contiguous; storage is fragmented into a large number of small holes. This fragmentation problem can be severe. In the worst case, we could have a block of free (or wasted) memory between every two processes. If all this memory were in one big free block, we might be able to run several more processes.

The selection of the first-fit versus best-fit strategies can affect the amount of fragmentation. (First fit is better for some systems, whereas best fit is better for others.) Another factor is which end of a free block is allocated. (Which is the leftover piece—the one on the top, or the one on the bottom?) No matter which algorithm is used, external fragmentation will be a problem.

Depending on the total amount of memory storage and the average process size, external fragmentation may be a minor or a major problem. Statistical analysis of first fit, for instance, reveals that, even with some optimization, given N allocated blocks, another $0.5N$ blocks will be lost due to fragmentation. That is, one-third of memory may be unusable! This property is known as the **50-percent rule**.

9.3.3 Fragmentation

Memory fragmentation can be internal as well as external. Consider a multiple-partition allocation scheme with a hole of 18,464 bytes. Suppose that the next process requests 18,462 bytes. If we allocate exactly the requested block, we are left with a hole of 2 bytes. The overhead to keep track of this hole will be substantially larger than the hole itself. The general approach is to break the physical memory into fixed-sized blocks, and allocate memory in unit of block sizes. With this approach, the memory allocated to a process may be slightly larger than the requested memory. The difference between these two numbers is **internal fragmentation**—memory that is internal to a partition but is not being used.

One solution to the problem of external fragmentation is **compaction**. The goal is to shuffle the memory contents to place all free memory together in one large block. Compaction is not always possible. If relocation is static and is done at assembly or load time, compaction cannot be done; compaction is possible *only* if relocation is dynamic, and is done at execution time. If addresses are relocated dynamically, relocation requires only moving the program and data, and then changing the base register to reflect the new base address. When compaction is possible, we must determine its cost. The simplest compaction algorithm is simply to move all processes toward one end of memory; all holes move in the other direction, producing one large hole of available memory. This scheme can be expensive.

Another possible solution to the external-fragmentation problem is to permit the logical-address space of a process to be noncontiguous, thus allowing a process to be allocated physical memory wherever the latter is available. Two complementary techniques achieve this solution: paging (Section 9.4) and segmentation (Section 9.5). These techniques can also be combined (Section 9.6).

9.4 ■ Paging

Paging is a memory-management scheme that permits the physical-address space of a process to be noncontiguous. Paging avoids the considerable problem of fitting the varying-sized memory chunks onto the backing store, from which most of the previous memory-management schemes suffered. When some code fragments or data residing in main memory need to be swapped out, space must be found on the backing store. The fragmentation problems discussed in connection with main memory are also prevalent with backing store, except that access is much slower, so compaction is impossible. Because of its advantages over the previous methods, paging in its various forms is commonly used in most operating systems.

Traditionally, support for paging has been handled by hardware. However, recent designs have implemented paging by closely integrating the hardware and operating system, especially on 64-bit microprocessors.

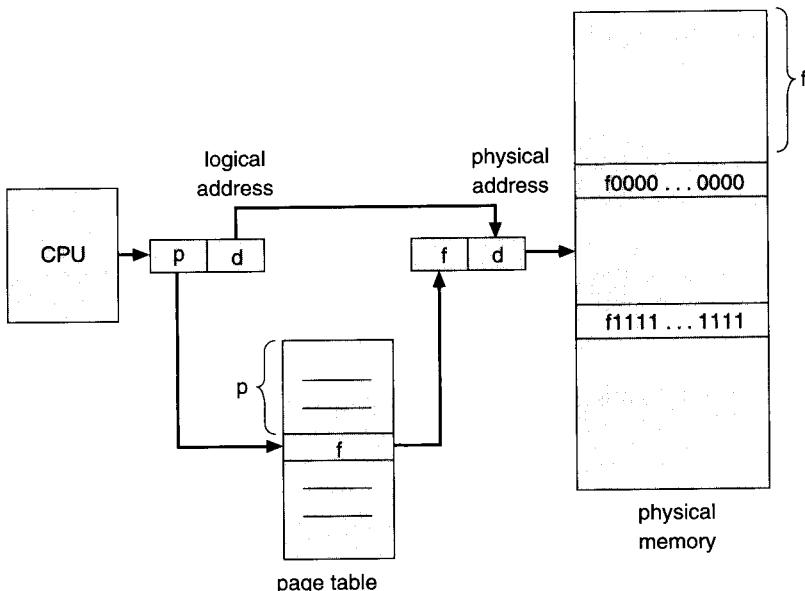


Figure 9.6 Paging hardware.

9.4.1 Basic Method

Physical memory is broken into fixed-sized blocks called **frames**. Logical memory is also broken into blocks of the same size called **pages**. When a process is to be executed, its pages are loaded into any available memory frames from the backing store. The backing store is divided into fixed-sized blocks that are of the same size as the memory frames.

The hardware support for paging is illustrated in Figure 9.6. Every address generated by the CPU is divided into two parts: a **page number (p)** and a **page offset (d)**. The page number is used as an index into a **page table**. The page table contains the base address of each page in physical memory. This base address is combined with the page offset to define the physical memory address that is sent to the memory unit. The paging model of memory is shown in Figure 9.7.

The page size (like the frame size) is defined by the hardware. The size of a page is typically a power of 2, varying between 512 bytes and 16 MB per page, depending on the computer architecture. The selection of a power of 2 as a page size makes the translation of a logical address into a page number and page offset particularly easy. If the size of logical-address space is 2^m , and a page size is 2^n addressing units (bytes or words), then the high-order $m - n$ bits of a logical address designate the page number, and the n low-order bits designate the page offset. Thus, the logical address is as follows:

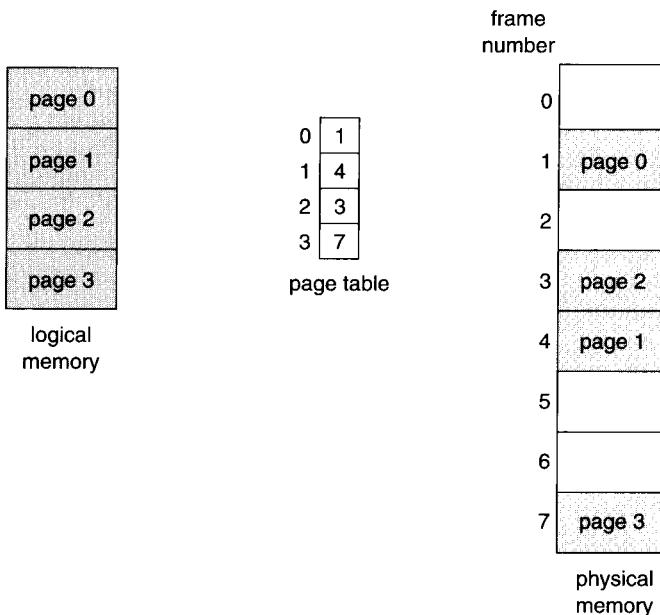


Figure 9.7 Paging model of logical and physical memory.

page number	page offset
p	d
$m - n$	n

where p is an index into the page table and d is the displacement within the page.

As a concrete (although minuscule) example, consider the memory in Figure 9.8. Using a page size of 4 bytes and a physical memory of 32 bytes (8 pages), we show how the user's view of memory can be mapped into physical memory. Logical address 0 is page 0, offset 0. Indexing into the page table, we find that page 0 is in frame 5. Thus, logical address 0 maps to physical address 20 ($= (5 \times 4) + 0$). Logical address 3 (page 0, offset 3) maps to physical address 23 ($= (5 \times 4) + 3$). Logical address 4 is page 1, offset 0; according to the page table, page 1 is mapped to frame 6. Thus, logical address 4 maps to physical address 24 ($= (6 \times 4) + 0$). Logical address 13 maps to physical address 9.

You may have noticed that paging itself is a form of dynamic relocation. Every logical address is bound by the paging hardware to some physical address. Using paging is similar to using a table of base (or relocation) registers, one for each frame of memory.

When we use a paging scheme, we have no external fragmentation: *Any* free frame can be allocated to a process that needs it. However, we may have some internal fragmentation. Notice that frames are allocated as units. If the

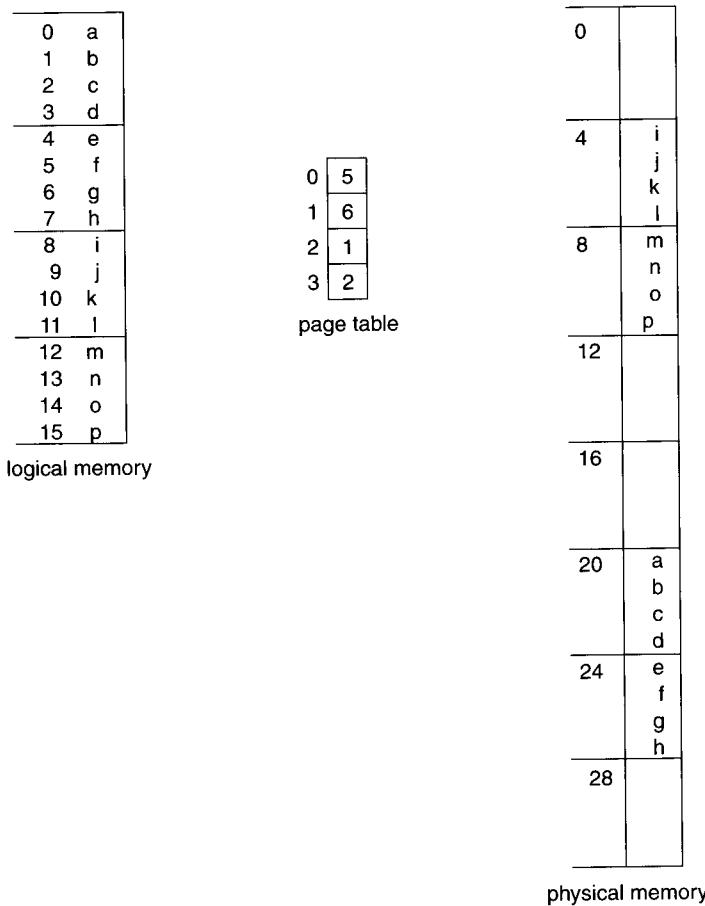


Figure 9.8 Paging example for a 32-byte memory with 4-byte pages.

memory requirements of a process do not happen to fall on page boundaries, the *last* frame allocated may not be completely full. For example, if pages are 2,048 bytes, a process of 72,766 bytes would need 35 pages plus 1,086 bytes. It would be allocated 36 frames, resulting in an internal fragmentation of $2048 - 1086 = 962$ bytes. In the worst case, a process would need n pages plus one byte. It would be allocated $n + 1$ frames, resulting in an internal fragmentation of almost an entire frame.

If process size is independent of page size, we expect internal fragmentation to average one-half page per process. This consideration suggests that small page sizes are desirable. However, overhead is involved in each page-table entry, and this overhead is reduced as the size of the pages increases. Also, disk I/O is more efficient when the number of data being transferred is larger (Chapter 14). Generally, page sizes have grown over time as processes, data

sets, and main memory have become larger. Today, pages typically are between 4 KB and 8 KB, and some systems support even larger page sizes. Some CPUs and kernels even support multiple page sizes. For instance, Solaris uses 8 KB and 4 MB page sizes, depending on the data stored by the pages. Researchers are now developing variable on-the-fly page-size support.

Each page-table entry is usually 4 bytes long, but that size can vary as well. A 32-bit entry can point to one of 2^{32} physical page frames. If a frame is 4 KB, then a system with 4-byte entries can address 2^{36} bytes (or 64 GB) of physical memory.

When a process arrives in the system to be executed, its size, expressed in pages, is examined. Each page of the process needs one frame. Thus, if the process requires n pages, at least n frames must be available in memory. If n frames are available, they are allocated to this arriving process. The first page of the process is loaded into one of the allocated frames, and the frame number is put in the page table for this process. The next page is loaded into another frame, and its frame number is put into the page table, and so on (Figure 9.9).

An important aspect of paging is the clear separation between the user's view of memory and the actual physical memory. The user program views that memory as one single contiguous space, containing only this one program. In fact, the user program is scattered throughout physical memory, which also holds other programs. The difference between the user's view of memory and

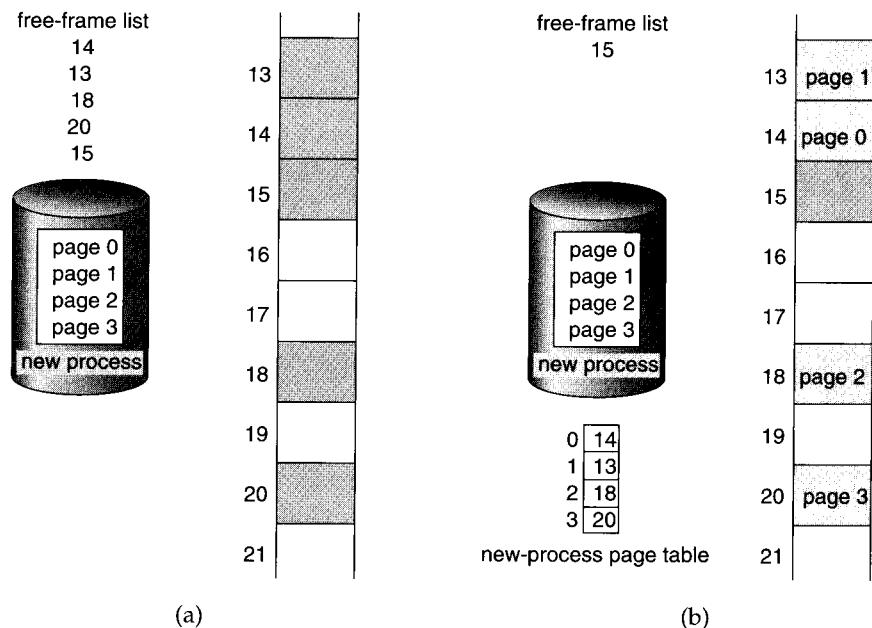


Figure 9.9 Free frames. (a) Before allocation. (b) After allocation.

the actual physical memory is reconciled by the address-translation hardware. The logical addresses are translated into physical addresses. This mapping is hidden from the user and is controlled by the operating system. Notice that the user process by definition is unable to access memory it does not own. It has no way of addressing memory outside of its page table, and the table includes only those pages that the process owns.

Since the operating system is managing physical memory, it must be aware of the allocation details of physical memory: which frames are allocated, which frames are available, how many total frames there are, and so on. This information is generally kept in a data structure called a **frame table**. The frame table has one entry for each physical page frame, indicating whether the latter is free or allocated and, if it is allocated, to which page of which process or processes.

In addition, the operating system must be aware that user processes operate in user space, and all logical addresses must be mapped to produce physical addresses. If a user makes a system call (to do I/O, for example) and provides an address as a parameter (a buffer, for instance), that address must be mapped to produce the correct physical address. The operating system maintains a copy of the page table for each process, just as it maintains a copy of the instruction counter and register contents. This copy is used to translate logical addresses to physical addresses whenever the operating system must map a logical address to a physical address manually. It is also used by the CPU dispatcher to define the hardware page table when a process is to be allocated the CPU. Paging therefore increases the context-switch time.

9.4.2 Hardware Support

Each operating system has its own methods for storing page tables. Most allocate a page table for each process. A pointer to the page table is stored with the other register values (like the instruction counter) in the process control block. When the dispatcher is told to start a process, it must reload the user registers and define the correct hardware page-table values from the stored user page table.

The hardware implementation of the page table can be done in several ways. In the simplest case, the page table is implemented as a set of dedicated **registers**. These registers should be built with very high-speed logic to make the paging-address translation efficient. Every access to memory must go through the paging map, so efficiency is a major consideration. The CPU dispatcher reloads these registers, just as it reloads the other registers. Instructions to load or modify the page-table registers are, of course, privileged, so that only the operating system can change the memory map. The DEC PDP-11 is an example of such an architecture. The address consists of 16 bits, and the page size is 8 KB. The page table thus consists of eight entries that are kept in fast registers.

The use of registers for the page table is satisfactory if the page table is reasonably small (for example, 256 entries). Most contemporary computers,

however, allow the page table to be very large (for example, 1 million entries). For these machines, the use of fast registers to implement the page table is not feasible. Rather, the page table is kept in main memory, and a **page-table base register (PTBR)** points to the page table. Changing page tables requires changing only this one register, substantially reducing context-switch time.

The problem with this approach is the time required to access a user memory location. If we want to access location i , we must first index into the page table, using the value in the PTBR offset by the page number for i . This task requires a memory access. It provides us with the frame number, which is combined with the page offset to produce the actual address. We can then access the desired place in memory. With this scheme, *two* memory accesses are needed to access a byte (one for the page-table entry, one for the byte). Thus, memory access is slowed by a factor of 2. This delay would be intolerable under most circumstances. We might as well resort to swapping!

The standard solution to this problem is to use a special, small, fast-lookup hardware cache, called **translation look-aside buffer (TLB)**. The TLB is associative, high-speed memory. Each entry in the TLB consists of two parts: a key (or tag) and a value. When the associative memory is presented with an item, it is compared with all keys simultaneously. If the item is found, the corresponding value field is returned. The search is fast; the hardware, however, is expensive. Typically, the number of entries in a TLB is small, often numbering between 64 and 1,024.

The TLB is used with page tables in the following way. The TLB contains only a few of the page-table entries. When a logical address is generated by the CPU, its page number is presented to the TLB. If the page number is found, its frame number is immediately available and is used to access memory. The whole task may take less than 10 percent longer than it would if an unmapped memory reference were used.

If the page number is not in the TLB (known as a **TLB miss**), a memory reference to the page table must be made. When the frame number is obtained, we can use it to access memory (Figure 9.10). In addition, we add the page number and frame number to the TLB, so that they will be found quickly on the next reference. If the TLB is already full of entries, the operating system must select one for replacement. Replacement policies range from least recently used (LRU) to random. Furthermore, some TLBs allow entries to be **wired down**, meaning that they cannot be removed from the TLB. Typically, TLB entries for kernel code are often wired down.

Some TLBs store **address-space identifiers (ASIDs)** in each entry of the TLB. An ASID uniquely identifies each process and is used to provide address space protection for that process. When the TLB attempts to resolve virtual page numbers, it ensures the ASID for the currently running process matches the ASID associated with the virtual page. If the ASIDs do not match, they are treated as a TLB miss. In addition to providing address-space protection, an ASID allows the TLB to contain entries for several different processes simultaneously.

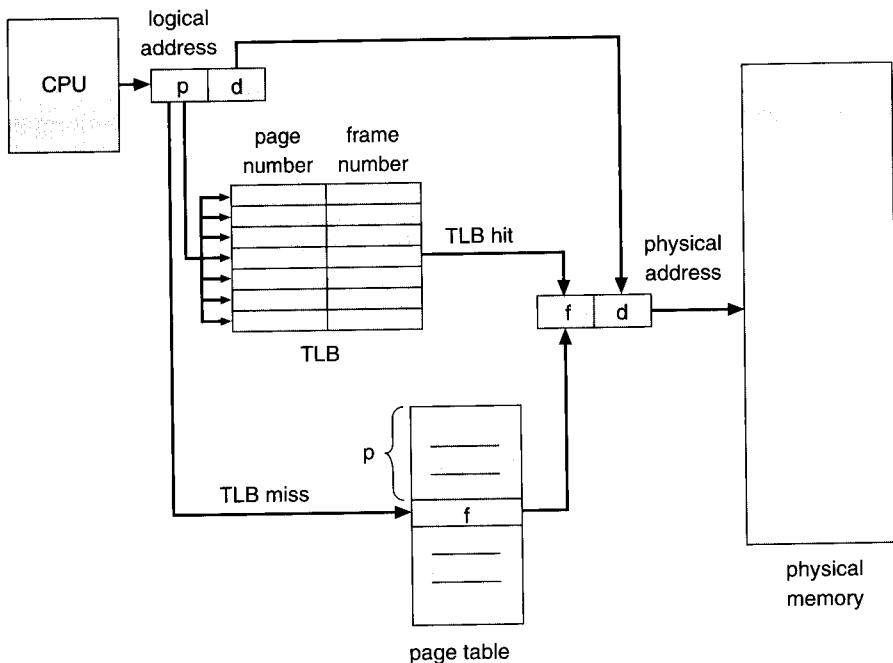


Figure 9.10 Paging hardware with TLB.

If the TLB does not support separate ASIDs, every time a new page table is selected (for instance, each context switch), the TLB must be **flushed** (or erased) to ensure that the next executing process does not use the wrong translation information. Otherwise, there could be old entries in the TLB that contain valid virtual addresses but have incorrect or invalid physical addresses left over from the previous process.

The percentage of times that a particular page number is found in the TLB is called the **hit ratio**. An 80-percent hit ratio means that we find the desired page number in the TLB 80 percent of the time. If it takes 20 nanoseconds to search the TLB, and 100 nanoseconds to access memory, then a mapped-memory access takes 120 nanoseconds when the page number is in the TLB. If we fail to find the page number in the TLB (20 nanoseconds), then we must first access memory for the page table and frame number (100 nanoseconds), and then access the desired byte in memory (100 nanoseconds), for a total of 220 nanoseconds. To find the **effective memory-access time**, we must weigh each case by its probability:

$$\begin{aligned}\text{effective access time} &= 0.80 \times 120 + 0.20 \times 220 \\ &= 140 \text{ nanoseconds.}\end{aligned}$$

In this example, we suffer a 40-percent slowdown in memory access time (from 100 to 140 nanoseconds).

For a 98-percent hit ratio, we have

$$\begin{aligned}\text{effective access time} &= 0.98 \times 120 + 0.02 \times 220 \\ &= 122 \text{ nanoseconds.}\end{aligned}$$

This increased hit rate produces only a 22-percent slowdown in access time. We will further explore the impact of the hit ratio on the TLB in Chapter 10.

9.4.3 Protection

Memory protection in a paged environment is accomplished by protection bits that are associated with each frame. Normally, these bits are kept in the page table. One bit can define a page to be read-write or read-only. Every reference to memory goes through the page table to find the correct frame number. At the same time that the physical address is being computed, the protection bits can be checked to verify that no writes are being made to a read-only page. An attempt to write to a read-only page causes a hardware trap to the operating system (or memory-protection violation).

We can easily expand this approach to provide a finer level of protection. We can create hardware to provide read-only, read-write, or execute-only protection. Or, by providing separate protection bits for each kind of access, we can allow any combination of these accesses; illegal attempts will be trapped to the operating system.

One more bit is generally attached to each entry in the page table: a **valid–invalid** bit. When this bit is set to “valid,” this value indicates that the associated page is in the process’ logical-address space, and is thus a legal (or valid) page. If the bit is set to “invalid,” this value indicates that the page is not in the process’ logical-address space. Illegal addresses are trapped by using the valid–invalid bit. The operating system sets this bit for each page to allow or disallow accesses to that page. For example, in a system with a 14-bit address space (0 to 16383), we may have a program that should use only addresses 0 to 10468. Given a page size of 2 KB, we get the situation shown in Figure 9.11. Addresses in pages 0, 1, 2, 3, 4, and 5 are mapped normally through the page table. Any attempt to generate an address in pages 6 or 7, however, finds that the valid–invalid bit is set to invalid, and the computer will trap to the operating system (invalid page reference).

Because the program extends to only address 10468, any reference beyond that address is illegal. However, references to page 5 are classified as valid, so accesses to addresses up to 12287 are valid. Only the addresses from 12288 to 16383 are invalid. This problem is a result of the 2 KB page size and reflects the internal fragmentation of paging.

Rarely does a process use all its address range. In fact, many processes use only a small fraction of the address space available to them. It would be

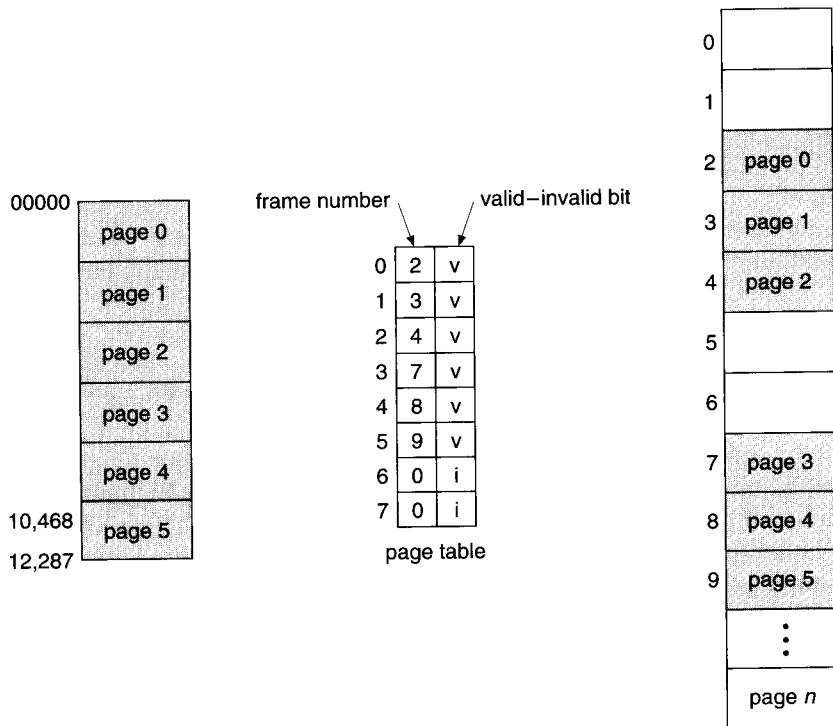


Figure 9.11 Valid (v) or invalid (i) bit in a page table.

wasteful in these cases to create a page table with entries for every page in the address range. Most of this table would be unused, but would take up valuable memory space. Some systems provide hardware, in the form of a **page-table length register (PTLR)**, to indicate the size of the page table. This value is checked against every logical address to verify that the address is in the valid range for the process. Failure of this test causes an error trap to the operating system.

9.4.4 Structure of the Page Table

In this section we explore some of the most common techniques for structuring the page table.

9.4.4.1 Hierarchical Paging

Most modern computer systems support a large logical-address space (2^{32} to 2^{64}). In such an environment, the page table itself becomes excessively large. For example, consider a system with a 32-bit logical-address space. If the page size in such a system is 4 KB (2^{12}), then a page table may consist of up to 1 million

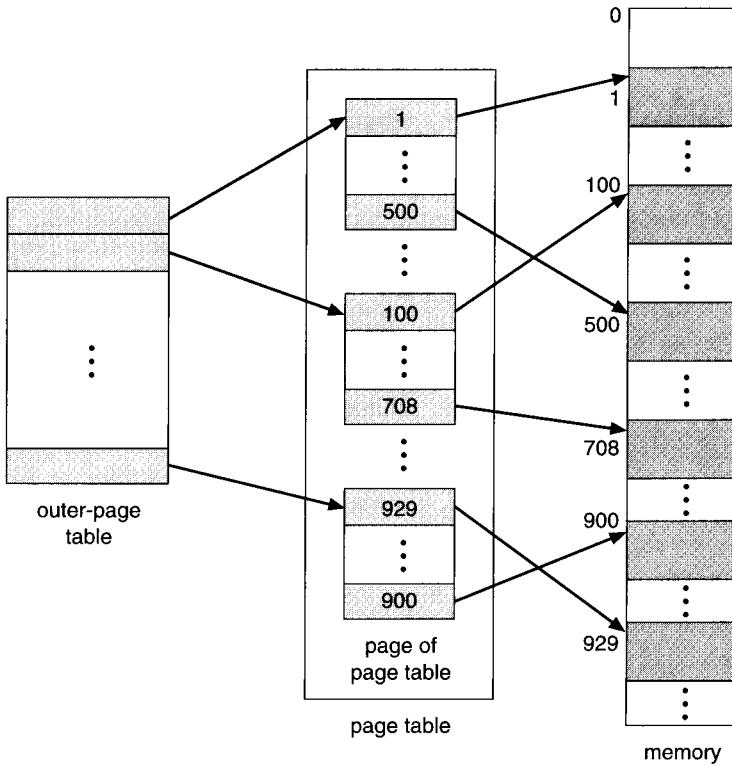


Figure 9.12 A two-level page-table scheme.

entries ($2^{32}/2^{12}$). Assuming that each entry consists of 4 bytes, each process may need up to 4 MB of physical-address space for the page table alone. Clearly, we would not want to allocate the page table contiguously in main memory. One simple solution to this problem is to divide the page table into smaller pieces. There are several ways to accomplish this division.

One way is to use a two-level paging algorithm, in which the page table itself is also paged (Figure 9.12). Remember our example to our 32-bit machine with a page size of 4 KB. A logical address is divided into a page number consisting of 20 bits, and a page offset consisting of 12 bits. Because we page the page table, the page number is further divided into a 10-bit page number and a 10-bit page offset. Thus, a logical address is as follows:

page number	page offset
p_1	p_2
10	12

where p_1 is an index into the outer page table and p_2 is the displacement within the page of the outer page table. The address-translation method for this architecture is shown in Figure 9.13. Because address translation works from the outer page table inwards, this scheme is also known as a **forward-mapped page table**. The Pentium-II uses this architecture.

The VAX architecture also supports a variation of two-level paging. The VAX is a 32-bit machine with page size of 512 bytes. The logical-address space of a process is divided into four equal sections, each of which consists of 2^{30} bytes. Each section represents a different part of the logical-address space of a process. The first 2 high-order bits of the logical address designate the appropriate section. The next 21 bits represent the logical page number of that section, and the final 9 bits represent an offset in the desired page. By partitioning the page table in this manner, the operating system can leave partitions unused until a process needs them. An address on the VAX architecture is as follows:

section	page	offset
s	p	d
2	21	9

where s designates the section number, p is an index into the page table, and d is the displacement within the page.

The size of a one-level page table for a VAX process using one section still is 2^{21} bits * 4 bytes per entry = 8 MB. So that main-memory use is reduced even further, the VAX pages the user-process page tables.

For a system with a 64-bit logical-address space, a two-level paging scheme is no longer appropriate. To illustrate this point, let us suppose that the page size in such a system is 4 KB (2^{12}). In this case, the page table will consist of up to 2^{52} entries. If we use a two-level paging scheme, then the inner page tables could conveniently be one page long, or contain 2^{10} 4-byte entries. The addresses would look like:

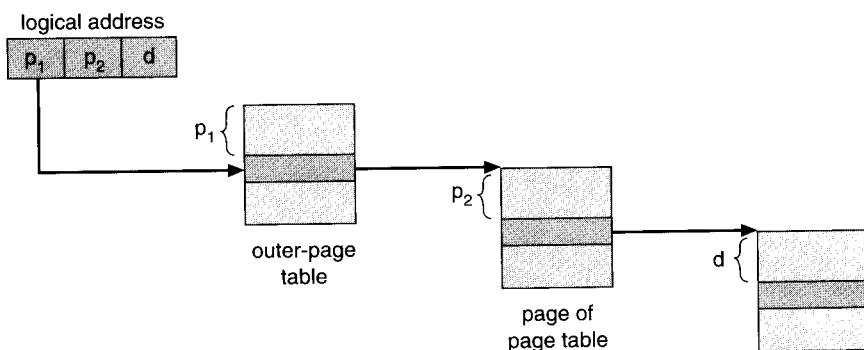


Figure 9.13 Address translation for a two-level 32-bit paging architecture.

outer page	inner page	offset
p_1	p_2	d
42	10	12

The outer page table will consist of 2^{42} entries, or 2^{44} bytes. The obvious method to avoid such a large table is to divide the outer page table into smaller pieces. This approach is also used on some 32-bit processors for added flexibility and efficiency.

We can divide the outer page table in various ways. We can page the outer page table, giving us a three-level paging scheme. Suppose that the outer page table is made up of standard-size pages (2^{10} entries, or 2^{12} bytes); a 64-bit address space is still daunting:

2nd outer page	outer page	inner page	offset
p_1	p_2	p_3	d
32	10	10	12

The outer page table is still 2^{34} bytes large.

The next step would be a four-level paging scheme, where the second-level outer page table itself is also paged. The SPARC architecture (with 32-bit addressing) supports a three-level paging scheme, whereas the 32-bit Motorola 68030 architecture supports a four-level paging scheme.

However, for 64-bit architectures, hierarchical page tables are generally considered inappropriate. For example, the 64-bit UltraSPARC would require seven levels of paging—a prohibitive number of memory accesses to translate each logical address.

9.4.4.2 Hashed Page Tables

A common approach for handling address spaces larger than 32 bits is to use a **hashed page table**, with the hash value being the virtual-page number. Each entry in the hash table contains a linked list of elements that hash to the same location (to handle collisions). Each element consists of three fields: (a) the virtual page number, (b) the value of the mapped page frame, and (c) a pointer to the next element in the linked list.

The algorithm works as follows: The virtual page number in the virtual address is hashed into the hash table. The virtual page number is compared to field (a) in the first element in the linked list. If there is a match, the corresponding page frame (field (b)) is used to form the desired physical address. If there is no match, subsequent entries in the linked list are searched for a matching virtual page number. This scheme is shown in Figure 9.14.

A variation to this scheme that is favorable for 64-bit address spaces has been proposed. **Clustered page tables** are similar to hashed page tables except that each entry in the hash table refers to several pages (such as 16) rather than a single page. Therefore, a single page-table entry can store the mappings for

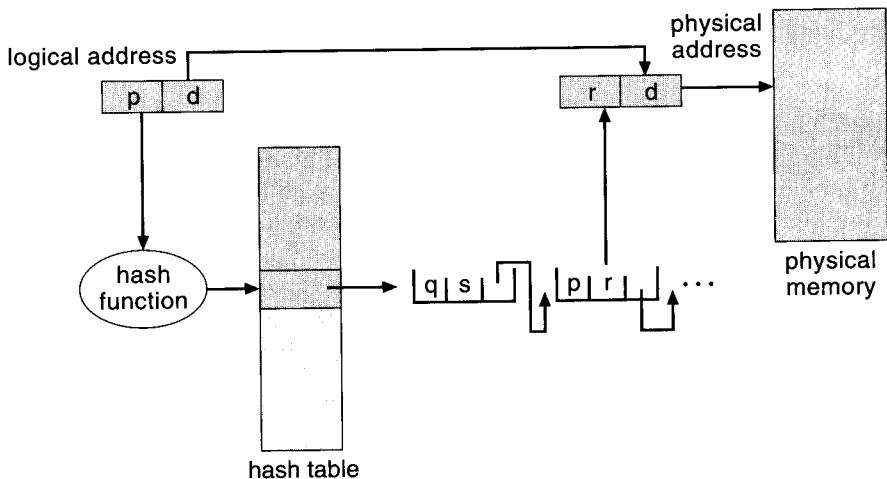


Figure 9.14 Hashed page table.

multiple physical-page frames. Clustered page tables are particularly useful for **sparse** address spaces where memory references are noncontiguous and scattered throughout the address space.

9.4.4.3 Inverted Page Table

Usually, each process has a page table associated with it. The page table has one entry for each page that the process is using (or one slot for each virtual address, regardless of the latter's validity). This table representation is a natural one, since processes reference pages through the pages' virtual addresses. The operating system must then translate this reference into a physical memory address. Since the table is sorted by virtual address, the operating system is able to calculate where in the table the associated physical-address entry is, and to use that value directly. One of the drawbacks of this method is that each page table may consist of millions of entries. These tables may consume large amounts of physical memory, which is required just to keep track of how the other physical memory is being used.

To solve this problem, we can use an **inverted page table**. An inverted page table has one entry for each real page (or frame) of memory. Each entry consists of the virtual address of the page stored in that real memory location, with information about the process that owns that page. Thus, only one page table is in the system, and it has only one entry for each page of physical memory. Figure 9.15 shows the operation of an inverted page table. Compare it to Figure 9.6, which depicts a standard page table in operation. Because only one page table is in the system yet there are usually several different address spaces mapping physical memory, inverted page tables often require an address-space identifier (Section 9.4.2) stored in each entry of the page table. Storing the

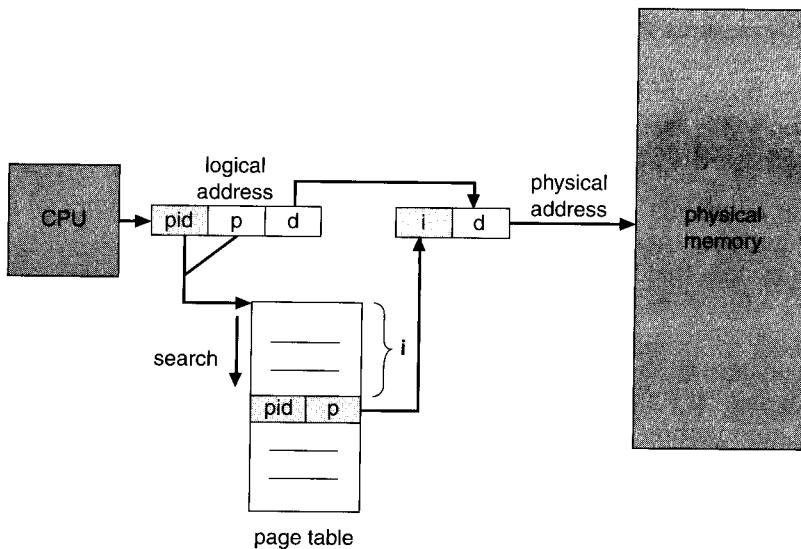


Figure 9.15 Inverted page table.

address-space identifier ensures the mapping of a logical page for a particular process to the corresponding physical page frame. Examples of systems using inverted page tables include the 64-bit UltraSPARC and PowerPC.

To illustrate this method, we describe a simplified version of the implementation of the inverted page table used in the IBM RT. Each virtual address in the system consists of a triple

$\langle \text{process-id}, \text{page-number}, \text{offset} \rangle$.

Each inverted page-table entry is a pair $\langle \text{process-id}, \text{page-number} \rangle$ where the process-id assumes the role of the address-space identifier. When a memory reference occurs, part of the virtual address, consisting of $\langle \text{process-id}, \text{page-number} \rangle$, is presented to the memory subsystem. The inverted page table is then searched for a match. If a match is found—say, at entry i —then the physical address $\langle i, \text{offset} \rangle$ is generated. If no match is found, then an illegal address access has been attempted.

Although this scheme decreases the amount of memory needed to store each page table, it increases the amount of time needed to search the table when a page reference occurs. Because the inverted page table is sorted by a physical address, but lookups occur on virtual addresses, the whole table might need to be searched for a match. This search would take far too long. To alleviate this problem, we use a hash table as described in Section 9.4.4.2 to limit the search to one—or at most a few—page-table entries. Of course, each access to the hash table adds a memory reference to the procedure, so one virtual-memory

reference requires at least two real-memory reads: one for the hash-table entry and one for the page table. To improve performance, recall that the TLB is searched first, before the hash table is consulted.

9.4.5 Shared Pages

Another advantage of paging is the possibility of *sharing* common code. This consideration is particularly important in a time-sharing environment. Consider a system that supports 40 users, each of whom executes a text editor. If the text editor consists of 150 KB of code and 50 KB of data space, we would need 8,000 KB to support the 40 users. If the code is **reentrant code**, however, it can be shared, as shown in Figure 9.16. Here we see a three-page editor—each page of size 50 KB; the large page size is used to simplify the figure—being shared among three processes. Each process has its own data page.

Reentrant code (or **pure code**) is non-self-modifying code. If the code is reentrant, then it never changes during execution. Thus, two or more processes can execute the same code at the same time. Each process has its own copy of registers and data storage to hold the data for the process' execution. The data for two different processes will, of course, vary for each process.

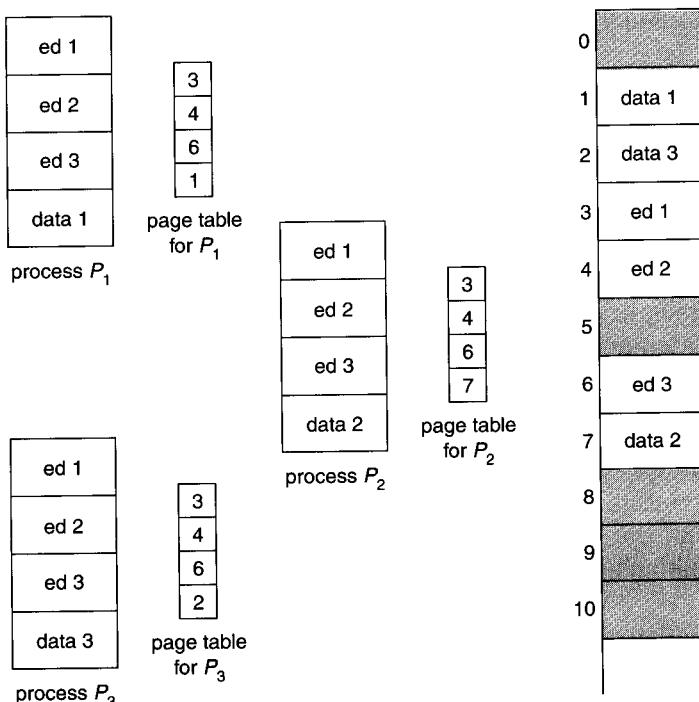


Figure 9.16 Sharing of code in a paging environment.

Only one copy of the editor needs to be kept in physical memory. Each user's page table maps onto the same physical copy of the editor, but data pages are mapped onto different frames. Thus, to support 40 users, we need only one copy of the editor (150 KB), plus 40 copies of the 50 KB of data space per user. The total space required is now 2,150 KB, instead of 8,000 KB—a significant savings.

Other heavily used programs can also be shared—compilers, window systems, run-time libraries, database systems, and so on. To be sharable, the code must be reentrant. The read-only nature of shared code should not be left to the correctness of the code; the operating system should enforce this property. This sharing of memory among processes on a system is similar to the sharing of the address space of a task by threads, described in Chapter 5. Furthermore, recall from Chapter 4 where we described shared memory as a method of interprocess communication. Some operating systems implement shared memory using shared pages.

Systems that use inverted page tables have difficulty implementing shared memory. Shared memory is usually implemented as multiple virtual addresses (one for each process sharing the memory) that are mapped to one physical address. This standard method cannot be used, however, as there is only one virtual page entry for every physical page, so one physical page cannot have two (or more) shared virtual addresses.

Organizing memory according to pages provides numerous other benefits in addition to allowing several processes to share the same physical pages. We will cover several other benefits in Chapter 10.

9.5 ■ Segmentation

An important aspect of memory management that became unavoidable with paging is the separation of the user's view of memory and the actual physical memory. The user's view of memory is not the same as the actual physical memory. The user's view is mapped onto physical memory. The mapping allows differentiation between logical memory and physical memory.

9.5.1 Basic Method

Do users think of memory as a linear array of bytes, some containing instructions and others containing data? Most people would say no. Rather, users prefer to view memory as a collection of variable-sized segments, with no necessary ordering among segments (Figure 9.17).

Consider how you think of a program when you are writing it. You think of it as a main program with a set of subroutines, procedures, functions, or modules. There may also be various data structures: tables, arrays, stacks, variables, and so on. Each of these modules or data elements is referred to by

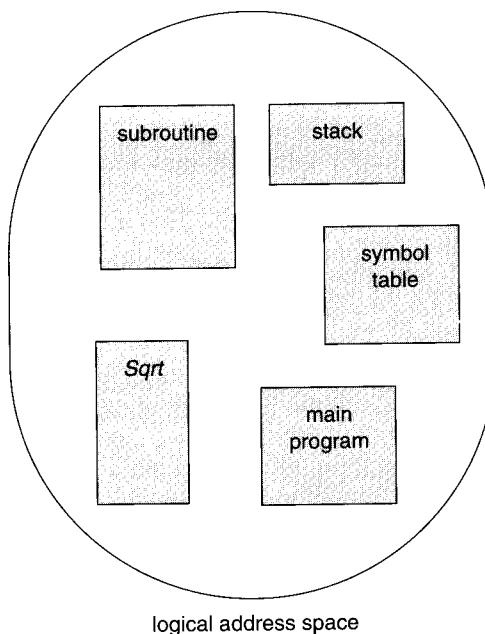


Figure 9.17 User's view of a program.

name. You talk about “the symbol table,” “function *Sqrt*,” “the main program,” without caring what addresses in memory these elements occupy. You are not concerned with whether the symbol table is stored before or after the *Sqrt* function. Each of these segments is of variable length; the length is intrinsically defined by the purpose of the segment in the program. Elements within a segment are identified by their offset from the beginning of the segment: The first statement of the program, the seventeenth entry in the symbol table, the fifth instruction of the *Sqrt* function, and so on.

Segmentation is a memory-management scheme that supports this user view of memory. A logical-address space is a collection of segments. Each segment has a name and a length. The addresses specify both the segment name and the offset within the segment. The user therefore specifies each address by two quantities: a segment name and an offset. (Contrast this scheme with the paging scheme, in which the user specified only a single address, which was partitioned by the hardware into a page number and an offset, all invisible to the programmer.)

For simplicity of implementation, segments are numbered and are referred to by a segment number, rather than by a segment name. Thus, a logical address consists of a *two tuple*:

$\langle \text{segment-number}, \text{offset} \rangle$.

Normally, the user program is compiled, and the compiler automatically constructs segments reflecting the input program. A Pascal compiler might create separate segments for the following:

1. the global variables;
2. the procedure call stack, to store parameters and return addresses;
3. the code portion of each procedure or function;
4. the local variables of each procedure and function.

A Fortran compiler might create a separate segment for each common block. Arrays might be assigned separate segments. The loader would take all these segments and assign them segment numbers.

9.5.2 Hardware

Although the user can now refer to objects in the program by a two-dimensional address, the actual physical memory is still, of course, a one-dimensional sequence of bytes. Thus, we must define an implementation to map two-dimensional user-defined addresses into one-dimensional physical addresses. This mapping is affected by a **segment table**. Each entry of the segment table has a segment *base* and a segment *limit*. The segment base contains the starting

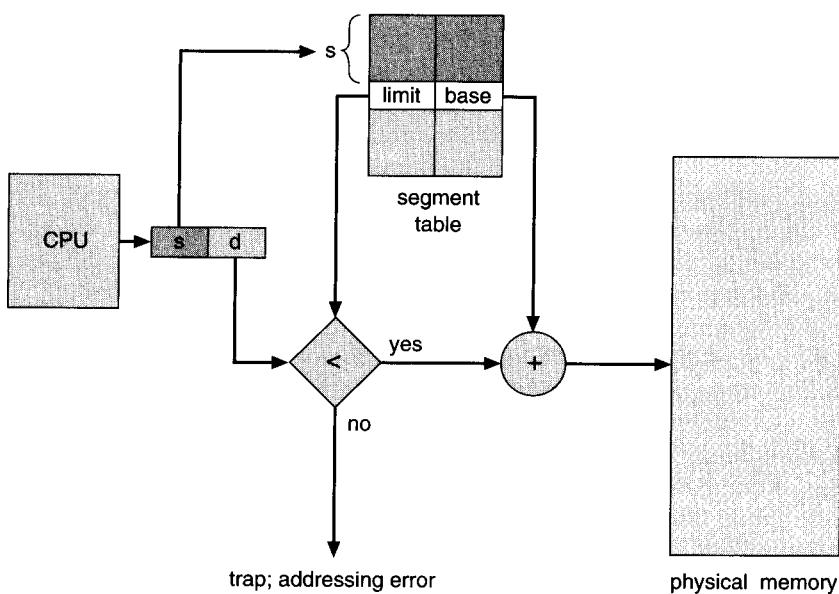


Figure 9.18 Segmentation hardware.

physical address where the segment resides in memory, whereas the segment limit specifies the length of the segment.

The use of a segment table is illustrated in Figure 9.18. A logical address consists of two parts: a segment number, s , and an offset into that segment, d . The segment number is used as an index into the segment table. The offset d of the logical address must be between 0 and the segment limit. If it is not, we trap to the operating system (logical addressing attempt beyond end of segment). If this offset is legal, it is added to the segment base to produce the address in physical memory of the desired byte. The segment table is thus essentially an array of base-limit register pairs.

As an example, consider the situation shown in Figure 9.19. We have five segments numbered from 0 through 4. The segments are stored in physical memory as shown. The segment table has a separate entry for each segment, giving the beginning address of the segment in physical memory (or base) and the length of that segment (or limit). For example, segment 2 is 400 bytes long

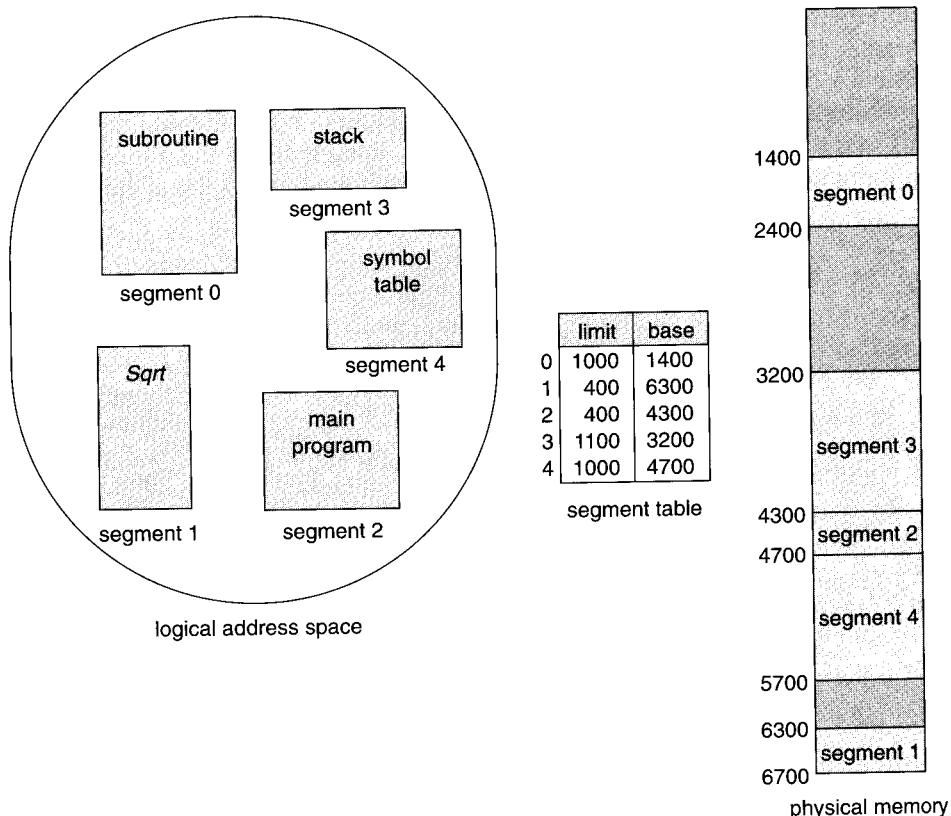


Figure 9.19 Example of segmentation.

and begins at location 4300. Thus, a reference to byte 53 of segment 2 is mapped onto location $4300 + 53 = 4353$. A reference to segment 3, byte 852, is mapped to 3200 (the base of segment 3) + 852 = 4052. A reference to byte 1222 of segment 0 would result in a trap to the operating system, as this segment is only 1,000 bytes long.

9.5.3 Protection and Sharing

A particular advantage of segmentation is the association of protection with the segments. Because the segments represent a semantically defined portion of the program, it is likely that all entries in the segment will be used the same way. Hence, some segments are instructions, whereas other segments are data. In a modern architecture, instructions are non-self-modifying, so instruction segments can be defined as read only or execute only. The memory-mapping hardware will check the protection bits associated with each segment-table entry to prevent illegal accesses to memory, such as attempts to write into a read-only segment, or to use an execute-only segment as data. By placing an array in its own segment, the memory-management hardware will automatically check that array indexes are legal and do not stray outside the array boundaries. Thus, many common program errors will be detected by the hardware before they can cause serious damage.

Another advantage of segmentation involves the *sharing* of code or data. Each process has a segment table associated with it, which the dispatcher uses to define the hardware segment table when this process is given the CPU. Segments are shared when entries in the segment tables of two different processes point to the same physical location (Figure 9.20).

The sharing occurs at the segment level. Thus, any information can be shared if it is defined to be a segment. Several segments can be shared, so a program composed of several segments can be shared.

For example, consider the use of a text editor in a time-sharing system. A complete editor might be quite large, composed of many segments. These segments can be shared among all users, limiting the physical memory needed to support editing tasks. Rather than n copies of the editor, we need only one copy. For each user, we still need separate, unique segments to store local variables. These segments, of course, would not be shared.

We can also share only parts of programs. For example, common subroutine packages can be shared among many users if they are defined as sharable, read-only segments. Two Fortran programs, for instance, may use the same *Sqrt* subroutine, but only one physical copy of the *Sqrt* routine would be needed.

Although this sharing appears simple, there are subtle considerations. Code segments typically contain references to themselves. For example, a conditional jump normally has a transfer address, which consists of a segment number and offset. The segment number of the transfer address will be the segment number of the code segment. If we try to share this segment, all

sharing processes must define the shared code segment to have the same segment number.

For instance, if we want to share the *Sqrt* routine, and one process wants to make it segment 4 and another wants to make it segment 17, how should the *Sqrt* routine refer to itself? Because there is only one physical copy of *Sqrt*, it must refer to itself in the same way for both users—it must have a unique segment number. As the number of users sharing the segment increases, so does the difficulty of finding an acceptable segment number.

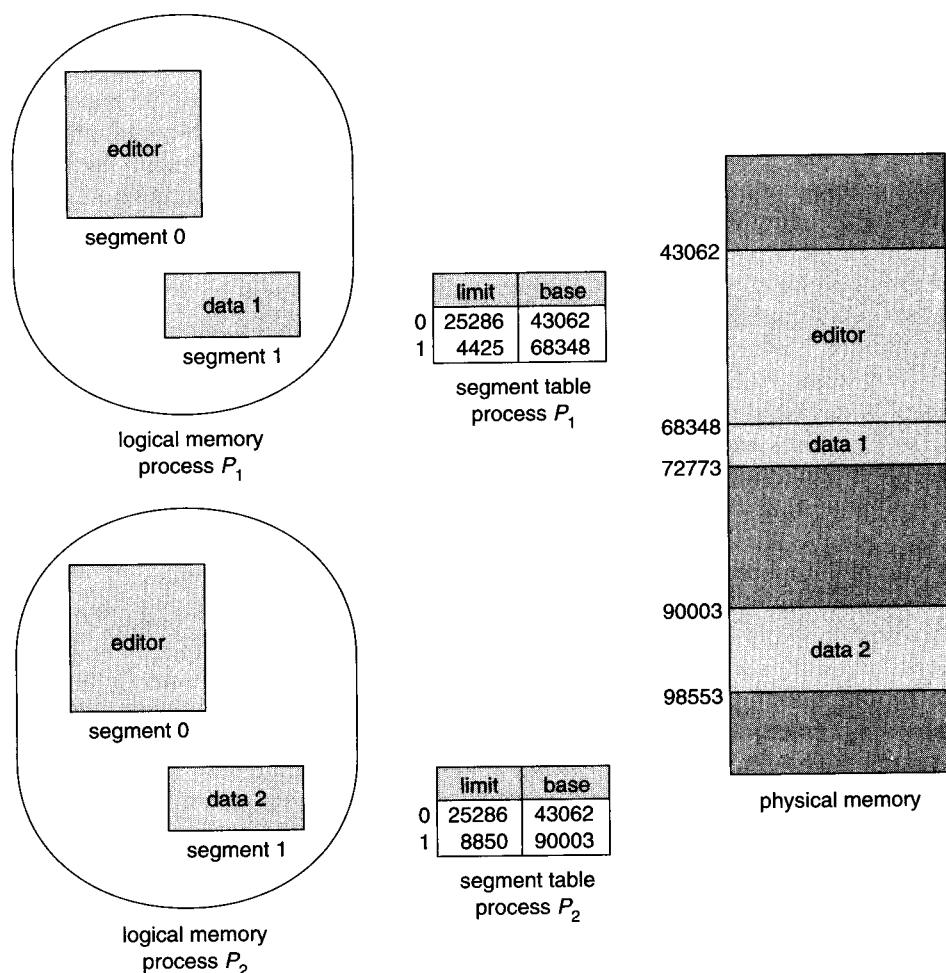


Figure 9.20 Sharing of segments in a segmented memory system.

Read-only data segments that contain no physical pointers may be shared as different segment numbers, as may code segments that refer to themselves not directly, but rather only indirectly. For example, conditional branches that specify the branch address as an offset from the current program counter or relative to a register containing the current segment number would allow code to avoid direct reference to the current segment number.

9.5.4 Fragmentation

The long-term scheduler must find and allocate memory for all the segments of a user program. This situation is similar to paging *except* that the segments are of *variable* length; pages are all the same size. Thus, as with the variable-sized partition scheme, memory allocation is a dynamic storage-allocation problem, usually solved with a best-fit or first-fit algorithm.

Segmentation may then cause external fragmentation, when all blocks of free memory are too small to accommodate a segment. In this case, the process may simply have to wait until more memory (or at least a larger hole) becomes available, or until compaction creates a larger hole. Because segmentation is by its nature a dynamic relocation algorithm, we can compact memory whenever we want. If the CPU scheduler must wait for one process, because of a memory-allocation problem, it may (or may not) skip through the CPU queue looking for a smaller, lower-priority process to run.

How serious a problem is external fragmentation for a segmentation scheme? Would long-term scheduling with compaction help? The answers depend mainly on the average segment size. At one extreme, we could define each process to be one segment. This approach reduces to the variable-sized partition scheme. At the other extreme, every byte could be put in its own segment and relocated separately. This arrangement eliminates external fragmentation altogether; however, every byte would need a base register for its relocation, doubling memory use! Of course, the next logical step—fixed-sized, small segments—is paging. Generally, if the average segment size is small, external fragmentation will also be small. (By analogy, consider putting suitcases in the trunk of a car; they never quite seem to fit. However, if you open the suitcases and put the individual items in the trunk, everything is more likely to fit.) Because the individual segments are smaller than the overall process, they are more likely to fit in the available memory blocks.

9.6 ■ Segmentation with Paging

Both paging and segmentation have advantages and disadvantages. In fact, of the two most popular microprocessors now being used, the Motorola 68000 line is designed based on a flat-address space, whereas the Intel 80x86 and Pentium family are based on segmentation. Both are merging memory models toward

a mixture of paging and segmentation. We can combine these two methods to improve on each. This combination is best illustrated by the architecture of the Intel 386.

The IBM OS/2 32-bit version is an operating system running on top of the Intel 386 (and later) architecture. The 386 uses segmentation with paging for memory management. The maximum number of segments per process is 16 KB, and each segment can be as large as 4 gigabytes. The page size is 4 KB. We shall not give a complete description of the memory-management structure of the 386 in this text. Rather, we shall present the major ideas.

The logical-address space of a process is divided into two partitions. The first partition consists of up to 8 KB segments that are private to that process. The second partition consists of up to 8 KB segments that are shared among all the processes. Information about the first partition is kept in the **local descriptor table (LDT)**, information about the second partition is kept in the **global descriptor table (GDT)**. Each entry in the LDT and GDT consists of 8 bytes, with detailed information about a particular segment including the base location and length of that segment.

The logical address is a pair (selector, offset), where the selector is a 16-bit number:

<i>s</i>	<i>g</i>	<i>p</i>
13	1	2

in which *s* designates the segment number, *g* indicates whether the segment is in the GDT or LDT, and *p* deals with protection. The offset is a 32-bit number specifying the location of the byte (or word) within the segment in question.

The machine has six segment registers, allowing six segments to be addressed at any one time by a process. It has six 8-byte microprogram registers to hold the corresponding descriptors from either the LDT or GDT. This cache lets the 386 avoid having to read the descriptor from memory for every memory reference.

The physical address on the 386 is 32 bits long and is formed as follows. The segment register points to the appropriate entry in the LDT or GDT. The base and limit information about the segment in question are used to generate a **linear address**. First, the limit is used to check for address validity. If the address is not valid, a memory fault is generated, resulting in a trap to the operating system. If it is valid, then the value of the offset is added to the value of the base, resulting in a 32-bit linear address. This address is then translated into a physical address.

As pointed out previously, each segment is paged, and each page is 4 KB. A page table may thus consist of up to 1 million entries. Because each entry consists of 4 bytes, each process may need up to 4 MB of physical-address space for the page table alone. Clearly, we would not want to allocate the page table contiguously in main memory. The solution adopted in the 386 is to use a two-level paging scheme. The linear address is divided into a page number

consisting of 20 bits, and a page offset consisting of 12 bits. Since we page the page table, the page number is further divided into a 10-bit page directory pointer and a 10-bit page table pointer. The logical address is as follows:

page number	page offset
p_1	d
10	12

The address-translation scheme for this architecture is similar to the scheme shown in Figure 9.13. The Intel address translation is shown in more detail in Figure 9.21. To improve the efficiency of physical-memory use, Intel 386 page tables can be swapped to disk. In this case, an invalid bit is used in the page-directory entry to indicate whether the table to which the entry is pointing is

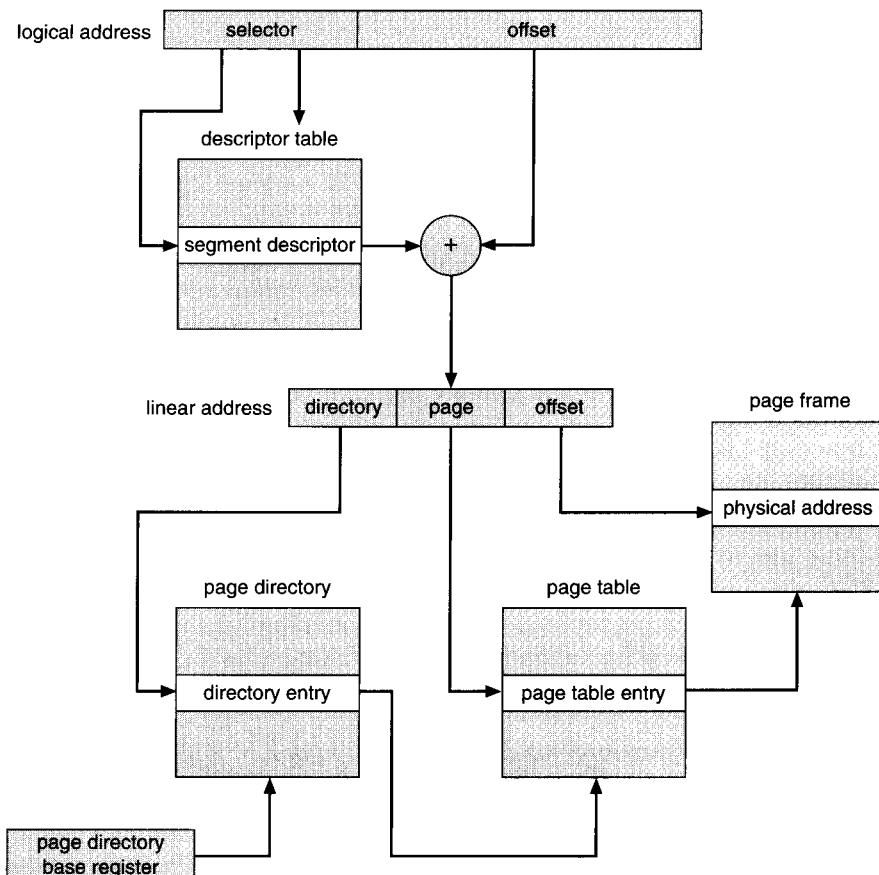


Figure 9.21 Intel 80386 address translation.

in memory or on disk. If the table is on disk, the operating system can use the other 31 bits to specify the disk location of the table; the table then can be brought into memory on demand.

9.7 ■ Summary

Memory-management algorithms for multiprogrammed operating systems range from the simple single-user system approach to paged segmentation. The greatest determinant of the method used in a particular system is the hardware provided. Every memory address generated by the CPU must be checked for legality and possibly mapped to a physical address. The checking cannot be implemented (efficiently) in software. Hence, we are constrained by the hardware available.

The memory-management algorithms discussed (contiguous allocation, paging, segmentation, and combinations of paging and segmentation) differ in many aspects. In comparing different memory-management strategies, you should use the following considerations:

- **Hardware support:** A simple base register or a pair of base and limit registers is sufficient for the single- and multiple-partition schemes, whereas paging and segmentation need mapping tables to define the address map.
- **Performance:** As the memory-management algorithm becomes more complex, the time required to map a logical address to a physical address increases. For the simple systems, we need only to compare or add to the logical address—operations that are fast. Paging and segmentation can be as fast if the table is implemented in fast registers. If the table is in memory, however, user memory accesses can be degraded substantially. A TLB can reduce the performance degradation to an acceptable level.
- **Fragmentation:** A multiprogrammed system will generally perform more efficiently if it has a higher level of multiprogramming. For a given set of processes, we can increase the multiprogramming level only by packing more processes into memory. To accomplish this task, we must reduce memory waste or fragmentation. Systems with fixed-sized allocation units, such as the single-partition scheme and paging, suffer from internal fragmentation. Systems with variable-sized allocation units, such as the multiple-partition scheme and segmentation, suffer from external fragmentation.
- **Relocation:** One solution to the external-fragmentation problem is compaction. Compaction involves shifting a program in memory without the program noticing the change. This consideration requires that logical addresses be relocated dynamically, at execution time. If addresses are relocated only at load time, we cannot compact storage.

- **Swapping:** Any algorithm can have swapping added to it. At intervals determined by the operating system, usually dictated by CPU-scheduling policies, processes are copied from main memory to a backing store, and later are copied back to main memory. This scheme allows more processes to be run than can be fit into memory at one time.
- **Sharing:** Another means of increasing the multiprogramming level is to share code and data among different users. Sharing generally requires that either paging or segmentation be used, to provide small packets of information (pages or segments) that can be shared. Sharing is a means of running many processes with a limited amount of memory, but shared programs and data must be designed carefully.
- **Protection:** If paging or segmentation is provided, different sections of a user program can be declared execute only, read only, or read-write. This restriction is necessary with shared code or data, and is generally useful in any case to provide simple run-time checks for common programming errors.

■ Exercises

- 9.1 Name two differences between logical and physical addresses.
- 9.2 Explain the difference between internal and external fragmentation.
- 9.3 Describe the following allocation algorithms:
 - a. First fit
 - b. Best fit
 - c. Worst fit
- 9.4 When a process is rolled out of memory, it loses its ability to use the CPU (at least for a while). Describe another situation where a process loses its ability to use the CPU, but where the process does not get rolled out.
- 9.5 Given memory partitions of 100 KB, 500 KB, 200 KB, 300 KB, and 600 KB (in order), how would each of the first-fit, best-fit, and worst-fit algorithms place processes of 212 KB, 417 KB, 112 KB, and 426 KB (in order)? Which algorithm makes the most efficient use of memory?
- 9.6 Consider a system where a program can be separated into two parts: code and data. The CPU knows whether it wants an instruction (instruction fetch) or data (data fetch or store). Therefore, two base-limit register pairs are provided: one for instructions and one for data. The instruction base-limit register pair is automatically set to read only, so programs can

be shared among different users. Discuss the advantages and disadvantages of this scheme.

- 9.7 Why are page sizes always powers of 2?
- 9.8 Consider a logical-address space of eight pages of 1,024 words each, mapped onto a physical memory of 32 frames.
 - a. How many bits are in the logical address?
 - b. How many bits are in the physical address?
- 9.9 On a system with paging, a process cannot access memory that it does not own; why? How could the operating system allow access to other memory? Why should it or should it not?
- 9.10 Consider a paging system with the page table stored in memory.
 - a. If a memory reference takes 200 nanoseconds, how long does a paged memory reference take?
 - b. If we add TLBs, and 75 percent of all page-table references are found in the TLBs, what is the effective memory reference time? (Assume that finding a page-table entry in the TLBs takes zero time, if the entry is there.)
- 9.11 What is the effect of allowing two entries in a page table to point to the same page frame in memory? Explain how you could use this effect to decrease the amount of time needed to copy a large amount of memory from one place to another. What would the effect of updating some byte in the one page be on the other page?
- 9.12 Why are segmentation and paging sometimes combined into one scheme?
- 9.13 Describe a mechanism by which one segment could belong to the address space of two different processes.
- 9.14 Explain why it is easier to share a reentrant module using segmentation than it is to do so when pure paging is used.
- 9.15 Sharing segments among processes without requiring the same segment number is possible in a dynamically linked segmentation system.
 - a. Define a system that allows static linking and sharing of segments without requiring that the segment numbers be the same.
 - b. Describe a paging scheme that allows pages to be shared without requiring that the page numbers be the same.
- 9.16 Consider the following segment table:

Segment	Base	Length
0	219	600
1	2300	14
2	90	100
3	1327	580
4	1952	96

What are the physical addresses for the following logical addresses?

- a. 0430
- b. 110
- c. 2500
- d. 3400
- e. 4112

9.17 Consider the Intel address-translation scheme shown in Figure 9.21.

- a. Describe all the steps that are taken by the Intel 80386 in translating a logical address into a physical address.
- b. What are the advantages to the operating system of hardware that provides such complicated memory-translation hardware?
- c. Are there any disadvantages to this address-translation system? If so, what are they? If not, why is it not used by every manufacturer?

9.18 In the IBM/370, memory protection is provided through the use of *keys*. A key is a 4-bit quantity. Each 2 KB block of memory has a key (the storage key) associated with it. The CPU also has a key (the protection key) associated with it. A store operation is allowed only if both keys are equal, or if either is zero. Which of the following memory-management schemes could be used successfully with this hardware?

- a. Bare machine
- b. Single-user system
- c. Multiprogramming with a fixed number of processes
- d. Multiprogramming with a variable number of processes
- e. Paging
- f. Segmentation

Bibliographical Notes

Dynamic storage allocation was discussed by Knuth [1973] (Section 2.5), who found through simulation results that first fit is generally superior to best fit. Knuth [1973] discussed the 50-percent rule.

The concept of paging can be credited to the designers of the Atlas system, which has been described by Kilburn et al. [1961] and by Howarth et al. [1961]. The concept of segmentation was first discussed by Dennis [1965]. Paged segmentation was first supported in the GE 645, on which MULTICS was originally implemented (Organick [1972]).

Inverted page tables were discussed in an article about the IBM RT storage manager by Chang and Mergen [1988].

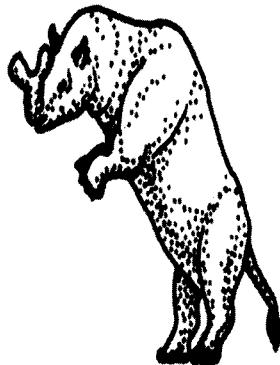
Address translation in software is covered in Jacob and Mudge [1997].

Cache memories, including associative memory, were described and analyzed by Smith [1982]. This paper also includes an extensive bibliography on the subject. Hennessy and Patterson [1996] discussed the hardware aspects of TLBs, caches, and MMUs. Talluri et al. [1995] discusses page tables for 64-bit address spaces. Dougan et al. [1999] discusses a technique for managing the TLB.

The Motorola 68000 microprocessor family was described in Motorola [1989a]. Information on the paging hardware of 80386 can be found in Intel [1986]. Tanenbaum [2001] also discussed Intel 80386 paging. The Intel 80486 hardware also is described in an Intel publication Intel [1989].

Memory management for several architectures—such as the Pentium II, PowerPC, and UltraSPARC—was described by Jacob and Mudge [1998a].

Chapter 10



VIRTUAL MEMORY

In Chapter 9, we discussed various memory-management strategies that are used in computer systems. All these strategies have the same goal: to keep many processes in memory simultaneously to allow multiprogramming. However, they tend to require the entire process to be in memory before the process can execute.

Virtual memory is a technique that allows the execution of processes that may not be completely in memory. One major advantage of this scheme is that programs can be larger than physical memory. Further, virtual memory abstracts main memory into an extremely large, uniform array of storage, separating logical memory as viewed by the user from physical memory. This technique frees programmers from the concerns of memory-storage limitations. Virtual memory also allows processes to easily share files and address spaces, and it provides an efficient mechanism for process creation.

Virtual memory is not easy to implement, however, and may substantially decrease performance if it is used carelessly. In this chapter, we discuss virtual memory in the form of demand paging, and examine its complexity and cost.

10.1 ■ Background

The memory-management algorithms outlined in Chapter 9 are necessary because of one basic requirement: The instructions being executed must be in physical memory. The first approach to meeting this requirement is to place the entire logical address space in physical memory. Overlays and dynamic loading

can help to ease this restriction, but they generally require special precautions and extra work by the programmer. This restriction seems both necessary and reasonable, but it is also unfortunate, since it limits the size of a program to the size of physical memory.

In fact, an examination of real programs shows us that, in many cases, the entire program is not needed. For instance,

- Programs often have code to handle unusual error conditions. Since these errors seldom, if ever, occur in practice, this code is almost never executed.
- Arrays, lists, and tables are often allocated more memory than they actually need. An array may be declared 100 by 100 elements, even though it is seldom larger than 10 by 10 elements. An assembler symbol table may have room for 3,000 symbols, although the average program has less than 200 symbols.
- Certain options and features of a program may be used rarely. For instance, the routines on U.S. government computers that balance the budget have only recently been used.

Even in those cases where the entire program is needed, it may not all be needed at the same time (such is the case with overlays, for example).

The ability to execute a program that is only partially in memory would confer many benefits:

- A program would no longer be constrained by the amount of physical memory that is available. Users would be able to write programs for an extremely large **virtual-address space**, simplifying the programming task.
- Because each user program could take less physical memory, more programs could be run at the same time, with a corresponding increase in CPU utilization and throughput, but with no increase in response time or turnaround time.
- Less I/O would be needed to load or swap each user program into memory, so each user program would run faster.

Thus, running a program that is not entirely in memory would benefit both the system and the user.

Virtual memory is the separation of user logical memory from physical memory. This separation allows an extremely large virtual memory to be provided for programmers when only a smaller physical memory is available (Figure 10.1). Virtual memory makes the task of programming much easier, because the programmer no longer needs to worry about the amount of physical memory available, or about what code can be placed in overlays; she can concentrate

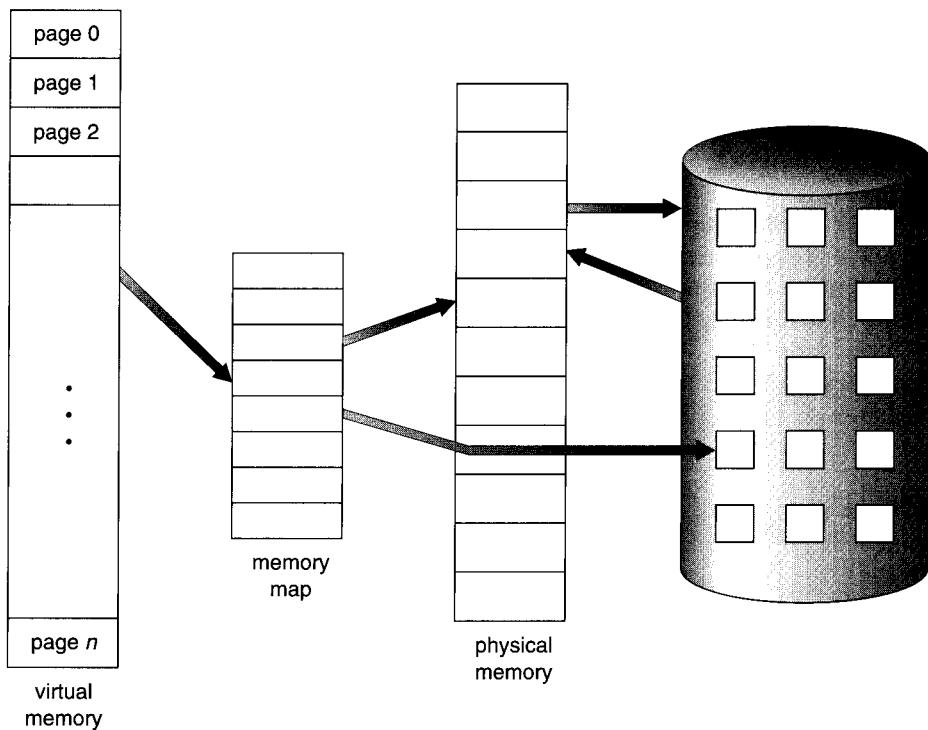


Figure 10.1 Diagram showing virtual memory that is larger than physical memory.

instead on the problem to be programmed. On systems that support virtual memory, overlays have almost disappeared.

In addition to separating logical memory from physical memory, virtual memory also allows files and memory to be shared by several different processes through page sharing (Section 9.4.5). The sharing of pages further allows performance improvements during process creation.

Virtual memory is commonly implemented by **demand paging**. It can also be implemented in a segmentation system. Several systems provide a paged segmentation scheme, where segments are broken into pages. Thus, the user view is segmentation, but the operating system can implement this view with demand paging. **Demand segmentation** can also be used to provide virtual memory. Burroughs' computer systems have used demand segmentation. The IBM OS/2 operating system also uses demand segmentation. However, segment-replacement algorithms are more complex than are page-replacement algorithms because the segments have variable sizes. We do not cover demand segmentation in this text; refer to the Bibliographical Notes for relevant references.

10.2 ■ Demand Paging

A demand-paging system is similar to a paging system with swapping (Figure 10.2). Processes reside on secondary memory (which is usually a disk). When we want to execute a process, we swap it into memory. Rather than swapping the entire process into memory, however, we use a **lazy swapper**. A lazy swapper never swaps a page into memory unless that page will be needed. Since we are now viewing a process as a sequence of pages, rather than as one large contiguous address space, use of *swap* is technically incorrect. A swapper manipulates entire processes, whereas a **pager** is concerned with the individual pages of a process. We thus use *pager*, rather than *swapper*, in connection with demand paging.

10.2.1 Basic Concepts

When a process is to be swapped in, the pager guesses which pages will be used before the process is swapped out again. Instead of swapping in a whole process, the pager brings only those necessary pages into memory. Thus, it avoids reading into memory pages that will not be used anyway, decreasing the swap time and the amount of physical memory needed.

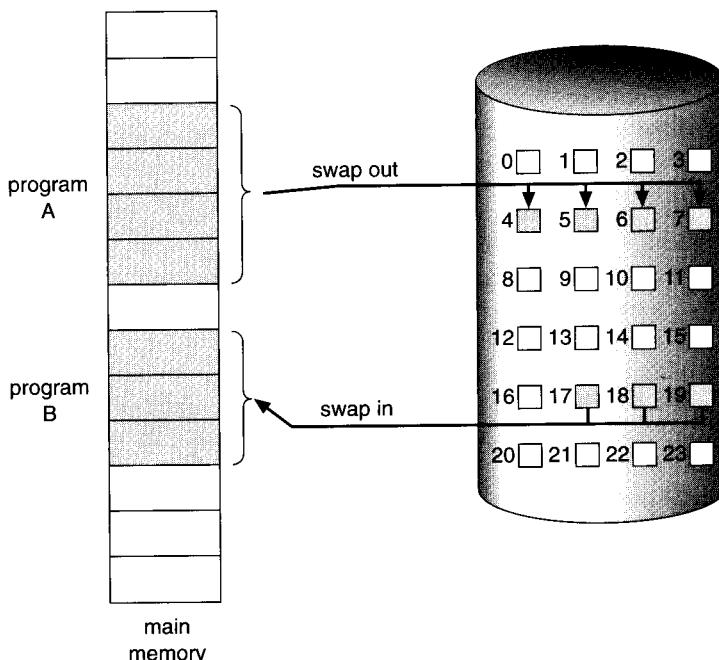


Figure 10.2 Transfer of a paged memory to contiguous disk space.

With this scheme, we need some form of hardware support to distinguish between those pages that are in memory and those pages that are on the disk. The valid–invalid bit scheme described in Section 9.4.4 can be used for this purpose. This time, however, when this bit is set to “valid,” this value indicates that the associated page is both legal and in memory. If the bit is set to “invalid,” this value indicates that the page either is not valid (that is, not in the logical address space of the process), or is valid but is currently on the disk. The page-table entry for a page that is brought into memory is set as usual, but the page-table entry for a page that is not currently in memory is simply marked invalid, or contains the address of the page on disk. This situation is depicted in Figure 10.3.

Notice that marking a page invalid will have no effect if the process never attempts to access that page. Hence, if we guess right and page in all and only those pages that are actually needed, the process will run exactly as though we

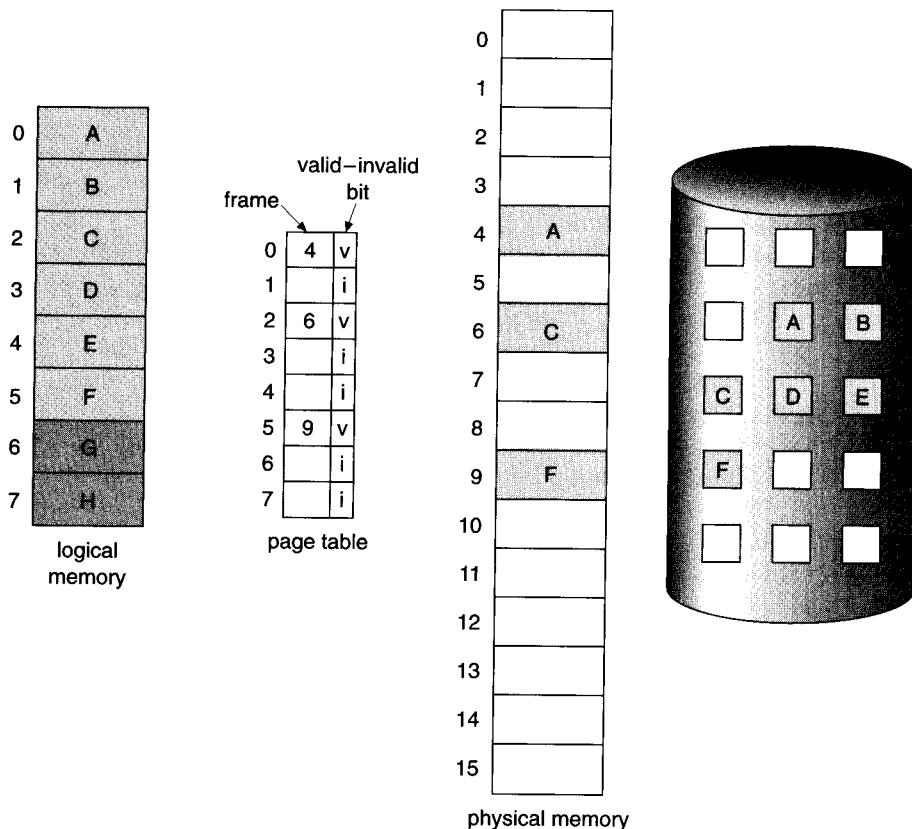


Figure 10.3 Page table when some pages are not in main memory.

had brought in all pages. While the process executes and accesses pages that are **memory resident**, execution proceeds normally.

But what happens if the process tries to access a page that was not brought into memory? Access to a page marked invalid causes a **page-fault trap**. The paging hardware, in translating the address through the page table, will notice that the invalid bit is set, causing a trap to the operating system. This trap is the result of the operating system's failure to bring the desired page into memory (in an attempt to minimize disk-transfer overhead and memory requirements), rather than an invalid address error as a result of an attempt to use an illegal memory address (such as an incorrect array subscript). We must therefore correct this oversight. The procedure for handling this page fault is straightforward (Figure 10.4):

1. We check an internal table (usually kept with the process control block) for this process, to determine whether the reference was a valid or invalid memory access.

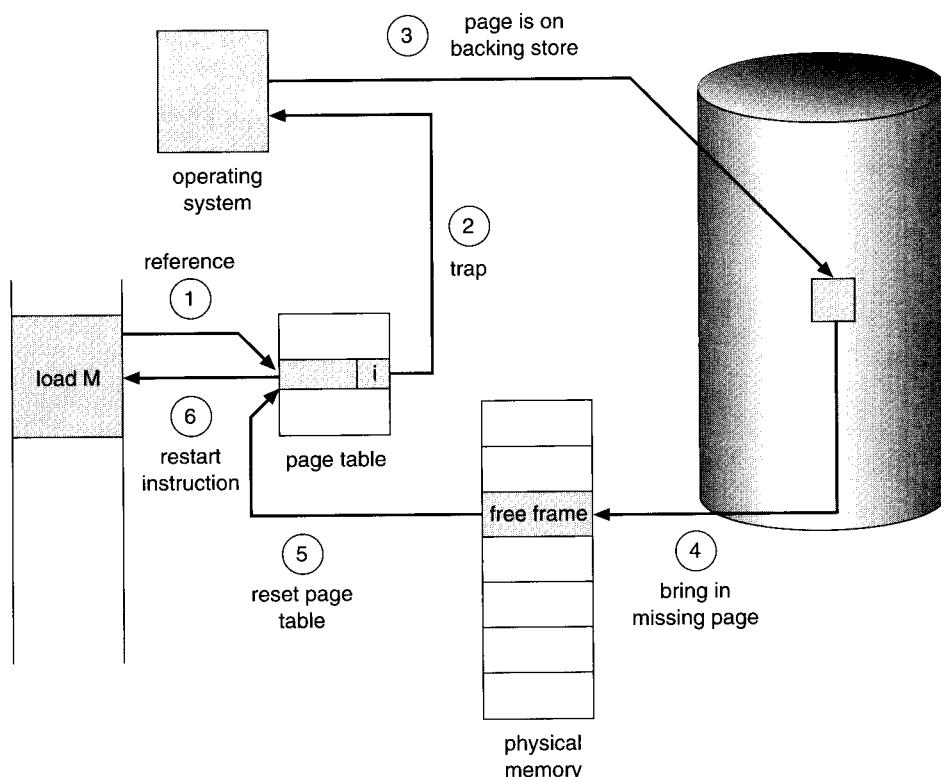


Figure 10.4 Steps in handling a page fault.

2. If the reference was invalid, we terminate the process. If it was valid, but we have not yet brought in that page, we now page it in.
3. We find a free frame (by taking one from the free-frame list, for example).
4. We schedule a disk operation to read the desired page into the newly allocated frame.
5. When the disk read is complete, we modify the internal table kept with the process and the page table to indicate that the page is now in memory.
6. We restart the instruction that was interrupted by the illegal address trap. The process can now access the page as though it had always been in memory.

It is important to realize that, because we save the state (registers, condition code, instruction counter) of the interrupted process when the page fault occurs, we can restart the process in *exactly* the same place and state, except that the desired page is now in memory and is accessible. In this way, we are able to execute a process, even though portions of it are not (yet) in memory. When the process tries to access locations that are not in memory, the hardware traps to the operating system (page fault). The operating system reads the desired page into memory and restarts the process as though the page had always been in memory.

In the extreme case, we could start executing a process with *no* pages in memory. When the operating system sets the instruction pointer to the first instruction of the process, which is on a non-memory-resident page, the process immediately faults for the page. After this page is brought into memory, the process continues to execute, faulting as necessary until every page that it needs is in memory. At that point, it can execute with no more faults. This scheme is **pure demand paging**: Never bring a page into memory until it is required.

Theoretically, some programs may access several new pages of memory with each instruction execution (one page for the instruction and many for data), possibly causing multiple page faults per instruction. This situation would result in unacceptable system performance. Fortunately, analysis of running processes shows that this behavior is exceedingly unlikely. Programs tend to have **locality of reference**, described in Section 10.6.1, which results in reasonable performance from demand paging.

The hardware to support demand paging is the same as the hardware for paging and swapping:

- **Page table:** This table has the ability to mark an entry invalid through a valid–invalid bit or special value of protection bits.
- **Secondary memory:** This memory holds those pages that are not present in main memory. The secondary memory is usually a high-speed disk. It is known as the swap device, and the section of disk used for this purpose is known as **swap space**. Swap-space allocation is discussed in Chapter 14.

In addition to this hardware support, considerable software is needed, as we shall see. Additional architectural constraints must be imposed. A crucial one is the need to be able to restart any instruction after a page fault. In most cases, this requirement is easy to meet. A page fault could occur at any memory reference. If the page fault occurs on the instruction fetch, we can restart by fetching the instruction again. If a page fault occurs while we are fetching an operand, we must fetch and decode the instruction again, and then fetch the operand.

As a worst-case example, consider a three-address instruction such as ADD the content of A to B placing the result in C. These are the steps to execute this instruction:

1. Fetch and decode the instruction (ADD).
2. Fetch A.
3. Fetch B.
4. Add A and B.
5. Store the sum in C.

If we faulted when we tried to store in C (because C is in a page not currently in memory), we would have to get the desired page, bring it in, correct the page table, and restart the instruction. The restart would require fetching the instruction again, decoding it again, fetching the two operands again, and then adding again. However, there is not much repeated work (less than one complete instruction), and the repetition is necessary only when a page fault occurs.

The major difficulty occurs when one instruction may modify several different locations. For example, consider the IBM System 360/370 MVC (move character) instruction, which can move up to 256 bytes from one location to another (possibly overlapping) location. If either block (source or destination) straddles a page boundary, a page fault might occur after the move is partially done. In addition, if the source and destination blocks overlap, the source block may have been modified, in which case we cannot simply restart the instruction.

This problem can be solved in two different ways. In one solution, the microcode computes and attempts to access both ends of both blocks. If a page fault is going to occur, it will happen at this step, before anything is modified. The move can then take place, as we know that no page fault can occur, since all the relevant pages are in memory. The other solution uses temporary registers to hold the values of overwritten locations. If there is a page fault, all the old values are written back into memory before the trap occurs. This action restores memory to its state before the instruction was started, so that the instruction can be repeated.

A similar architectural problem occurs in machines that use special addressing modes, including autodecrement and autoincrement modes (for example, the PDP-11). These addressing modes use a register as a pointer and automatically decrement or increment the register as indicated. Autodecrement automatically decrements the register *before* using its contents as the operand address; autoincrement automatically increments the register *after* using its contents as the operand address. Thus, the instruction

MOV (R2)+, -(R3)

copies the contents of the location pointed to by *register*₂ into the location pointed to by *register*₃. *Register*₂ is incremented (by two for a word, since the PDP-11 is a byte-addressable computer) after it is used as a pointer; *register*₃ is decremented (by two) before it is used as a pointer. Now consider what will happen if we get a fault when trying to store into the location pointed to by *register*₃. To restart the instruction, we must reset the two registers to the values they had before we started the execution of the instruction. One solution is to create a new special status register to record the register number and amount modified for any register that is changed during the execution of an instruction. This status register allows the operating system to *undo* the effects of a partially executed instruction that causes a page fault.

These are by no means the only architectural problems resulting from adding paging to an existing architecture to allow demand paging, but they illustrate some of the difficulties. Paging is added between the CPU and the memory in a computer system. It should be entirely transparent to the user process. Thus, people often assume that paging could be added to any system. Although this assumption is true for a non-demand-paging environment, where a page fault represents a fatal error, it is not true where a page fault means only that an additional page must be brought into memory and the process restarted.

10.2.2 Performance of Demand Paging

Demand paging can have a significant effect on the performance of a computer system. To see why, let us compute the **effective access time** for a demand-paged memory. For most computer systems, the memory-access time, denoted *ma*, now ranges from 10 to 200 nanoseconds. As long as we have no page faults, the effective access time is equal to the memory access time. If, however, a page fault occurs, we must first read the relevant page from disk, and then access the desired word.

Let *p* be the probability of a page fault ($0 \leq p \leq 1$). We would expect *p* to be close to zero; that is, there will be only a few page faults. The **effective access time** is then

$$\text{effective access time} = (1 - p) \times ma + p \times \text{page fault time}.$$

To compute the effective access time, we must know how much time is needed to service a page fault. A page fault causes the following sequence to occur:

1. Trap to the operating system.
2. Save the user registers and process state.
3. Determine that the interrupt was a page fault.
4. Check that the page reference was legal and determine the location of the page on the disk.
5. Issue a read from the disk to a free frame:
 - a. Wait in a queue for this device until the read request is serviced.
 - b. Wait for the device seek and/or latency time.
 - c. Begin the transfer of the page to a free frame.
6. While waiting, allocate the CPU to some other user (CPU scheduling; optional).
7. Interrupt from the disk (I/O completed).
8. Save the registers and process state for the other user (if step 6 is executed).
9. Determine that the interrupt was from the disk.
10. Correct the page table and other tables to show that the desired page is now in memory.
11. Wait for the CPU to be allocated to this process again.
12. Restore the user registers, process state, and new page table, then resume the interrupted instruction.

Not all of these steps are necessary in every case. For example, we are assuming that, in step 6, the CPU is allocated to another process while the I/O occurs. This arrangement allows multiprogramming to maintain CPU utilization, but requires additional time to resume the page-fault service routine when the I/O transfer is complete.

In any case, we are faced with three major components of the page-fault service time:

1. Service the page-fault interrupt.
2. Read in the page.
3. Restart the process.

The first and third tasks may be reduced, with careful coding, to several hundred instructions. These tasks may take from 1 to 100 microseconds each. The page-switch time, on the other hand, will probably be close to 24 milliseconds. A typical hard disk has an average latency of 8 milliseconds, a seek of 15 milliseconds, and a transfer time of 1 millisecond. Thus, the total paging time would be close to 25 milliseconds, including hardware and software time. Remember also that we are looking at only the device-service time. If a queue of processes is waiting for the device (other processes that have caused page faults), we have to add device-queueing time as we wait for the paging device to be free to service our request, increasing even more the time to swap.

If we take an average page-fault service time of 25 milliseconds and a memory-access time of 100 nanoseconds, then the effective access time in nanoseconds is

$$\begin{aligned}\text{effective access time} &= (1 - p) \times (100) + p (25 \text{ milliseconds}) \\ &= (1 - p) \times 100 + p \times 25,000,000 \\ &= 100 + 24,999,900 \times p.\end{aligned}$$

We see then that the effective access time is directly proportional to the **page-fault rate**. If one access out of 1,000 causes a page fault, the effective access time is 25 microseconds. The computer would be slowed down by a factor of 250 because of demand paging! If we want less than 10-percent degradation, we need

$$\begin{aligned}110 &> 100 + 25,000,000 \times p, \\ 10 &> 25,000,000 \times p, \\ p &< 0.0000004.\end{aligned}$$

That is, to keep the slowdown due to paging to a reasonable level, we can allow only less than one memory access out of 2,500,000 to page fault.

It is important to keep the page-fault rate low in a demand-paging system. Otherwise, the effective access time increases, slowing process execution dramatically.

One additional aspect of demand paging is the handling and overall use of swap space. Disk I/O to swap space is generally faster than that to the file system. It is faster because swap space is allocated in much larger blocks, and file lookups and indirect allocation methods are not used (Chapter 14). It is therefore possible for the system to gain better paging throughput, by copying an entire file image into the swap space at process startup, and then performing demand paging from the swap space. Another option is to demand pages from the file system initially, but to write the pages to swap space as they are replaced. This approach will ensure that only needed pages are ever read from the file system, but all subsequent paging is done from swap space.

Some systems attempt to limit the amount of swap space when binary files are used. Demand pages for such files are brought directly from the file system. However, when page replacement is called for, these pages can simply be overwritten (because they are never modified) and read in from the file system again if needed. Using this approach, the file system itself serves as the backing store. However, swap space must still be used for pages not associated with a file; these pages include the **stack** and **heap** for a process. This technique is used in several systems including Solaris 2. This method appears to be a good compromise; it is used in BSD UNIX.

10.3 ■ Process Creation

In Section 10.2, we illustrated how a process is first started using demand paging. Using this technique, a process can start quickly by merely demand paging in the page containing the first instruction. However, paging and virtual memory can also provide for other benefits during process creation. In this section, we will explore two techniques made available by virtual memory that enhance performance creating and running processes.

10.3.1 Copy-on-Write

Demand paging is used when reading a file from disk into memory and such files may include binary executables. However, process creation using the `fork()` system call may initially bypass the need for demand paging by using a technique similar to page sharing (covered in Section 9.4.5). This technique provides for rapid process creation and minimizes the number of new pages that must be allocated to the newly created process.

Recall the `fork()` system call creates a child process as a duplicate of its parent. Traditionally `fork()` worked by creating a copy of the parent's address space for the child, duplicating the pages belonging to the parent. However, considering that many child processes invoke the `exec()` system call immediately after creation, the copying of the parent's address space may be unnecessary. Alternatively, we can use a technique known as **copy-on-write**. This works by allowing the parent and child processes to initially share the same pages. These shared pages are marked as copy-on-write pages, meaning that if either process writes to a shared page, a copy of the shared page is created. For example, assume the child process attempts to modify a page containing portions of the stack; the operating system recognizes this as a copy-on-write page. The operating system will then create a copy of this page, mapping it to the address space of the child process. Therefore, the child process will modify its copied page and not the page belonging to the parent process. Using the copy-on-write technique, it is obvious that only the pages that are modified by either process are copied; all non-modified pages may

be shared by the parent and child processes. Note that only pages that may be modified need be marked as copy-on-write. Pages that cannot be modified (i.e., pages containing executable code) may be shared by the parent and child. Copy-on-write is a common technique used by several operating systems when duplicating processes, including Windows 2000, Linux, and Solaris 2.

When it is determined a page is going to be duplicated using copy-on-write, it is important to note where the free page will be allocated from. Many operating systems provide a **pool** of free pages for such requests. These free pages are typically allocated when the stack or heap for a process must expand or for managing copy-on-write pages. Operating systems typically allocate these pages using a technique known as **zero-fill-on-demand**. Zero-fill-on-demand pages have been zeroed-out before being allocated, thus erasing the previous contents of the page. With copy-on-write, the page being copied will be copied to a zero-filled page. Pages allocated for the stack or heap are similarly assigned zero-filled pages.

Several versions of UNIX (including Solaris 2) also provide a variation of the `fork()` system call—`vfork()` (for **virtual memory fork**). `vfork()` operates differently than `fork()` with copy-on-write. With `vfork()` the parent process is suspended and the child process uses the address space of the parent. Because `vfork()` does not use copy-on-write, if the child process changes any pages of the parent's address space, the altered pages will be visible to the parent once it resumes. Therefore, `vfork()` must be used with caution, ensuring that the child process does not modify the address space of the parent. `vfork()` is intended to be used when the child process calls `exec()` immediately after creation. Because no copying of pages takes place, `vfork()` is an extremely efficient method of process creation and is sometimes used to implement UNIX command-line shell interfaces.

10.3.2 Memory-Mapped Files

Consider a sequential read of a file on disk using the standard system calls `open()`, `read()`, and `write()`. Every time the file is accessed requires a system call and disk access. Alternatively, we can use the virtual-memory techniques discussed so far to treat file I/O as routine memory accesses. This approach is known as **memory mapping** a file, allowing a part of the virtual-address space to be logically associated with a file. Memory mapping a file is possible by mapping a disk block to a page (or page(s)) in memory. Initial access to the file proceeds using ordinary demand paging, resulting in a page fault. However, a page-sized portion of the file is read from the file system into a physical page (some systems may in fact opt to read in more than a page-sized chunk of memory at a time.) Subsequent reads and writes to the file are handled as routine memory accesses, thereby simplifying file access and usage by allowing file manipulation through memory rather than the overhead of using the `read()` and `write()` system calls. Note that writes to the file

mapped in memory may not necessarily be immediate writes to the file on disk. Some systems may choose to update the physical file when the operating system periodically checks if the page in memory mapping the file has been modified. Closing the file results in all the memory-mapped data being written back to disk and removed from the virtual memory of the process.

Some operating systems provide memory mapping only through a specific system call and treat all other file I/O using the standard system calls. However, some systems choose to memory map a file regardless of whether a file was specified as memory mapped or not. Let's take Solaris 2 as an example. If a file is specified as memory-mapped (using the `mmap()` system call), Solaris 2 maps the file into the address space of the process. However, if a file is opened and accessed using ordinary system calls such as `open()`, `read()`, and `write()`, Solaris 2 still memory maps the file, however mapping it to the kernel address space. Regardless how the file is opened, Solaris 2 treats all file I/O as memory-mapped, allowing file access to take place in memory.

Multiple processes may be allowed to map the same file into the virtual memory of each, to allow sharing of data. Writes by any of the processes modify the data in virtual memory and can be seen by all others that map the same section of the file. Given our knowledge of virtual memory, it should be clear how the sharing of memory-mapped sections of memory is implemented: The virtual-memory map of each sharing process points to the same page of physical memory—the page that holds a copy of the disk block. This memory sharing is illustrated in Figure 10.5. The memory-mapping system calls can also support copy-on-write functionality, allowing processes to share a file in read-only mode, but to have their own copies of any data they modify. So that access to the shared data is coordinated, the processes involved might use one of the mechanisms for achieving mutual exclusion described in Chapter 7.

10.4 ■ Page Replacement

In our presentation so far, the page-fault rate has not been a serious problem, because each page has faulted at most once, when it is first referenced. This representation is not strictly accurate. If a process of ten pages actually uses only one-half of them, then demand paging saves the I/O necessary to load the five pages that are never used. We could also increase our degree of multiprogramming by running twice as many processes. Thus, if we had 40 frames, we could run eight processes, rather than the four that could run if each required 10 frames (five of which were never used).

If we increase our degree of multiprogramming, we are **over-allocating** memory. If we run six processes, each of which is ten pages in size, but actually uses only five pages, we have higher CPU utilization and throughput, with 10 frames to spare. It is possible, however, that each of these processes, for a particular data set, may suddenly try to use all ten of its pages, resulting in a

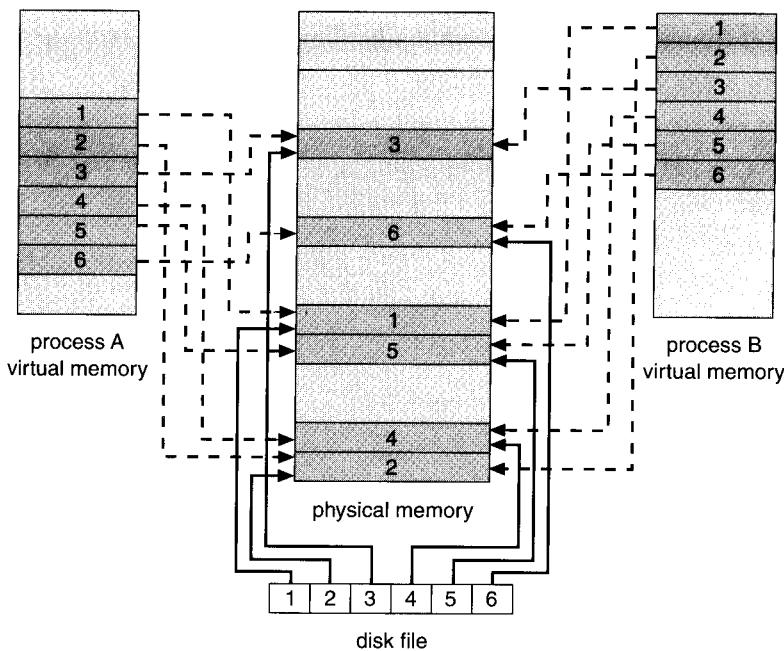


Figure 10.5 Memory-mapped files.

need for 60 frames, when only 40 are available. Although this situation may be unlikely, it becomes much more likely as we increase the multiprogramming level, so that the average memory usage is close to the available physical memory. (In our example, why stop at a multiprogramming level of six, when we can move to a level of seven or eight?)

Further, consider that system memory is not used only for holding program pages. Buffers for I/O also consume a significant amount of memory. This use can increase the strain on memory-placement algorithms. Deciding how much memory to allocate to I/O and how much to program pages is a significant challenge. Some systems allocate a fixed percentage of memory for I/O buffers, whereas others allow both user processes and the I/O subsystem to compete for all system memory.

Over-allocation manifests itself as follows. While a user process is executing, a page fault occurs. The hardware traps to the operating system, which checks its internal tables to see that this page fault is a genuine one rather than an illegal memory access. The operating system determines where the desired page is residing on the disk, but then finds that there are *no* free frames on the free-frame list: All memory is in use (Figure 10.6).

The operating system has several options at this point. It could terminate the user process. However, demand paging is the operating system's attempt to improve the computer system's utilization and throughput. Users should not

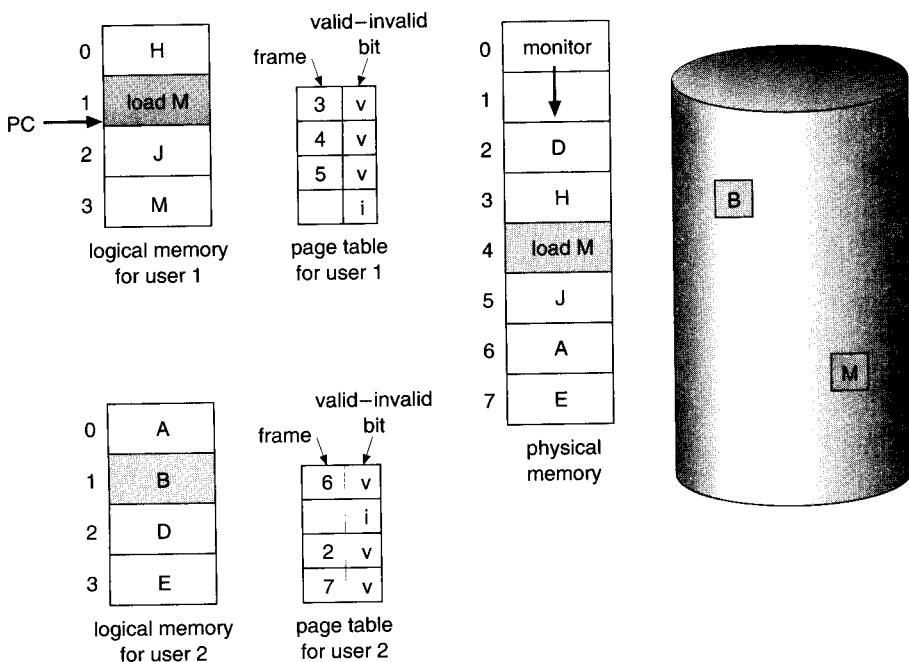


Figure 10.6 Need for page replacement.

be aware that their processes are running on a paged system—paging should be logically transparent to the user. So this option is not the best choice.

The operating system could swap out a process, freeing all its frames, and reducing the level of multiprogramming. This option is a good one in certain circumstances; we consider it further in Section 10.6. Here, we discuss a more intriguing possibility: **page replacement**.

10.4.1 Basic Scheme

Page replacement takes the following approach. If no frame is free, we find one that is not currently being used and free it. We can free a frame by writing its contents to swap space, and changing the page table (and all other tables) to indicate that the page is no longer in memory (Figure 10.7). We can now use the freed frame to hold the page for which the process faulted. We modify the page-fault service routine to include page replacement:

1. Find the location of the desired page on the disk.
2. Find a free frame:
 - a. If there is a free frame, use it.

- b. If there is no free frame, use a page-replacement algorithm to select a **victim** frame.
- c. Write the victim page to the disk; change the page and frame tables accordingly.
3. Read the desired page into the (newly) free frame; change the page and frame tables.
4. Restart the user process.

Notice that, if no frames are free, *two* page transfers (one out and one in) are required. This situation effectively doubles the page-fault service time and increases the effective access time accordingly.

We can reduce this overhead by using a **modify bit** (or **dirty bit**). Each page or frame may have a modify bit associated with it in the hardware. The modify bit for a page is set by the hardware whenever any word or byte in the page is written into, indicating that the page has been modified. When we select a page for replacement, we examine its modify bit. If the bit is set, we know that the page has been modified since it was read in from the disk. In this case, we must write that page to the disk. If the modify bit is not set, however, the page has *not* been modified since it was read into memory. Therefore, if the copy of the

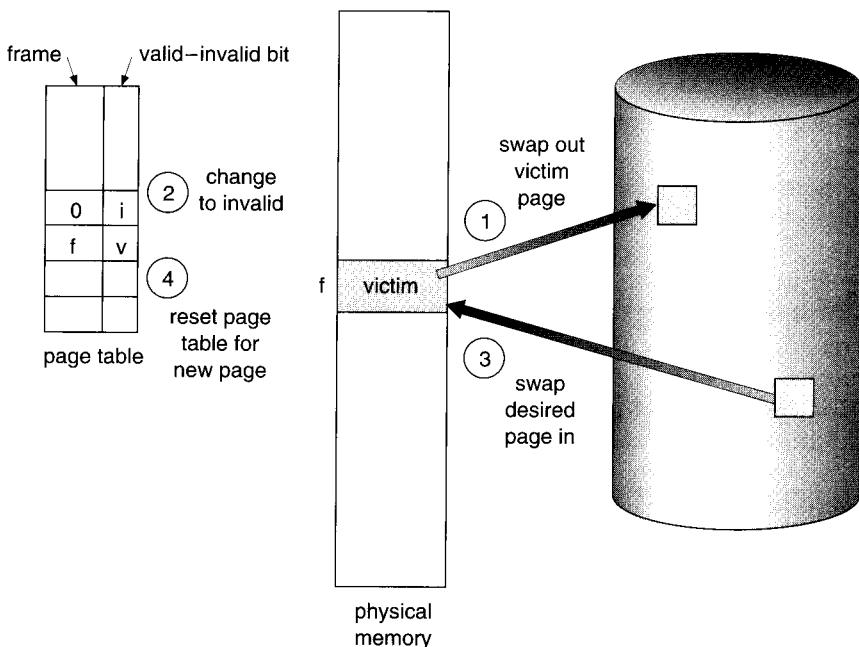


Figure 10.7 Page replacement.

page on the disk has not been overwritten (by some other page, for example), then we can avoid writing the memory page to the disk: it is already there. This technique also applies to read-only pages (for example, pages of binary code). Such pages cannot be modified; thus, they may be discarded when desired. This scheme can reduce significantly the time required to service a page fault, since it reduces I/O time by one-half if the page is not modified.

Page replacement is basic to demand paging. It completes the separation between logical memory and physical memory. With this mechanism, an enormous virtual memory can be provided for programmers on a smaller physical memory. With non-demand paging, user addresses are mapped into physical addresses, so the two sets of addresses can be different. All the pages of a process still must be in physical memory, however. With demand paging, the size of the logical address space is no longer constrained by physical memory. If we have a user process of 20 pages, we can execute it in ten frames simply by using demand paging, and using a replacement algorithm to find a free frame whenever necessary. If a page that has been modified is to be replaced, its contents are copied to the disk. A later reference to that page will cause a page fault. At that time, the page will be brought back into memory, perhaps replacing some other page in the process.

We must solve two major problems to implement demand paging: We must develop a **frame-allocation algorithm** and a **page-replacement algorithm**. If we have multiple processes in memory, we must decide how many frames to allocate to each process. Further, when page replacement is required, we must select the frames that are to be replaced. Designing appropriate algorithms to solve these problems is an important task, because disk I/O is so expensive. Even slight improvements in demand-paging methods yield large gains in system performance.

There are many different page-replacement algorithms. Every operating system probably has its own replacement scheme. How do we select a particular replacement algorithm? In general, we want the one with the lowest page-fault rate.

We evaluate an algorithm by running it on a particular string of memory references and computing the number of page faults. The string of memory references is called a **reference string**. We can generate reference strings artificially (by a random-number generator, for example) or we can trace a given system and record the address of each memory reference. The latter choice produces a large number of data (on the order of 1 million addresses per second). To reduce the number of data, we use two facts.

First, for a given page size (and the page size is generally fixed by the hardware or system), we need to consider only the page number, rather than the entire address. Second, if we have a reference to a page p , then any *immediately* following references to page p will never cause a page fault. Page p will be in memory after the first reference; the immediately following references will not fault.

For example, if we trace a particular process, we might record the following address sequence:

0100, 0432, 0101, 0612, 0102, 0103, 0104, 0101, 0611, 0102, 0103,
0104, 0101, 0610, 0102, 0103, 0104, 0101, 0609, 0102, 0105,

which, at 100 bytes per page, is reduced to the following reference string

1, 4, 1, 6, 1, 6, 1, 6, 1.

To determine the number of page faults for a particular reference string and page-replacement algorithm, we also need to know the number of page frames available. Obviously, as the number of frames available increases, the number of page faults decreases. For the reference string considered previously, for example, if we had three or more frames, we would have only three faults, one fault for the first reference to each page. On the other hand, with only one frame available, we would have a replacement with every reference, resulting in 11 faults. In general, we expect a curve such as that in Figure 10.8. As the number of frames increases, the number of page faults drops to some minimal level. Of course, adding physical memory increases the number of frames.

To illustrate the page-replacement algorithms, we shall use the reference string

7, 0, 1, 2, 0, 3, 0, 4, 2, 3, 0, 3, 2, 1, 2, 0, 1, 7, 0, 1

for a memory with three frames.

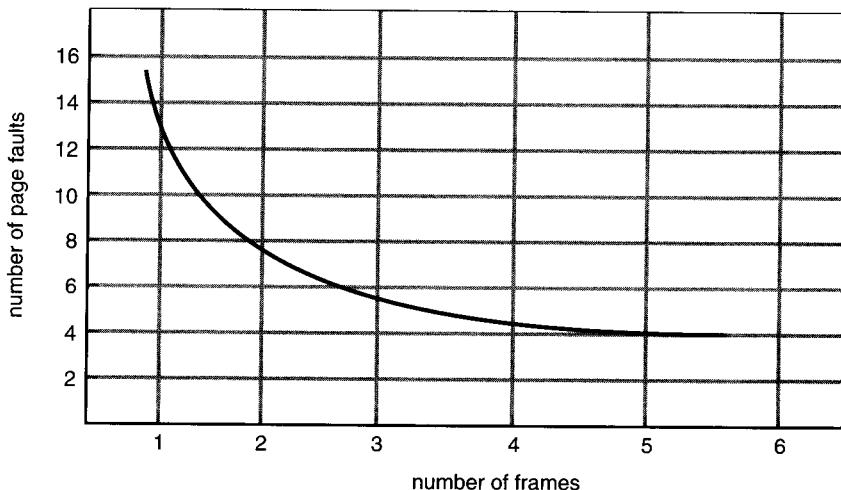


Figure 10.8 Graph of page faults versus the number of frames.

10.4.2 FIFO Page Replacement

The simplest page-replacement algorithm is a FIFO algorithm. A FIFO replacement algorithm associates with each page the time when that page was brought into memory. When a page must be replaced, the oldest page is chosen. Notice that it is not strictly necessary to record the time when a page is brought in. We can create a FIFO queue to hold all pages in memory. We replace the page at the head of the queue. When a page is brought into memory, we insert it at the tail of the queue.

For our example reference string, our three frames are initially empty. The first three references (7, 0, 1) cause page faults, and are brought into these empty frames. The next reference (2) replaces page 7, because page 7 was brought in first. Since 0 is the next reference and 0 is already in memory, we have no fault for this reference. The first reference to 3 results in page 0 being replaced, since it was the first of the three pages in memory (0, 1, and 2) to be brought in. Because of this replacement, the next reference, to 0, will fault. Page 1 is then replaced by page 0. This process continues as shown in Figure 10.9. Every time a fault occurs, we show which pages are in our three frames. There are 15 faults altogether.

The FIFO page-replacement algorithm is easy to understand and program. However, its performance is not always good. The page replaced may be an initialization module that was used a long time ago and is no longer needed. On the other hand, it could contain a heavily used variable that was initialized early and is in constant use.

Notice that, even if we select for replacement a page that is in active use, everything still works correctly. After we page out an active page to bring in a new one, a fault occurs almost immediately to retrieve the active page. Some other page will need to be replaced to bring the active page back into memory. Thus, a bad replacement choice increases the page-fault rate and slows process execution, but does not cause incorrect execution.

To illustrate the problems that are possible with a FIFO page-replacement algorithm, we consider the reference string

$$1, 2, 3, 4, 1, 2, 5, 1, 2, 3, 4, 5.$$

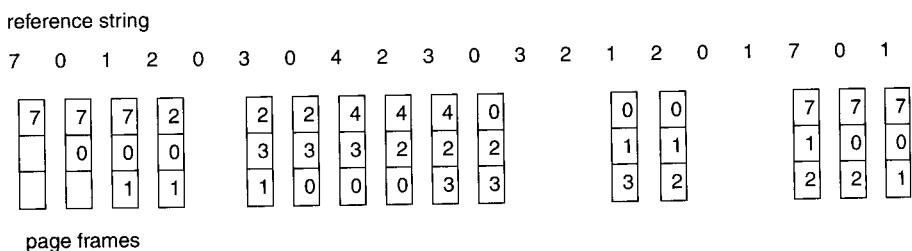


Figure 10.9 FIFO page-replacement algorithm.

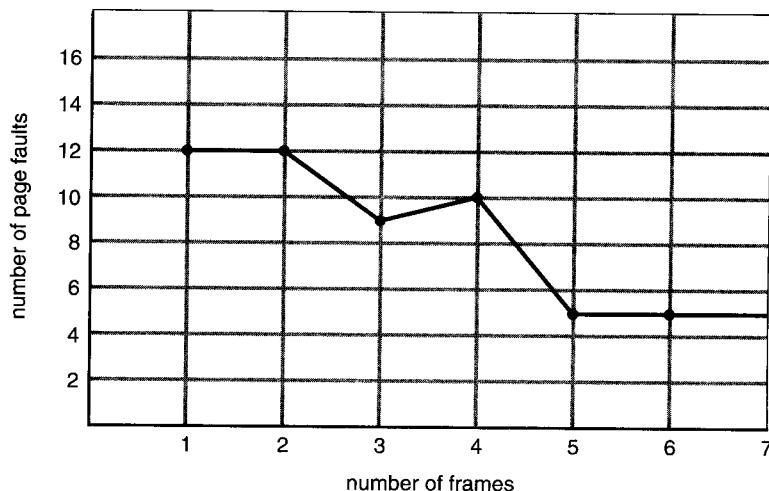


Figure 10.10 Page-fault curve for FIFO replacement on a reference string.

Figure 10.10 shows the curve of page faults versus the number of available frames. We notice that the number of faults for four frames (10) is *greater* than the number of faults for three frames (nine)! This most unexpected result is known as **Belady's anomaly**: For some page-replacement algorithms, the page-fault rate may *increase* as the number of allocated frames increases. We would expect that giving more memory to a process would improve its performance. In some early research, investigators noticed that this assumption was not always true. Belady's anomaly was discovered as a result.

10.4.3 Optimal Page Replacement

One result of the discovery of Belady's anomaly was the search for an **optimal page-replacement algorithm**. An optimal page-replacement algorithm has the lowest page-fault rate of all algorithms, and will never suffer from Belady's anomaly. Such an algorithm does exist, and has been called OPT or MIN. It is simply

Replace the page that will not be used
for the longest period of time.

Use of this page-replacement algorithm guarantees the lowest possible page-fault rate for a fixed number of frames.

For example, on our sample reference string, the optimal page-replacement algorithm would yield nine page faults, as shown in Figure 10.11. The first three references cause faults that fill the three empty frames. The reference to page

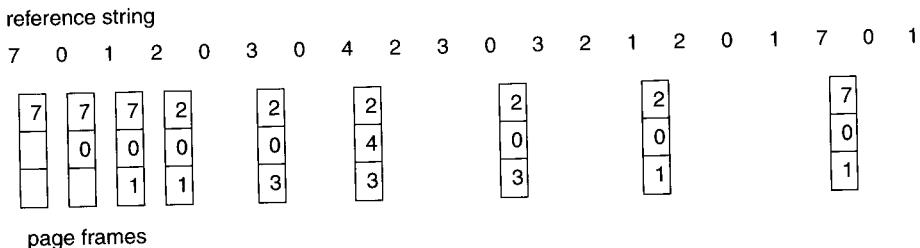


Figure 10.11 Optimal page-replacement algorithm.

2 replaces page 7, because 7 will not be used until reference 18, whereas page 0 will be used at 5, and page 1 at 14. The reference to page 3 replaces page 1, as page 1 will be the last of the three pages in memory to be referenced again. With only nine page faults, optimal replacement is much better than a FIFO algorithm, which had 15 faults. (If we ignore the first three, which all algorithms must suffer, then optimal replacement is twice as good as FIFO replacement.) In fact, no replacement algorithm can process this reference string in three frames with less than nine faults.

Unfortunately, the optimal page-replacement algorithm is difficult to implement, because it requires future knowledge of the reference string. (We encountered a similar situation with the SJF CPU-scheduling algorithm in Section 6.3.2.) As a result, the optimal algorithm is used mainly for comparison studies. For instance, it may be useful to know that, although a new algorithm is not optimal, it is within 12.3 percent of optimal at worst, and within 4.7 percent on average.

10.4.4 LRU Page Replacement

If the optimal algorithm is not feasible, perhaps an approximation to the optimal algorithm is possible. The key distinction between the FIFO and OPT algorithms (other than looking backward or forward in time) is that the FIFO algorithm uses the time when a page was brought into memory; the OPT algorithm uses the time when a page is to be *used*. If we use the recent past as an approximation of the near future, then we will replace the page that *has not been used* for the longest period of time (Figure 10.12). This approach is the **least-recently-used (LRU)** algorithm.

LRU replacement associates with each page the time of that page's last use. When a page must be replaced, LRU chooses that page that has not been used for the longest period of time. This strategy is the optimal page-replacement algorithm looking backward in time, rather than forward. (Strangely, if we let S^R be the reverse of a reference string S , then the page-fault rate for the OPT algorithm on S is the same as the page-fault rate for the OPT algorithm on

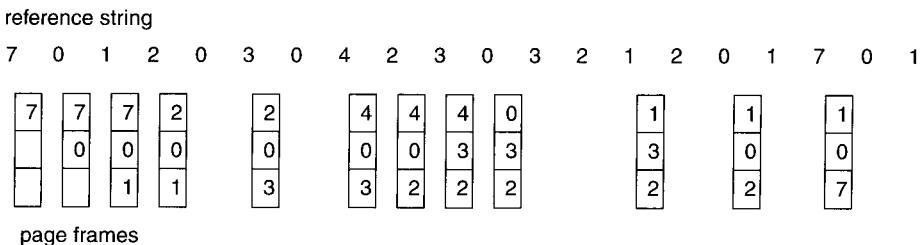


Figure 10.12 LRU page-replacement algorithm.

S^R . Similarly, the page-fault rate for the LRU algorithm on S is the same as the page-fault rate for the LRU algorithm on S^R .)

The result of applying LRU replacement to our example reference string is shown in Figure 10.12. The LRU algorithm produces 12 faults. Notice that the first five faults are the same as the optimal replacement. When the reference to page 4 occurs, however, LRU replacement sees that, of the three frames in memory, page 2 was used least recently. The most recently used page is page 0, and just before that page 3 was used. Thus, the LRU algorithm replaces page 2, not knowing that page 2 is about to be used. When it then faults for page 2, the LRU algorithm replaces page 3 since, of the three pages in memory $\{0, 3, 4\}$, page 3 is the least recently used. Despite these problems, LRU replacement with 12 faults is still much better than FIFO replacement with 15.

The LRU policy is often used as a page-replacement algorithm and is considered to be good. The major problem is *how* to implement LRU replacement. An LRU page-replacement algorithm may require substantial hardware assistance. The problem is to determine an order for the frames defined by the time of last use. Two implementations are feasible:

- **Counters:** In the simplest case, we associate with each page-table entry a time-of-use field, and add to the CPU a logical clock or counter. The clock is incremented for every memory reference. Whenever a reference to a page is made, the contents of the clock register are copied to the time-of-use field in the page-table entry for that page. In this way, we always have the “time” of the last reference to each page. We replace the page with the smallest time value. This scheme requires a search of the page table to find the LRU page, and a write to memory (to the time-of-use field in the page table) for each memory access. The times must also be maintained when page tables are changed (due to CPU scheduling). Overflow of the clock must be considered.
- **Stack:** Another approach to implementing LRU replacement is to keep a stack of page numbers. Whenever a page is referenced, it is removed from the stack and put on the top. In this way, the top of the stack is always the most recently used page and the bottom is the LRU page (Figure

10.13). Because entries must be removed from the middle of the stack, it is best implemented by a doubly linked list, with a head and tail pointer. Removing a page and putting it on the top of the stack then requires changing six pointers at worst. Each update is a little more expensive, but there is no search for a replacement; the tail pointer points to the bottom of the stack, which is the LRU page. This approach is particularly appropriate for software or microcode implementations of LRU replacement.

Neither optimal replacement nor LRU replacement suffers from Belady's anomaly. There is a class of page-replacement algorithms, called **stack algorithms**, that can never exhibit Belady's anomaly. A stack algorithm is an algorithm for which it can be shown that the set of pages in memory for n frames is always a *subset* of the set of pages that would be in memory with $n + 1$ frames. For LRU replacement, the set of pages in memory would be the n most recently referenced pages. If the number of frames is increased, these n pages will still be the most recently referenced and so will still be in memory.

Note that neither implementation of LRU would be conceivable without hardware assistance beyond the standard TLB registers. The updating of the clock fields or stack must be done for *every* memory reference. If we were to use an interrupt for every reference, to allow software to update such data structures, it would slow every memory reference by a factor of at least ten, hence slowing every user process by a factor of ten. Few systems could tolerate that level of overhead for memory management.

10.4.5 LRU Approximation Page Replacement

Few computer systems provide sufficient hardware support for true LRU page replacement. Some systems provide no hardware support, and other page-

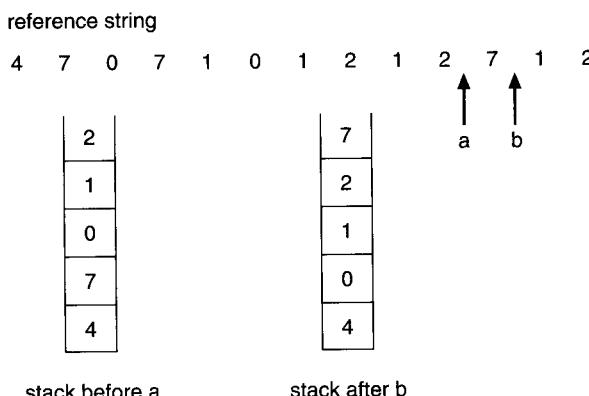


Figure 10.13 Use of a stack to record the most recent page references.

replacement algorithms (such as a FIFO algorithm) must be used. Many systems provide some help, however, in the form of a **reference bit**. The reference bit for a page is set, by the hardware, whenever that page is referenced (either a read or a write to any byte in the page). Reference bits are associated with each entry in the page table.

Initially, all bits are cleared (to 0) by the operating system. As a user process executes, the bit associated with each page referenced is set (to 1) by the hardware. After some time, we can determine which pages have been used and which have not been used by examining the reference bits. We do not know the *order* of use, but we know which pages were used and which were not used. This partial ordering information leads to many page-replacement algorithms that approximate LRU replacement.

10.4.5.1 Additional-Reference-Bits Algorithm

We can gain additional ordering information by recording the reference bits at regular intervals. We can keep an 8-bit byte for each page in a table in memory. At regular intervals (say, every 100 milliseconds), a timer interrupt transfers control to the operating system. The operating system shifts the reference bit for each page into the high-order bit of its 8-bit byte, shifting the other bits right 1 bit, discarding the low-order bit. These 8-bit shift registers contain the history of page use for the last eight time periods. If the shift register contains 00000000, then the page has not been used for eight time periods; a page that is used at least once each period would have a shift register value of 11111111.

A page with a history register value of 11000100 has been used more recently than has one with 01110111. If we interpret these 8-bit bytes as unsigned integers, the page with the lowest number is the LRU page, and it can be replaced. Notice that the numbers are not guaranteed to be unique, however. We can either replace (swap out) all pages with the smallest value, or use a FIFO selection among them.

The number of bits of history can be varied, of course, and would be selected (depending on the hardware available) to make the updating as fast as possible. In the extreme case, the number can be reduced to zero, leaving only the reference bit itself. This algorithm is called the **second-chance page-replacement algorithm**.

10.4.5.2 Second-Chance Algorithm

The basic algorithm of second-chance replacement is a FIFO replacement algorithm. When a page has been selected, however, we inspect its reference bit. If the value is 0, we proceed to replace this page. If the reference bit is set to 1, however, we give that page a second chance and move on to select the next FIFO page. When a page gets a second chance, its reference bit is cleared and its arrival time is reset to the current time. Thus, a page that is given a second

chance will not be replaced until all other pages are replaced (or given second chances). In addition, if a page is used often enough to keep its reference bit set, it will never be replaced.

One way to implement the second-chance (sometimes referred to as the *clock*) algorithm is as a circular queue. A pointer indicates which page is to be replaced next. When a frame is needed, the pointer advances until it finds a page with a 0 reference bit. As it advances, it clears the reference bits (Figure 10.14). Once a victim page is found, the page is replaced, and the new page is inserted in the circular queue in that position. Notice that, in the worst case, when all bits are set, the pointer cycles through the whole queue, giving each page a second chance. It clears all the reference bits before selecting the next page for replacement. Second-chance replacement degenerates to FIFO replacement if all bits are set.

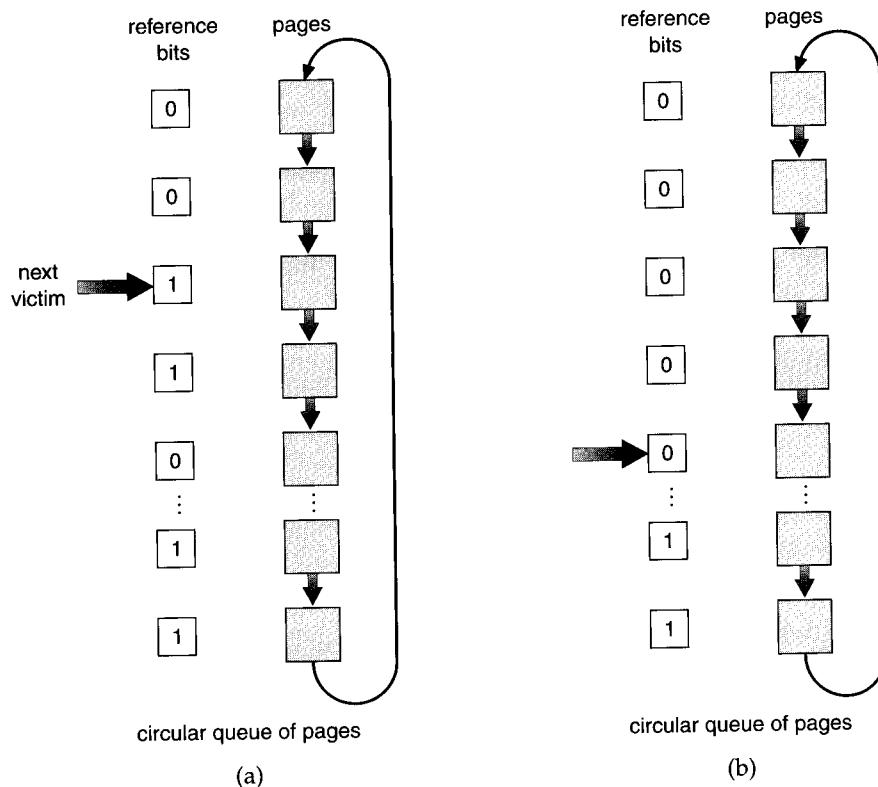


Figure 10.14 Second-chance (clock) page-replacement algorithm.

10.4.5.3 Enhanced Second-Chance Algorithm

We can enhance the second-chance algorithm by considering both the reference bit and the modify bit (Section 10.4) as an ordered pair. With these two bits, we have the following four possible classes:

1. (0, 0) neither recently used nor modified—best page to replace
2. (0, 1) not recently used but modified—not quite as good, because the page will need to be written out before replacement
3. (1, 0) recently used but clean—it probably will be used again soon
4. (1, 1) recently used and modified—it probably will be used again soon, and the page will be need to be written out to disk before it can be replaced

When page replacement is called for, each page is in one of these four classes. We use the same scheme as the clock algorithm, but instead of examining whether the page to which we are pointing has the reference bit set to 1, we examine the class to which that page belongs. We replace the first page encountered in the lowest nonempty class. Notice that we may have to scan the circular queue several times before we find a page to be replaced.

This algorithm is used in the Macintosh virtual-memory-management scheme. The major difference between this algorithm and the simpler clock algorithm is that here we give preference to those pages that have been modified to reduce the number of I/Os required.

10.4.6 Counting-Based Page Replacement

There are many other algorithms that can be used for page replacement. For example, we could keep a counter of the number of references that have been made to each page, and develop the following two schemes.

- The **least frequently used (LFU) page-replacement algorithm** requires that the page with the smallest count be replaced. The reason for this selection is that an actively used page should have a large reference count. This algorithm suffers from the situation in which a page is used heavily during the initial phase of a process, but then is never used again. Since it was used heavily, it has a large count and remains in memory even though it is no longer needed. One solution is to shift the counts right by 1 bit at regular intervals, forming an exponentially decaying average usage count.
- The **most frequently used (MFU) page-replacement algorithm** is based on the argument that the page with the smallest count was probably just brought in and has yet to be used.

As you might expect, neither MFU nor LFU replacement is common. The implementation of these algorithms is expensive, and they do not approximate OPT replacement well.

10.4.7 Page-Buffering Algorithm

Other procedures are often used in addition to a specific page-replacement algorithm. For example, systems commonly keep a pool of free frames. When a page fault occurs, a victim frame is chosen as before. However, the desired page is read into a free frame from the pool before the victim is written out. This procedure allows the process to restart as soon as possible, without waiting for the victim page to be written out. When the victim is later written out, its frame is added to the free-frame pool.

An expansion of this idea is to maintain a list of modified pages. Whenever the paging device is idle, a modified page is selected and is written to the disk. Its modify bit is then reset. This scheme increases the probability that a page will be clean when it is selected for replacement, and will not need to be written out.

Another modification is to keep a pool of free frames, but to remember which page was in each frame. Since the frame contents are not modified when a frame is written to the disk, the old page can be reused directly from the free-frame pool if it is needed before that frame is reused. No I/O is needed in this case. When a page fault occurs, we first check whether the desired page is in the free-frame pool. If it is not, we must select a free frame and read into it.

This technique is used in the VAX/VMS system, with a FIFO replacement algorithm. When the FIFO replacement algorithm mistakenly replaces a page that is still in active use, that page is quickly retrieved from the free-frame buffer, and no I/O is necessary. The free-frame buffer provides protection against the relatively poor, but simple, FIFO replacement algorithm. This method is necessary because the early versions of the VAX did not implement the reference bit correctly.

10.5 ■ Allocation of Frames

How do we allocate the fixed amount of free memory among the various processes? If we have 93 free frames and two processes, how many frames does each process get?

The simplest case of virtual memory is the single-user system. Consider a single-user system with 128 KB memory composed of pages of size 1 KB. Thus, there are 128 frames. The operating system may take 35 KB, leaving 93 frames for the user process. Under pure demand paging, all 93 frames would initially be put on the free-frame list. When a user process started execution, it would generate a sequence of page faults. The first 93 page faults would all get free

frames from the free-frame list. When the free-frame list was exhausted, a page-replacement algorithm would be used to select one of the 93 in-memory pages to be replaced with the ninety-fourth, and so on. When the process terminated, the 93 frames would once again be placed on the free-frame list.

There are many variations on this simple strategy. We can require that the operating system allocate all its buffer and table space from the free-frame list. When this space is not in use by the operating system, it can be used to support user paging. We can try to keep three free frames reserved on the free-frame list at all times. Thus, when a page fault occurs, there is a free frame available to page into. While the page swap is taking place, a replacement can be selected, which is then written to the disk as the user process continues to execute.

Other variants are also possible, but the basic strategy is clear: The user process is allocated any free frame.

A different problem arises when demand paging is combined with multiprogramming. Multiprogramming puts two (or more) processes in memory at the same time.

10.5.1 Minimum Number of Frames

Our strategies for the allocations of frames are constrained in various ways. We cannot allocate more than the total number of available frames (unless there is page sharing). We must also allocate at least a minimum number of frames. Obviously, as the number of frames allocated to each process decreases, the page-fault-rate increases, slowing process execution.

Besides the undesirable performance properties of allocating only a few frames, there is a minimum number of frames that must be allocated. This minimum number is defined by the instruction-set architecture. Remember that, when a page fault occurs before an executing instruction is complete, the instruction must be restarted. Consequently, we must have enough frames to hold all the different pages that any single instruction can reference.

For example, consider a machine in which all memory-reference instructions have only one memory address. Thus, we need at least one frame for the instruction and one frame for the memory reference. In addition, if one-level indirect addressing is allowed (for example, a load instruction on page 16 can refer to an address on page 0, which is an indirect reference to page 23), then paging requires at least three frames per process. Think about what might happen if a process had only two frames.

The minimum number of frames is defined by the given computer architecture. For example, the move instruction for the PDP-11 is more than one word for some addressing modes, and thus the instruction itself may straddle two pages. In addition, each of its two operands may be indirect references, for a total of six frames. The worst case for the IBM 370 is probably the MVC instruction. Since the instruction is storage to storage, it takes 6 bytes and can straddle two pages. The block of characters to move and the area to be moved

to can each also straddle two pages. This situation would require six frames. (Actually, the worst case occurs when the MVC instruction is the operand of an EXECUTE instruction that straddles a page boundary; in this case, we need eight frames.)

The worst-case scenario occurs in computer architectures that allow multiple levels of indirection (for example, each 16-bit word could contain a 15-bit address plus a 1-bit indirect indicator). Theoretically, a simple load instruction could reference an indirect address that could reference an indirect address (on another page) that could also reference an indirect address (on yet another page), and so on, until every page in virtual memory had been touched. Thus, in the worst case, the entire virtual memory must be in physical memory. To overcome this difficulty, we must place a limit on the levels of indirection (for example, limit an instruction to at most 16 levels of indirection). When the first indirection occurs, a counter is set to 16; the counter is then decremented for each successive indirection for this instruction. If the counter is decremented to 0, a trap occurs (excessive indirection). This limitation reduces the maximum number of memory references per instruction to 17, requiring the same number of frames.

The minimum number of frames per process is defined by the architecture, whereas the maximum number is defined by the amount of available physical memory. In between, we are still left with significant choice in frame allocation.

10.5.2 Allocation Algorithms

The easiest way to split m frames among n processes is to give everyone an equal share, m/n frames. For instance, if there are 93 frames and five processes, each process will get 18 frames. The leftover three frames could be used as a free-frame buffer pool. This scheme is called **equal allocation**.

An alternative is to recognize that various processes will need differing amounts of memory. Consider a system with a 1 KB frame size. If a small student process of 10 KB and an interactive database of 127 KB are the only two processes running in a system with 62 free frames, it does not make much sense to give each process 31 frames. The student process does not need more than ten frames, so the other 21 are strictly wasted.

To solve this problem, we can use **proportional allocation**. We allocate available memory to each process according to its size. Let the size of the virtual memory for process p_i be s_i , and define

$$S = \sum s_i.$$

Then, if the total number of available frames is m , we allocate a_i frames to process p_i , where a_i is approximately

$$a_i = s_i / S \times m.$$

Of course, we must adjust each a_i to be an integer that is greater than the minimum number of frames required by the instruction set, with a sum not exceeding m .

For proportional allocation, we would split 62 frames between two processes, one of 10 pages and one of 127 pages, by allocating 4 frames and 57 frames, respectively, since

$$\begin{aligned} 10/137 \times 62 &\approx 4, \\ 127/137 \times 62 &\approx 57. \end{aligned}$$

In this way, both processes share the available frames according to their “needs,” rather than equally.

In both equal and proportional allocation, of course, the allocation to each process may vary according to the multiprogramming level. If the multiprogramming level is increased, each process will lose some frames to provide the memory needed for the new process. On the other hand, if the multiprogramming level decreases, the frames that had been allocated to the departed process can now be spread over the remaining processes.

Notice that, with either equal or proportional allocation, a high-priority process is treated the same as a low-priority process. By its definition, however, we may want to give the high-priority process more memory to speed its execution, to the detriment of low-priority processes.

One approach is to use a proportional allocation scheme where the ratio of frames depends not on the relative sizes of processes, but rather on the priorities of processes, or on a combination of size and priority.

10.5.3 Global Versus Local Allocation

Another important factor in the way frames are allocated to the various processes is page replacement. With multiple processes competing for frames, we can classify page-replacement algorithms into two broad categories: **global replacement** and **local replacement**. Global replacement allows a process to select a replacement frame from the set of all frames, even if that frame is currently allocated to some other process; one process can take a frame from another. Local replacement requires that each process select from only its own set of allocated frames.

For example, consider an allocation scheme where we allow high-priority processes to select frames from low-priority processes for replacement. A process can select a replacement from among its own frames or the frames of any lower-priority process. This approach allows a high-priority process to increase its frame allocation at the expense of the low-priority process.

With a local replacement strategy, the number of frames allocated to a process does not change. With global replacement, a process may happen to select only frames allocated to other processes, thus increasing the number of

frames allocated to it (assuming that other processes do not choose *its* frames for replacement).

One problem with a global replacement algorithm is that a process cannot control its own page-fault rate. The set of pages in memory for a process depends not only on the paging behavior of that process, but also on the paging behavior of other processes. Therefore, the same process may perform quite differently (taking 0.5 seconds for one execution and 10.3 seconds for the next execution) due to totally external circumstances. Such is not the case with a local replacement algorithm. Under local replacement, the set of pages in memory for a process is affected by the paging behavior of only that process. For its part, local replacement might hinder a process by not making available to it other, less used pages of memory. Thus, global replacement generally results in greater system throughput, and is therefore the more common method.

10.6 ■ Thrashing

If the number of frames allocated to a low-priority process falls below the minimum number required by the computer architecture, we must suspend that process' execution. We should then page out its remaining pages, freeing all its allocated frames. This provision introduces a swap-in, swap-out level of intermediate CPU scheduling.

In fact, look at any process that does not have “enough” frames. Although it is technically possible to reduce the number of allocated frames to the minimum, there is some (larger) number of pages in active use. If the process does not have this number of frames, it will quickly page fault. At this point, it must replace some page. However, since all its pages are in active use, it must replace a page that will be needed again right away. Consequently, it quickly faults again, and again, and again. The process continues to fault, replacing pages for which it then faults and brings back in right away.

This high paging activity is called **thrashing**. A process is thrashing if it is spending more time paging than executing.

10.6.1 Cause of Thrashing

Thrashing results in severe performance problems. Consider the following scenario, which is based on the actual behavior of early paging systems.

The operating system monitors CPU utilization. If CPU utilization is too low, we increase the degree of multiprogramming by introducing a new process to the system. A global page-replacement algorithm is used; it replaces pages with no regard to the process to which they belong. Now suppose that a process enters a new phase in its execution and needs more frames. It starts faulting and taking frames away from other processes. These processes need those pages, however, and so they also fault, taking frames from other processes. These

faulting processes must use the paging device to swap pages in and out. As they queue up for the paging device, the ready queue empties. As processes wait for the paging device, CPU utilization decreases.

The CPU scheduler sees the decreasing CPU utilization, and *increases* the degree of multiprogramming as a result. The new process tries to get started by taking frames from running processes, causing more page faults, and a longer queue for the paging device. As a result, CPU utilization drops even further, and the CPU scheduler tries to increase the degree of multiprogramming even more. Thrashing has occurred and system throughput plunges. The page-fault rate increases tremendously. As a result, the effective memory access time increases. No work is getting done, because the processes are spending all their time paging.

This phenomenon is illustrated in Figure 10.15, in which CPU utilization is plotted against the degree of multiprogramming. As the degree of multiprogramming increases, CPU utilization also increases, although more slowly, until a maximum is reached. If the degree of multiprogramming is increased even further, thrashing sets in and CPU utilization drops sharply. At this point, to increase CPU utilization and stop thrashing, we must *decrease* the degree of multiprogramming.

We can limit the effects of thrashing by using a **local replacement algorithm** (or **priority replacement algoritm**). With local replacement, if one process starts thrashing, it cannot steal frames from another process and cause the latter to thrash also. Pages are replaced with regard to the process of which they are a part. However, if processes are thrashing, they will be in the queue for the paging device most of the time. The average service time for a page fault will increase, due to the longer average queue for the paging device. Thus, the effective access time will increase even for a process that is not thrashing.

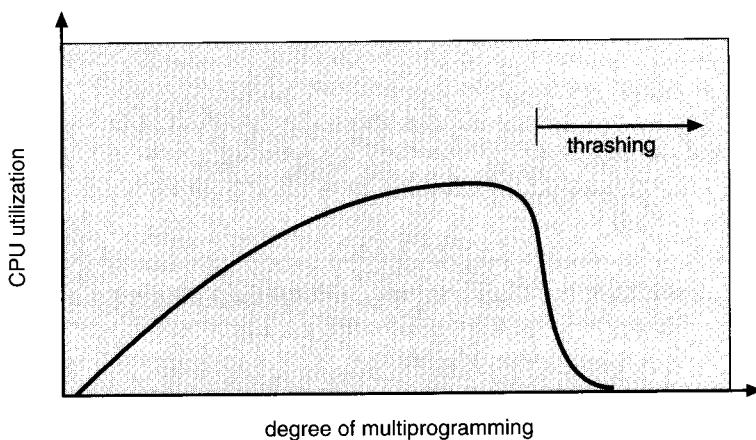


Figure 10.15 Thrashing.

To prevent thrashing, we must provide a process as many frames as it needs. But how do we know how many frames it “needs”? There are several techniques. The working-set strategy (Section 10.6.2) starts by looking at how many frames a process is actually using. This approach defines the **locality model** of process execution.

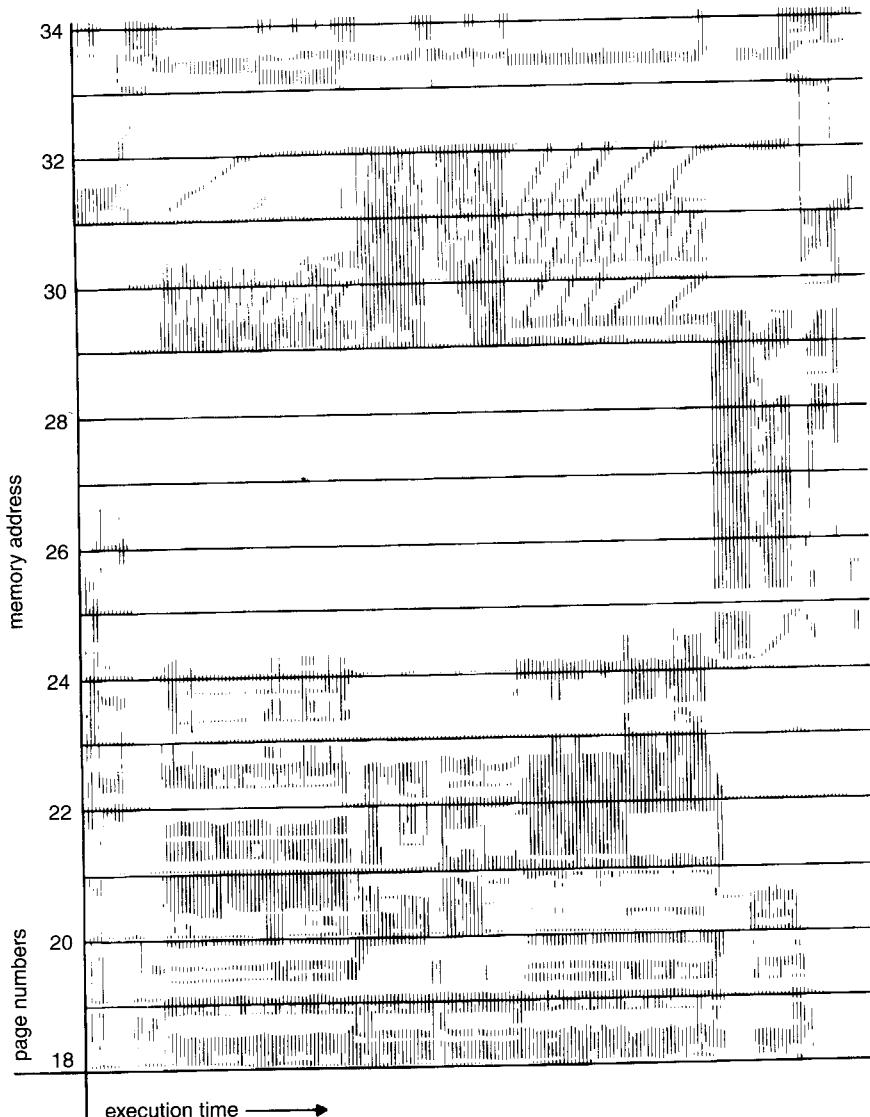


Figure 10.16 Locality in a memory-reference pattern.

The locality model states that, as a process executes, it moves from locality to locality. A locality is a set of pages that are actively used together (Figure 10.16). A program is generally composed of several different localities, which may overlap.

For example, when a subroutine is called, it defines a new locality. In this locality, memory references are made to the instructions of the subroutine, its local variables, and a subset of the global variables. When the subroutine is exited, the process leaves this locality, since the local variables and instructions of the subroutine are no longer in active use. We may return to this locality later. Thus, we see that localities are defined by the program structure and its data structures. The locality model states that all programs will exhibit this basic memory reference structure. Note that the locality model is the unstated principle behind the caching discussions so far in this book. If accesses to any types of data were random rather than patterned, caching would be useless.

Suppose that we allocate enough frames to a process to accommodate its current locality. It will fault for the pages in its locality until all these pages are in memory; then, it will not fault again until it changes localities. If we allocate fewer frames than the size of the current locality, the process will thrash, since it cannot keep in memory all the pages that it is actively using.

10.6.2 Working-Set Model

The **working-set model** is based on the assumption of locality. This model uses a parameter, Δ , to define the **working-set window**. The idea is to examine the most recent Δ page references. The set of pages in the most recent Δ page references is the **working set** (Figure 10.17). If a page is in active use, it will be in the working set. If it is no longer being used, it will drop from the working set Δ time units after its last reference. Thus, the working set is an approximation of the program's locality.

For example, given the sequence of memory references shown in Figure 10.17, if $\Delta = 10$ memory references, then the working set at time t_1 is $\{1, 2, 5, 6, 7\}$. By time t_2 , the working set has changed to $\{3, 4\}$.

The accuracy of the working set depends on the selection of Δ . If Δ is too small, it will not encompass the entire locality; if Δ is too large, it may overlap

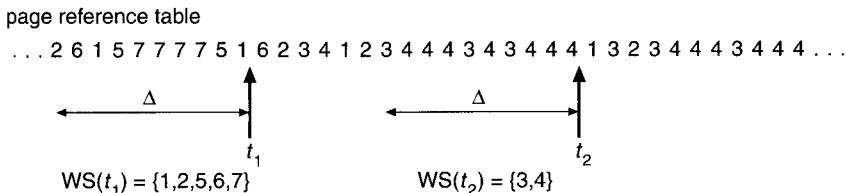


Figure 10.17 Working-set model.

several localities. In the extreme, if Δ is infinite, the working set is the set of pages touched during the process execution.

The most important property of the working set is its size. If we compute the working-set size, WSS_i , for each process in the system, we can then consider

$$D = \sum WSS_i,$$

where D is the total demand for frames. Each process is actively using the pages in its working set. Thus, process i needs WSS_i frames. If the total demand is greater than the total number of available frames ($D > m$), thrashing will occur, because some processes will not have enough frames.

Use of the working-set model is then simple. The operating system monitors the working set of each process and allocates to that working set enough frames to provide it with its working-set size. If there are enough extra frames, another process can be initiated. If the sum of the working-set sizes increases, exceeding the total number of available frames, the operating system selects a process to suspend. The process' pages are written out and its frames are reallocated to other processes. The suspended process can be restarted later.

This working-set strategy prevents thrashing while keeping the degree of multiprogramming as high as possible. Thus, it optimizes CPU utilization.

The difficulty with the working-set model is keeping track of the working set. The working-set window is a moving window. At each memory reference, a new reference appears at one end and the oldest reference drops off the other end. A page is in the working set if it is referenced anywhere in the working-set window. We can approximate the working-set model with a fixed interval timer interrupt and a reference bit.

For example, assume Δ is 10,000 references and we can cause a timer interrupt every 5,000 references. When we get a timer interrupt, we copy and clear the reference-bit values for each page. Thus, if a page fault occurs, we can examine the current reference bit and 2 in-memory bits to determine whether a page was used within the last 10,000 to 15,000 references. If it was used, at least 1 of these bits will be on. If it has not been used, these bits will be off. Those pages with at least 1 bit on will be considered to be in the working set. Note that this arrangement is not entirely accurate, because we cannot tell where, within an interval of 5,000, a reference occurred. We can reduce the uncertainty by increasing the number of our history bits and the frequency of interrupts (for example, 10 bits and interrupts every 1,000 references). However, the cost to service these more frequent interrupts will be correspondingly higher.

10.6.3 Page-Fault Frequency

The working-set model is successful, and knowledge of the working set can be useful for prepaging (Section 10.8.1), but it seems a clumsy way to control thrashing. A strategy that uses the **page-fault frequency (PFF)** takes a more direct approach.

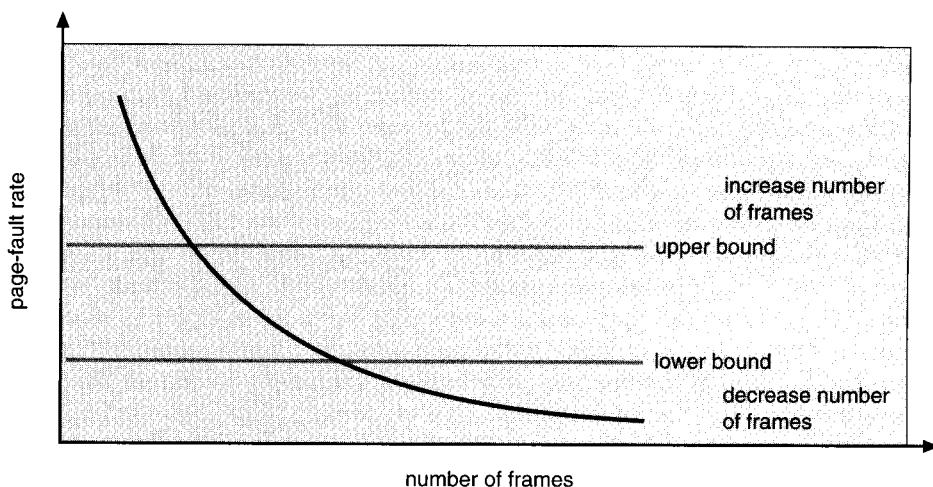


Figure 10.18 Page-fault frequency.

The specific problem is how to prevent thrashing. Thrashing has a high page-fault rate. Thus, we want to control the page-fault rate. When it is too high, we know that the process needs more frames. Similarly, if the page-fault rate is too low, then the process may have too many frames. We can establish upper and lower bounds on the desired page-fault rate (Figure 10.18). If the actual page-fault rate exceeds the upper limit, we allocate that process another frame; if the page-fault rate falls below the lower limit, we remove a frame from that process. Thus, we can directly measure and control the page-fault rate to prevent thrashing.

As with the working-set strategy, we may have to suspend a process. If the page-fault rate increases and no free frames are available, we must select some process and suspend it. The freed frames are then distributed to processes with high page-fault rates.

10.7 ■ Operating-System Examples

In this section we describe how Windows NT and Solaris 2 implement virtual memory.

10.7.1 Window NT

Windows NT implements virtual memory using demand paging with **clustering**. Clustering handles page faults by bringing in not only the faulting page, but also multiple pages surrounding the faulting page. When a process is first created, it is assigned a working-set minimum and maximum. The **working-set**

minimum is the minimum number of pages the process is guaranteed to have in memory. If sufficient memory is available, a process may be assigned as many pages as its **working-set maximum**. (In some circumstances, a process may be allowed to exceed its working-set maximum.) The virtual-memory manager maintains a list of free page frames. Associated with this list is a threshold value that is used to indicate whether there is sufficient free memory available or not. If a page fault occurs for a process that is below its working-set maximum, the virtual-memory manager allocates a page from this list of free pages. If a process is at its working-set maximum and it incurs a page fault, it must select a page for replacement using a local page-replacement policy. When the amount of free memory falls below the threshold, the virtual-memory manager uses a tactic known as **automatic working-set trimming** to restore the value above the threshold. Automatic working-set trimming works by evaluating the number of pages allocated to processes. If a process has been allocated more pages than its working-set minimum, the virtual-memory manager removes pages until the process reaches its working-set minimum. A process that is at its working-set minimum may be allocated pages from the free page frame list once sufficient free memory is available.

The algorithm used to determine which page to remove from a working set depends upon the type of processor the operating system is running on. On single-processor x86 systems, Windows NT uses a variation of the *clock* algorithm discussed in Section 10.4.5.2. On Alpha and multiprocessor x86 systems, clearing the reference bit may require invalidating the entry in the translation look-aside buffer on other processors. Rather than involving this overhead, Windows NT uses a variation on the FIFO algorithm discussed in Section 10.4.2.

10.7.2 Solaris 2

When a thread incurs a page fault, the kernel assigns a page to the faulting thread from the list of free pages it maintains. Therefore, it is imperative that the kernel maintain a sufficient amount of free memory available. Associated with this list of free pages is a parameter—*lotsfree*—which represents a threshold to begin paging. *lotsfree* is typically set to 1/64 of the size of the physical memory. Four times per second, the kernel checks if the amount of free memory is less than *lotsfree*.

If the number of free pages falls below *lotsfree*, a process known as the **pageout** starts up. The **pageout** process is similar to the second-chance algorithm described in Section 10.4.5.2 (also known as the **two-handed-clock algorithm**), which works as follows. The first hand of the clock scans all pages in memory, setting the reference bit to 0. At a later point in time, the second hand of the clock examines the reference bit for the pages in memory, returning those pages whose bit is still set to 0 to the free list.

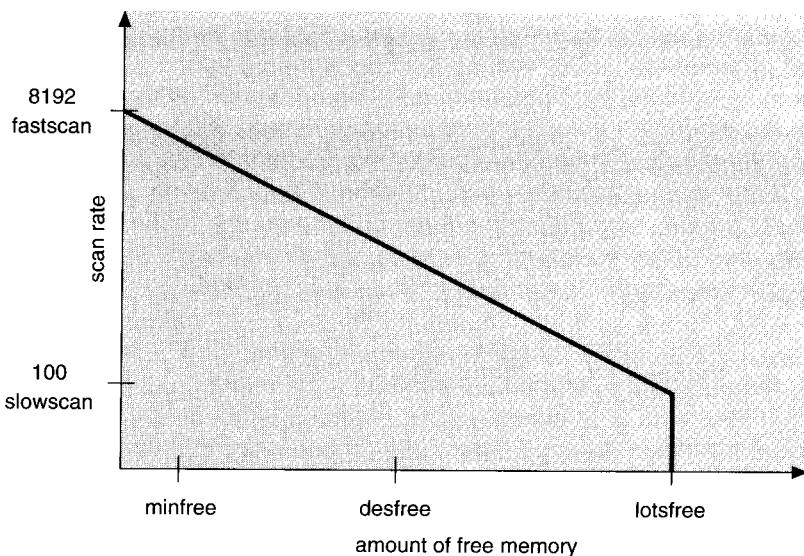


Figure 10.19 Solaris 2 page scanner.

The **pageout** algorithm uses several parameters to control the rate at which pages are scanned (known as the *scanrate*). The scanrate is expressed in pages per second and ranges from *slowscan* to *fastscan*. When free memory falls below *lotsfree*, scanning occurs at *slowscan* pages per second and progresses to *fastscan* depending upon the amount of free memory available. The default value of *slowscan* is 100 pages per second. *fastscan* is typically set to $(\text{TotalPhysicalPages})/2$ pages per second with a maximum of 8,192 pages per second. This is shown in Figure 10.19 (with *fastscan* set to the maximum).

The distance (in pages) between the hands of the clock is determined by a system parameter *handspread*. The amount of time between the front hand clearing a bit and the back hand investigating its value depends upon the *scanrate* and the *handspread*. If *scanrate* is 100 pages per second and *handspread* is 1,024 pages, 10 seconds can pass between the time a bit is set by the front hand and checked by the back hand. However, because of the demands placed on the memory system, a *scanrate* of several thousand is not uncommon. This means that the amount of time between checking and investigating a bit is often a few seconds.

As described above, the **pageout** process checks memory four times per second. However, if free memory falls below *desfree* (Figure 10.19) **pageout** will run 100 times per second with the intention of keeping at least *desfree* free memory available. If the **pageout** process is unable to keep the amount of free memory for a 30-second average at *desfree*, the kernel begins swapping processes, thereby freeing all pages allocated to the process. In general, the kernel looks for processes that have been idle for long periods of time. Last,

if the system is unable to maintain the amount of free memory at *minfree*, the **pageout** process is called for every request for a new page.

Recent releases of the Solaris 2 kernel have provided enhancements to the paging algorithm. One such enhancement is recognizing pages from shared libraries. Pages belonging to libraries that are being shared by several processes—even if they are eligible to be claimed by the scanner—are skipped during the page-scanning process. Another enhancement concerns distinguishing pages that have been allocated to processes from pages allocated to regular files. This is known as **priority paging** and is covered in Section 12.6.2.

10.8 ■ Other Considerations

The selections of a replacement algorithm and allocation policy are the major decisions that we make for a paging system. There are many other considerations as well.

10.8.1 Prepaging

An obvious property of a pure demand-paging system is the large number of page faults that occur when a process is started. This situation is a result of trying to get the initial locality into memory. The same situation may arise at other times. For instance, when a swapped-out process is restarted, all its pages are on the disk and each must be brought in by its own page fault. **Prepaging** is an attempt to prevent this high level of initial paging. The strategy is to bring into memory at one time all the pages that will be needed.

In a system using the working-set model, for example, we keep with each process a list of the pages in its working set. If we must suspend a process (due to an I/O wait or a lack of free frames), we remember the working set for that process. When the process is to be resumed (I/O completion or enough free frames), we automatically bring back into memory its entire working set before restarting the process.

Prepaging may be an advantage in some cases. The question is simply whether the cost of using prepaging is less than the cost of servicing the corresponding page faults. It may well be the case that many of the pages brought back into memory by prepaging are not used.

Assume that s pages are prepaged and a fraction α of these s pages is actually used ($0 \leq \alpha \leq 1$). The question is whether the cost of the $s * \alpha$ saved page faults is greater or less than the cost of prepaging $s * (1 - \alpha)$ unnecessary pages. If α is close to zero, prepaging loses; if α is close to one, prepaging wins.

10.8.2 Page Size

The designers of an operating system for an existing machine seldom have a choice concerning the page size. However, when new machines are being

designed, a decision regarding the best page size must be made. As you might expect, there is no single best page size. Rather, there is a set of factors that support various sizes. Page sizes are invariably powers of 2, generally ranging from 4,096 (2^{12}) to 4,194,304 (2^{22}) bytes.

How do we select a page size? One concern is the size of the page table. For a given virtual-memory space, decreasing the page size increases the number of pages, and hence the size of the page table. For a virtual memory of 4 MB (2^{22}), there would be 4,096 pages of 1,024 bytes, but only 512 pages of 8,192 bytes. Because each active process must have its own copy of the page table, a large page size is desirable.

On the other hand, memory is better utilized with smaller pages. If a process is allocated memory starting at location 00000, and continuing until it has as much as it needs, it probably will not end exactly on a page boundary. Thus, a part of the final page must be allocated (because pages are the units of allocation) but is unused (internal fragmentation). Assuming independence of process size and page size, we would expect that, on the average, one-half of the final page of each process will be wasted. This loss would be only 256 bytes for a page of 512 bytes, but would be 4,096 bytes for a page of 8,192 bytes. To minimize internal fragmentation, we need a small page size.

Another problem is the time required to read or write a page. I/O time is composed of seek, latency, and transfer times. Transfer time is proportional to the amount transferred (that is, the page size)—a fact that would seem to argue for a small page size. Remember from Chapter 2, however, that latency and seek time normally dwarf transfer time. At a transfer rate of 2 MB per second, it takes only 0.2 milliseconds to transfer 512 bytes. Latency, on the other hand, is perhaps 8 milliseconds and seek time 20 milliseconds. Of the total I/O time (28.2 milliseconds), therefore, 1 percent is attributable to the actual transfer. Doubling the page size increases I/O time to only 28.4 milliseconds. It takes 28.4 milliseconds to read a single page of 1,024 bytes, but 56.4 milliseconds to read the same amount as two pages of 512 bytes each. Thus, a desire to minimize I/O time argues for a larger page size.

With a smaller page size, however, total I/O should be reduced, since locality will be improved. A smaller page size allows each page to match program locality more accurately. For example, consider a process of size 200 KB, of which only one-half (100 KB) are actually used in an execution. If we have only one large page, we must bring in the entire page, a total of 200 KB transferred and allocated. If we had pages of only 1 byte, then we could bring in only the 100 KB that are actually used, resulting in only 100 KB being transferred and allocated. With a smaller page size, we have better **resolution**, allowing us to isolate only the memory that is actually needed. With a larger page size, we must allocate and transfer not only what is needed, but also anything else that happens to be in the page, whether it is needed or not. Thus, a smaller page size should result in less I/O and less total allocated memory.

On the other hand, did you notice that with a page size of 1 byte, we would have a page fault for *each* byte? A process of 200 KB, using only one-half of that memory, would generate only one page fault with a page size of 200 KB, but 102,400 page faults for a page size of 1 byte. Each page fault generates the large amount of overhead needed for processing the interrupt, saving registers, replacing a page, queueing for the paging device, and updating tables. To minimize the number of page faults, we need to have a large page size.

There are other factors to consider (such as the relationship between page size and sector size on the paging device). The problem has no best answer. Some factors (internal fragmentation, locality) argue for a small page size, whereas others (table size, I/O time) argue for a large page size. However, the historical trend is toward larger page sizes. Indeed, the first edition of **Operating Systems Concepts** (1983) used 4,096 bytes as the upper bound on page sizes, and this value was the most common page size in 1990. However, modern systems may now use page sizes that are much larger than this. We explore this in the following section.

10.8.3 TLB Reach

In Chapter 9, we introduced the **hit ratio** of the TLB. Recall the hit ratio for the TLB refers to the percentage of virtual address translations that are resolved in the TLB rather than the page table. Clearly, the hit ratio is related to the number of entries in the TLB and the way to increase the hit ratio is by increasing the number of entries in the TLB. This, however, does not come cheaply, as the associative memory used to construct the TLB is both expensive and power-hungry.

Related to the hit ratio is a similar metric: the **TLB reach**. The TLB reach refers to the amount of memory accessible from the TLB and is simply the number of entries multiplied by the page size. Ideally, the working set for a process is stored in the TLB. If not, the process will spend a considerable amount of time resolving memory references in the page table rather than TLB. If we double the number of entries in the TLB, we double the TLB reach. However, for some memory-intensive applications this may still prove insufficient for storing the working set.

Another approach for increasing the TLB reach is by either increasing the size of the page or providing multiple page sizes. If we increase the page size —say from 8 KB to 32 KB—we quadruple the TLB reach. However, this may lead to an increase in fragmentation for some applications that do not require such a large page size as 32 KB. Alternatively, an operating system may provide several different page sizes. For example, the UltraSparc II supports 8 KB, 64 KB, 512 KB, and 4 MB page sizes. Of these available pages sizes, Solaris 2 uses both 8 KB and 4 MB page sizes. And with a 64-entry TLB, the TLB reach for Solaris 2 ranges from 512 KB with 8 KB pages to 256 MB with 4 MB pages. For

the majority of applications, the 8 KB page size is sufficient, although Solaris 2 maps the first 4 MB of kernel code and data with two 4 MB pages. Solaris 2 also allows applications—such as databases—to take advantage of the large 4 MB page size as well.

However, providing support for multiple page sizes requires the operating system—not hardware—to manage the TLB. For example, one of the fields in a TLB entry must indicate the size of the page frame corresponding to the TLB entry. Managing the TLB in software and not hardware comes at a cost in performance. However, the increased hit ratio and TLB reach offsets the performance costs. Indeed, recent trends indicate a move towards software-managed TLBs and operating-system support for multiple page sizes. The UltraSparc, MIPS, and Alpha architectures employ software-managed TLBs. The PowerPC and Pentium manage the TLB in hardware.

10.8.4 Inverted Page Table

In Section 9.4.4.3, the concept of an inverted page table was introduced. The purpose of this form of page management was to reduce the amount of physical memory that is needed to track virtual-to-physical address translations. We accomplish this savings by creating a table that has one entry per physical memory page, indexed by the pair \langle process-id, page-number \rangle .

Because they keep information about which virtual-memory page is stored in each physical frame, inverted page tables reduce the amount of physical memory needed to store this information. However, the inverted page table no longer contains complete information about the logical address space of a process, and that information is required if a referenced page is not currently in memory. Demand paging requires this information to process page faults. For this information to be available, an external page table (one per process) must be kept. Each such table looks like the traditional per-process page table, containing information on where each virtual page is located.

But do external page tables negate the utility of inverted page tables? Since these tables are referenced only when a page fault occurs, they do not need to be available quickly. Instead, they are themselves paged in and out of memory as necessary. Unfortunately, a page fault may now result in the virtual-memory manager causing another page fault as it pages in the external page table it needs to locate the virtual page on the backing store. This special case requires careful handling in the kernel and a delay in the page-lookup processing.

10.8.5 Program Structure

Demand paging is designed to be transparent to the user program. In many cases, the user is completely unaware of the paged nature of memory. In other cases, however, system performance can be improved if the user (or compiler) has an awareness of the underlying demand paging.

Let's look at a contrived but informative example. Assume that pages are 128 words in size. Consider a Java program whose function is to initialize to 0 each element of a 128 by 128 array. The following code is typical:

```
int A[][] = new int[128][128];

for (int j = 0; j < A.length; j++)
    for (int i = 0; i < A.length; i++)
        A[i][j] = 0;
```

Notice that the array is stored row major. That is, the array is stored $A[0][0]$, $A[0][1], \dots, A[0][127]$, $A[1][0], A[1][1], \dots, A[127][127]$. For pages of 128 words, each row takes one page. Thus, the preceding code zeros one word in each page, then another word in each page, and so on. If the operating system allocates less than 128 frames to the entire program, then its execution will result in $128 \times 128 = 16,384$ page faults. Changing the code to

```
int A[][] = new int[128][128];

for (int i = 0; i < A.length; i++)
    for (int j = 0; j < A.length; j++)
        A[i][j] = 0;
```

on the other hand, zeros all the words on one page before starting the next page, reducing the number of page faults to 128.

Careful selection of data structures and programming structures can increase locality and hence lower the page-fault rate and the number of pages in the working set. A stack has good locality, since access is always made to the top. A hash table, on the other hand, is designed to scatter references, producing bad locality. Of course, locality of reference is just one measure of the efficiency of the use of a data structure. Other heavily weighted factors include search speed, total number of memory references, and total number of pages touched.

At a later stage, the compiler and loader can have a significant effect on paging. Separating code and data and generating reentrant code means that code pages can be read only and hence will never be modified. Clean pages do not have to be paged out to be replaced. The loader can avoid placing routines across page boundaries, keeping each routine completely in one page. Routines that call each other many times can be packed into the same page. This packaging is a variant of the bin-packing problem of operations research: Try to pack the variable-sized load segments into the fixed-sized pages so that interpage references are minimized. Such an approach is particularly useful for large page sizes.

The choice of programming language can affect paging as well. For example, C and C++ use pointers frequently, and pointers tend to randomize access to memory, thereby potentially diminishing a process' locality. Some studies

have shown that object-oriented programs also tend to have a poor locality of reference. Contrast these languages with Java, which does not provide pointers. Java programs will have better locality of reference than C or C++ programs on a virtual-memory system.

10.8.6 I/O Interlock

When demand paging is used, we sometimes need to allow some of the pages to be **locked** in memory. One such situation occurs when I/O is done to or from user (virtual) memory. I/O is often implemented by a separate I/O processor. For example, a magnetic-tape controller is generally given the number of bytes to transfer and a memory address for the buffer (Figure 10.20). When the transfer is complete, the CPU is interrupted.

We must be sure the following sequence of events does not occur: A process issues an I/O request, and is put in a queue for that I/O device. Meanwhile, the CPU is given to other processes. These processes cause page faults, and, using a global replacement algorithm, one of them replaces the page containing the memory buffer for the waiting process. The pages are paged out. Some time later, when the I/O request advances to the head of the device queue, the I/O

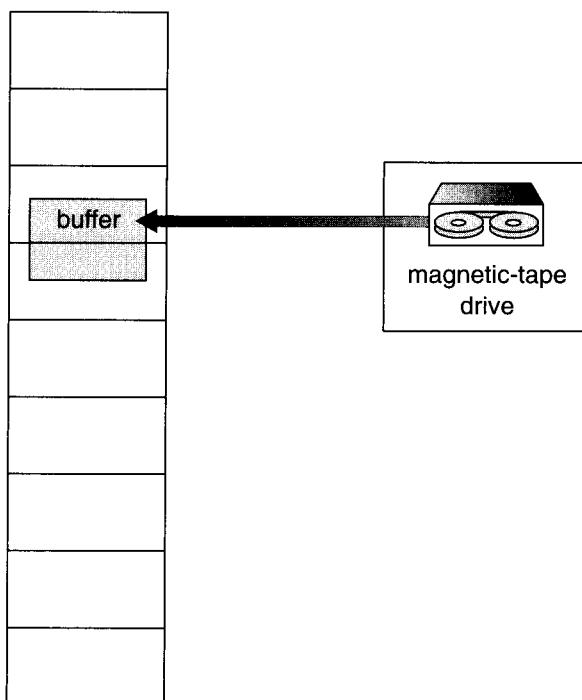


Figure 10.20 The reason why frames used for I/O must be in memory.

occurs to the specified address. However, this frame is now being used for a different page belonging to another process.

There are two common solutions to this problem. One solution is never to execute I/O to user memory. Instead, data are always copied between system memory and user memory. I/O takes place only between system memory and the I/O device. To write a block on tape, we first copy the block to system memory, and then write it to tape.

This extra copying may result in unacceptably high overhead. Another solution is to allow pages to be locked into memory. A lock bit is associated with every frame. If the frame is locked, it cannot be selected for replacement. Under this approach, to write a block on tape, we lock into memory the pages containing the block. The system can then continue as usual. Locked pages cannot be replaced. When the I/O is complete, the pages are unlocked.

Frequently, some or all of the operating-system kernel is locked into memory. Most operating systems cannot tolerate a page fault caused by the kernel. Consider the result of the page-replacement routine causing a page fault.

Another use for a lock bit involves normal page replacement. Consider the following sequence of events. A low-priority process faults. Selecting a replacement frame, the paging system reads the necessary page into memory. Ready to continue, the low-priority process enters the ready queue and waits for the CPU. Since it is a low-priority process, it may not be selected by the CPU scheduler for a while. While the low-priority process waits, a high-priority process faults. Looking for a replacement, the paging system sees a page that is in memory but has not been referenced or modified: It is the page that the low-priority process just brought in. This page looks like a perfect replacement: It is clean and will not need to be written out, and it apparently has not been used for a long time.

Whether the high-priority process should be able to replace the low-priority process is a policy decision. After all, we are simply delaying the low-priority process for the benefit of the high-priority process. On the other hand, we are wasting the effort spent to bring in the page of the low-priority process. If we decide to prevent replacing a newly brought-in page until it can be used at least once, then we can use the lock bit to implement this mechanism. When a page is selected for replacement, its lock bit is turned on; it remains on until the faulting process is again dispatched.

Using a lock bit can be dangerous, however: The lock bit may get turned on but never turned off. Should this situation occur (due to a bug in the operating system, for example), the locked frame becomes unusable. The Macintosh Operating System provides a page-locking mechanism because it is a single-user system, and the overuse of locking would hurt only the user doing the locking. Multiuser systems must be less trusting of users. For instance, Solaris 2 allows locking “hints,” but it is free to disregard these hints if the free-frame pool becomes too small or if an individual process requests that too many pages be locked in memory.

10.8.7 Real-Time Processing

The discussions in this chapter have concentrated on providing the best overall utilization of a computer system by optimizing the use of memory. By using memory for active data and moving inactive data to disk, we increase overall system throughput. However, individual processes may suffer as a result, because they now may take additional page faults during their execution.

Consider a real-time process or thread, as described in Chapter 4. Such a process expects to gain control of the CPU, and to run to completion with a minimum of delays. Virtual memory is the antithesis of real-time computing, because it can introduce unexpected long-term delays in the execution of a process while pages are brought into memory. Therefore, real-time systems almost never have virtual memory.

In the case of Solaris 2, the developers at Sun Microsystems wanted to allow both time-sharing and real-time computing within a system. To solve the page-fault problem, Solaris 2 allows a process to tell it which pages are important to that process. In addition to allowing “hints” on page use, the operating system allows privileged users to require pages to be locked into memory. If abused, this mechanism could lock all other processes out of the system. It is necessary to allow real-time processes to have bounded and low-dispatch latency.

10.9 ■ Summary

It is desirable to be able to execute a process whose logical address space is larger than the available physical address space. The programmer can make such a process executable by restructuring it using overlays, but doing so is generally a difficult programming task. Virtual memory is a technique to allow a large logical address space to be mapped onto a smaller physical memory. Virtual memory allows extremely large processes to be run, and also allows the degree of multiprogramming to be raised, increasing CPU utilization. Further, it frees application programmers from worrying about memory availability.

Pure demand paging never brings in a page until that page is referenced. The first reference causes a page fault to the operating-system resident monitor. The operating system consults an internal table to determine where the page is located on the backing store. It then finds a free frame and reads the page in from the backing store. The page table is updated to reflect this change, and the instruction that caused the page fault is restarted. This approach allows a process to run even though its entire memory image is not in main memory at once. As long as the page-fault rate is reasonably low, performance is acceptable.

We can use demand paging to reduce the number of frames allocated to a process. This arrangement can increase the degree of multiprogramming (allowing more processes to be available for execution at one time) and—in theory, at least—the CPU utilization of the system. It also allows processes to be

run even though their memory requirements exceed the total available physical memory. Such processes run in virtual memory.

If total memory requirements exceed the physical memory, then it may be necessary to replace pages from memory to free frames for new pages. Various page-replacement algorithms are used. FIFO page replacement is easy to program, but suffers from Belady's anomaly. Optimal page replacement requires future knowledge. LRU replacement is an approximation of optimal, but even it may be difficult to implement. Most page-replacement algorithms, such as the second-chance algorithm, are approximations of LRU replacement.

In addition to a page-replacement algorithm, a frame-allocation policy is needed. Allocation can be fixed, suggesting local page replacement, or dynamic, suggesting global replacement. The working-set model assumes that processes execute in localities. The working set is the set of pages in the current locality. Accordingly, each process should be allocated enough frames for its current working set.

If a process does not have enough memory for its working set, it will thrash. Providing enough frames to each process to avoid thrashing may require process swapping and scheduling.

In addition to requiring that we solve the major problems of page replacement and frame allocation, the proper design of a paging system requires that we consider page size, I/O, locking, prepaging, process creation, program structure, thrashing, and other topics. Virtual memory can be thought of as one level of a hierarchy of storage levels in a computer system. Each level has its own access time, size, and cost parameters. A full example of a hybrid, functional virtual-memory system is presented in the Mach chapter, which is available on our web site (<http://www.bell-labs.com/topic/books/os-book>).

■ Exercises

- 10.1 Under what circumstances do page faults occur? Describe the actions taken by the operating system when a page fault occurs.
- 10.2 Assume that you have a page-reference string for a process with m frames (initially all empty). The page-reference string has length p ; n distinct page numbers occur in it. Answer these questions for any page-replacement algorithms:
 - a. What is a lower bound on the number of page faults?
 - b. What is an upper bound on the number of page faults?
- 10.3 A certain computer provides its users with a virtual-memory space of 2^{32} bytes. The computer has 2^{18} bytes of physical memory. The virtual memory is implemented by paging, and the page size is 4,096 bytes. A user process generates the virtual address 11123456. Explain how

the system establishes the corresponding physical location. Distinguish between software and hardware operations.

- 10.4** Which of the following programming techniques and structures are “good” for a demand-paged environment? Which are “bad”? Explain your answers.
- Stack
 - Hashed symbol table
 - Sequential search
 - Binary search
 - Pure code
 - Vector operations
 - Indirection
- 10.5** Assume that we have a demand-paged memory. The page table is held in registers. It takes 8 milliseconds to service a page fault if an empty frame is available or if the replaced page is not modified, and 20 milliseconds if the replaced page is modified. Memory-access time is 100 nanoseconds. Assume that the page to be replaced is modified 70 percent of the time. What is the maximum acceptable page-fault rate for an effective access time of no more than 200 nanoseconds?
- 10.6** Consider the following page-replacement algorithms. Rank these algorithms on a five-point scale from “bad” to “perfect” according to their page-fault rate. Separate those algorithms that suffer from Belady’s anomaly from those that do not.
- LRU replacement
 - FIFO replacement
 - Optimal replacement
 - Second-chance replacement
- 10.7** When virtual memory is implemented in a computing system, it carries certain costs and certain benefits. List those costs and the benefits. It is possible for the costs to exceed the benefits. Explain what measures you can take to ensure that this imbalance does not occur.
- 10.8** An operating system supports a paged virtual memory, using a central processor with a cycle time of 1 microsecond. It costs an additional 1 microsecond to access a page other than the current one. Pages have 1,000 words, and the paging device is a drum that rotates at 3,000 revolutions

per minute and transfers one million words per second. The following statistical measurements were obtained from the system:

- One percent of all instructions executed accessed a page other than the current page.
- Of the instructions that accessed another page, 80 percent accessed a page already in memory.
- When a new page was required, the replaced page was modified 50 percent of the time.

Calculate the effective instruction time on this system, assuming that the system is running one process only, and that the processor is idle during drum transfers.

- 10.9** Consider a demand-paging system with the following time-measured utilizations:

CPU utilization	20%
Paging disk	97.7%
Other I/O devices	5%

For each of the following, say whether it will (or is likely to) improve CPU utilization. Explain your answers.

- a. Install a faster CPU.
 - b. Install a bigger paging disk.
 - c. Increase the degree of multiprogramming.
 - d. Decrease the degree of multiprogramming.
 - e. Install more main memory.
 - f. Install a faster hard disk, or multiple controllers with multiple hard disks.
 - g. Add prepaging to the page-fetch algorithms.
 - h. Increase the page size.
- 10.10** Consider the two-dimensional array A:

```
int A[][] = new int[100][100];
```

where A [0] [0] is stored at location 200, in a paged memory system with pages of size 200. A small process resides in page 0 (locations 0 to 199) for manipulating the A matrix; thus, every instruction fetch will be from page 0.

For three page frames, how many page faults are generated by the following array-initialization loops, using LRU replacement, and assuming page frame 1 has the process in it, and the other two are initially empty:

- a.

```
for (int j = 0; j < 100; j++)
    for (int i = 0; i < 100; i++)
        A[i][j] = 0;
```
- b.

```
for (int i = 0; i < 100; i++)
    for (int j = 0; j < 100; j++)
        A[i][j] = 0;
```

10.11 Consider the following page-reference string:

1, 2, 3, 4, 2, 1, 5, 6, 2, 1, 2, 3, 7, 6, 3, 2, 1, 2, 3, 6.

How many page faults would occur for the following replacement algorithms, assuming one, two, three, four, five, six, or seven frames? Remember that all frames are initially empty, so your first unique pages will all cost one fault each.

- LRU replacement
- FIFO replacement
- Optimal replacement

10.12 Suppose that you want to use a paging algorithm that requires a reference bit (such as second-chance replacement or working-set model), but the hardware does not provide one. Sketch how you could simulate a reference bit even if one were not provided by the hardware. Calculate the cost of doing so.

10.13 You have devised a new page-replacement algorithm that you think may be optimal. In some contorted test cases, Belady's anomaly occurs. Is the new algorithm optimal? Explain your answer.

10.14 Suppose that your replacement policy (in a paged system) is to examine each page regularly and to discard that page if it has not been used since the last examination. What would you gain and what would you lose by using this policy rather than LRU or second-chance replacement?

10.15 Segmentation is similar to paging, but uses variable-sized “pages.” Define two segment-replacement algorithms based on FIFO and LRU page-replacement schemes. Remember that, since segments are not the same size, the segment that is chosen to be replaced may not be big enough to leave enough consecutive locations for the needed segment. Consider strategies for systems where segments cannot be relocated, and those for systems where they can.

10.16 A page-replacement algorithm should minimize the number of page faults. We can do this minimization by distributing heavily used pages evenly over all of memory, rather than having them compete for a small number of page frames. We can associate with each page frame a counter of the number of pages that are associated with that frame. Then, to replace a page, we search for the page frame with the smallest counter.

- a. Define a page-replacement algorithm using this basic idea. Specifically address the problems of:
 - i. what the initial value of the counters is,
 - ii. when counters are increased,
 - iii. when counters are decreased,
 - iv. how the page to be replaced is selected.
- b. How many page faults occur for your algorithm for the following reference string, for four page frames?

1, 2, 3, 4, 5, 3, 4, 1, 6, 7, 8, 7, 8, 9, 7, 8, 9, 5, 4, 5, 4, 2.

- c. What is the minimum number of page faults for an optimal page-replacement strategy for the reference string in part b with four page frames?

10.17 Consider a demand-paging system with a paging disk that has an average access and transfer time of 20 milliseconds. Addresses are translated through a page table in main memory, with an access time of 1 microsecond per memory access. Thus, each memory reference through the page table takes two accesses. To improve this time, we have added an associative memory that reduces access time to one memory reference, if the page-table entry is in the associative memory.

Assume that 80 percent of the accesses are in the associative memory, and that, of the remaining, 10 percent (or 2 percent of the total) cause page faults. What is the effective memory access time?

10.18 Consider a demand-paged computer system where the degree of multiprogramming is currently fixed at four. The system was recently measured to determine utilization of CPU and the paging disk. The results are one of the following alternatives. For each case, what is happening? Can you increase the degree of multiprogramming to increase the CPU utilization? Is the paging helping in improving performance?

- a. CPU utilization, 13 percent; disk utilization, 97 percent
- b. CPU utilization, 87 percent; disk utilization, 3 percent
- c. CPU utilization, 13 percent; disk utilization, 3 percent

- 10.19** We have an operating system for a machine that uses base and limit registers, but we have modified the machine to provide a page table. Can we set up the page tables to simulate base and limit registers? How can we do so, or why can we not do so?
- 10.20** What is the cause of thrashing? How does the system detect thrashing? Once it detects thrashing, what can the system do to eliminate this problem?
- 10.21** Write a program that implements the FIFO and LRU page-replacement algorithms presented in this chapter. First, generate a random page-reference string where page numbers range from 0 to 9. Apply the random page-reference string to each algorithm and record the number of page faults incurred by each algorithm. Implement the replacement algorithms so that the number of page frames can vary from 1 to 7. Assume that demand paging is used.

Bibliographical Notes

Demand paging was first used in the Atlas system, implemented on the Manchester University MUSE computer around 1960 (Kilburn et al. [1961]). Another early demand-paging system was MULTICS, implemented on the GE 645 system (Organick [1972]).

Belady et al. [1969] were the first researchers to observe that the FIFO replacement strategy may have the anomaly that bears Belady's name. Mattson et al. [1970] demonstrated that stack algorithms are not subject to Belady's anomaly.

The optimal replacement algorithm was presented by Belady [1966]. It was proved to be optimal by Mattson et al. [1970]. Belady's optimal algorithm is for a fixed allocation; Prieve and Fabry [1976] have an optimal algorithm for situations where the allocation can vary.

The enhanced clock algorithm was discussed by Carr and Hennessy [1981]; it is used in the Macintosh virtual memory-management scheme, and was described by Goldman [1989].

The working-set model was developed by Denning [1968]. Discussions concerning the working-set model were presented by Denning [1980].

The scheme for monitoring the page-fault rate was developed by Wulf [1969], who successfully applied this technique to the Burroughs B5500 computer system. Gupta and Franklin [1978] provided a performance comparison between the working-set scheme and the page-fault-frequency replacement scheme.

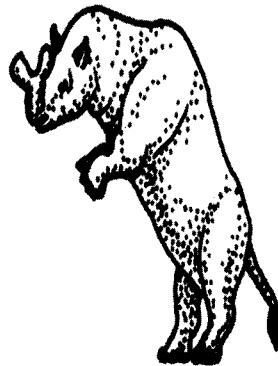
Solomon [1998] describes how Windows NT implements virtual memory. Mauro and McDougall [2001] discusses virtual memory on Solaris 2. Virtual-memory techniques in Linux and BSD are described by Bovet and Cesati [2001].

and McKusick et al. [1996] respectively. Details of OS/2 and demand segmentation were described by Iacobucci [1988]. Ganapathy and Schimmel [1998] discuss operating system support for multiple page sizes.

Good discussions are available for the Intel 80386 paging hardware Intel [1986] and the Motorola 68030 hardware Motorola [1989b]. Virtual-memory management in the VAX/VMS operating system was discussed by Levy and Lipman [1982]. Discussions concerning workstation operating systems and virtual memory were presented by Hagmann [1989]. A comparison of an implementation of virtual memory in the MIPS, PowerPC, and Pentium architectures can be found in Jacob and Mudge [1998b]. A companion article (Jacob and Mudge [1998a]) describes the hardware support necessary for implementation of virtual memory in six different architectures, including the UltraSPARC.

Chapter 11

FILE-SYSTEM INTERFACE



For most users, the file system is the most visible aspect of an operating system. It provides the mechanism for on-line storage of and access to both data and programs of the operating system and all the users of the computer system. The file system consists of two distinct parts: a collection of *files*, each storing related data, and a *directory structure*, which organizes and provides information about all the files in the system. Some file systems have a third part, *partitions*, which are used to separate physically or logically large collections of directories. In this chapter, we consider the various aspects of files and the major directory structures. We also discuss ways to handle *file protection*, necessary when multiple users have access to files and we want to control by whom and in what ways files may be accessed. Finally, we discuss the semantics of sharing files among multiple processes, users, and computers.

11.1 ■ File Concept

Computers can store information on several different storage media, such as magnetic disks, magnetic tapes, and optical disks. So that the computer system will be convenient to use, the operating system provides a uniform logical view of information storage. The operating system abstracts from the physical properties of its storage devices to define a logical storage unit (the *file*). Files are mapped, by the operating system, onto physical devices. These storage devices are usually nonvolatile, so the contents are persistent through power failures and system reboots.

A file is a named collection of related information that is recorded on secondary storage. From a user's perspective, a file is the smallest allotment of logical secondary storage; that is, data cannot be written to secondary storage unless they are within a file. Commonly, files represent programs (both source and object forms) and data. Data files may be numeric, alphabetic, alphanumeric, or binary. Files may be free form, such as text files, or may be formatted rigidly. In general, a file is a sequence of bits, bytes, lines, or records, the meaning of which is defined by the file's creator and user. The concept of a file is thus extremely general.

The information in a file is defined by its creator. Many different types of information may be stored in a file—source programs, object programs, executable programs, numeric data, text, payroll records, graphic images, sound recordings, and so on. A file has a certain defined **structure** according to its type. A *text* file is a sequence of characters organized into lines (and possibly pages). A *source* file is a sequence of subroutines and functions, each of which is further organized as declarations followed by executable statements. An *object* file is a sequence of bytes organized into blocks understandable by the system's linker. An *executable* file is a series of code sections that the loader can bring into memory and execute.

11.1.1 File Attributes

A file is named, for the convenience of its human users, and is referred to by its name. A name is usually a string of characters, such as *example.c*. Some systems differentiate between upper- and lowercase characters in names, whereas other systems consider the two cases to be equivalent. When a file is named, it becomes independent of the process, the user, and even the system that created it. For instance, one user might create the file *example.c*, whereas another user might edit that file by specifying its name. The file's owner might write the file to a floppy disk, send it in an e-mail, or copy it across a network, and it could still be called *example.c* on the destination system.

A file has certain other attributes, which vary from one operating system to another, but typically consist of these:

- **Name:** The symbolic file name is the only information kept in human-readable form.
- **Identifier:** This unique tag, usually a number, identifies the file within the file system; it is the non-human-readable name for the file.
- **Type:** This information is needed for those systems that support different types.
- **Location:** This information is a pointer to a device and to the location of the file on that device.

- **Size:** The current size of the file (in bytes, words, or blocks), and possibly the maximum allowed size are included in this attribute.
- **Protection:** Access-control information determines who can do reading, writing, executing, and so on.
- **Time, date, and user identification:** This information may be kept for creation, last modification, and last use. These data can be useful for protection, security, and usage monitoring.

The information about all files is kept in the directory structure that also resides on secondary storage. Typically, the directory entry consists of the file's name and its unique identifier. The identifier in turn locates the other file attributes. It may take more than a kilobyte to record this information for each file. In a system with many files, the size of the directory itself may be megabytes. Because directories, like files, must be nonvolatile, they must be stored on the device and brought into memory piecemeal, as needed.

11.1.2 File Operations

A file is an **abstract data type**. To define a file properly, we need to consider the operations that can be performed on files. The operating system can provide system calls to create, write, read, reposition, delete, and truncate files. Let us also consider what the operating system must do for each of the six basic file operations. It should then be easy to see how similar operations, such as renaming a file, would be implemented.

- **Creating a file:** Two steps are necessary to create a file. First, space in the file system must be found for the file. We shall discuss how to allocate space for the file in Chapter 12. Second, an entry for the new file must be made in the directory. The directory entry records the name of the file and the location in the file system, and possibly other information.
- **Writing a file:** To write a file, we make a system call specifying both the name of the file and the information to be written to the file. Given the name of the file, the system searches the directory to find the location of the file. The system must keep a *write* pointer to the location in the file where the next write is to take place. The write pointer must be updated whenever a write occurs.
- **Reading a file:** To read from a file, we use a system call that specifies the name of the file and where (in memory) the next block of the file should be put. Again, the directory is searched for the associated directory entry, and the system needs to keep a *read* pointer to the location in the file where the next read is to take place. Once the read has taken place, the read pointer is updated. A given process is usually only reading or writing a given file, and

the current operation location is kept as a per-process **current-file-position pointer**. Both the read and write operations use this same pointer, saving space and reducing the system complexity.

- **Repositioning within a file:** The directory is searched for the appropriate entry, and the current-file-position is set to a given value. Repositioning within a file does not need to involve any actual I/O. This file operation is also known as a file *seek*.
- **Deleting a file:** To delete a file, we search the directory for the named file. Having found the associated directory entry, we release all file space, so that it can be reused by other files, and erase the directory entry.
- **Truncating a file:** The user may want to erase the contents of a file but keep its attributes. Rather than forcing the user to delete the file and then recreate it, this function allows all attributes to remain unchanged—except for file length—but lets the file be reset to length zero and its file space released.

These six basic operations certainly comprise the minimal set of required file operations. Other common operations include *appending* new information to the end of an existing file and *renaming* an existing file. These primitive operations may then be combined to perform other file operations. For instance, creating a *copy* of a file, or copying the file to another I/O device, such as a printer or a display, may be accomplished by creating a new file, and reading from the old and writing to the new. We also want to have operations that allow a user to get and set the various attributes of a file. For example, we may want to have operations that allow a user to determine the status of a file, such as the file's length, and allow a user to set file attributes, such as the file's owner.

Most of the file operations mentioned involve searching the directory for the entry associated with the named file. To avoid this constant searching, many systems require that an open system call be used before that file is first used actively. The operating system keeps a small table containing information about all open files (the **open-file table**). When a file operation is requested, the file is specified via an index into this table, so no searching is required. When the file is no longer actively used, it is *closed* by the process and the operating system removes its entry in the open-file table.

Some systems implicitly open a file when the first reference is made to it. The file is automatically closed when the job or program that opened the file terminates. Most systems, however, require that the programmer open a file explicitly with the open system call before that file can be used. The open operation takes a file name and searches the directory, copying the directory entry into the open-file table. The open call can also accept access-mode information—create, read-only, read-write, append-only, and so on. This mode is checked against the file's permissions. If the request mode is allowed, the file is opened for the process. The open system call will typically return a pointer

to the entry in the open-file table. This pointer, not the actual file name, is used in all I/O operations, avoiding any further searching and simplifying the system-call interface.

The implementation of the `open` and `close` operations in a multiuser environment, such as UNIX, is more complicated. In such a system, several users may open the file at the same time. Typically, the operating system uses two levels of internal tables: a per-process table and a system-wide table. The per-process table tracks all files that a process has open. Stored in this table is information regarding the use of the file by the process. For instance, the current file pointer for each file is found here, indicating the location in the file that the next `read` or `write` call will affect. Access rights to the file and accounting information can also be included.

Each entry in the per-process table in turn points to a system-wide open-file table. The system-wide table contains process-independent information, such as the location of the file on disk, access dates, and file size. Once a file is opened by one process, another process executing an `open` call simply results in a new entry being added to the process' open-file table, pointing to the appropriate entry in the system-wide table. Typically, the open-file table also has an *open count* associated with each file, indicating the number of processes that have the file open. Each `close` decreases this *count*, and when the *open count* reaches zero, the file is no longer in use, and the file's entry is removed from the open file table. In summary, several pieces of information are associated with an open file.

- **File pointer:** On systems that do not include a file offset as part of the `read` and `write` system calls, the system must track the last read–write location as a current-file-position pointer. This pointer is unique to each process operating on the file, and therefore must be kept separate from the on-disk file attributes.
- **File open count:** As files are closed, the operating system must reuse its open-file table entries, or it could run out of space in the table. Because multiple processes may open a file, the system must wait for the last file to close before removing the open-file table entry. This counter tracks the number of opens and closes and reaches zero on the last close. The system can then remove the entry.
- **Disk location of the file:** Most file operations require the system to modify data within the file. The information needed to locate the file on disk is kept in memory to avoid having to read it from disk for each operation.
- **Access rights:** Each process opens a file in an access mode. This information is stored on the per-process table so the operating system can allow or deny subsequent I/O requests.

Some operating systems provide facilities to lock sections of an open file for multiprocess access, to share sections of a file among several processes (using shared pages), and to map sections of a file into memory on virtual-memory systems using memory mapping (Section 10.3.2).

11.1.3 File Types

When we design a file system, indeed the entire operating system, we always consider whether the operating system should recognize and support file types. If an operating system recognizes the type of a file, it can then operate on the file in reasonable ways. For example, a common mistake occurs when a user tries to print the binary-object form of a program. This attempt normally produces garbage, but can be prevented if the operating system has been told that the file is a binary-object program.

A common technique for implementing file types is to include the type as part of the file name. The name is split into two parts—a name and an *extension*, usually separated by a period character (Figure 11.1). In this way, the user and the operating system can tell from the name alone what the type of a file is. For example, in MS-DOS, a name can consist of up to eight characters followed by a period and terminated by an extension of up to three characters. The system uses the extension to indicate the *type* of the file and the type of operations that can be done on that file. For instance, only a file with a *.com*, *.exe*, or *.bat* extension can be *executed*. The *.com* and *.exe* files are two forms of binary executable files, whereas a *.bat* file is a **batch file** containing, in ASCII format, commands to the operating system. MS-DOS recognizes only a few extensions, but application programs also use extensions to indicate file types in which they are interested. For example, assemblers expect source files to have an *.asm* extension, and the WordPerfect word processor expects its file to end with a *.wp* extension. These extensions are not required, so a user may specify a file without the extension (to save typing), and the application will look for a file with the given name and the extension it expects. Because these extensions are not supported by the operating system, they can be considered as “hints” to applications that operate on them.

Another example of the utility of file types comes from the TOPS-20 operating system. If the user tries to execute an object program whose source file has been modified (or edited) since the object file was produced, the source file will be recompiled automatically. This function ensures that the user always runs an up-to-date object file. Otherwise, the user could waste a significant amount of time executing the old object file. For this function to be possible, the operating system must be able to discriminate the source file from the object file, to check the time that each file was created or last modified, and to determine the language of the source program (in order to use the correct compiler).

Consider the Apple Macintosh operating system. In this system, each file has a type, such as *text* or *pict*. Each file also has a creator attribute containing

file type	usual extension	function
executable	exe, com, bin or none	read to run machine-language program
object	obj, o	compiled, machine language, not linked
source code	c, cc, java, pas, asm, a	source code in various languages
batch	bat, sh	commands to the command interpreter
text	txt, doc	textual data, documents
word processor	wp, tex, rrf, doc	various word-processor formats
library	lib, a, so, dll, mpeg, mov, rm	libraries of routines for programmers
print or view	arc, zip, tar	ASCII or binary file in a format for printing or viewing
archive	arc, zip, tar	related files grouped into one file, sometimes compressed, for archiving or storage
multimedia	mpeg, mov, rm	binary file containing audio or A/V information

Figure 11.1 Common file types.

the name of the program that created it. This attribute is set by the operating system during the `create` call, so its use is enforced and supported by the system. For instance, a file produced by a word processor has the word processor's name as its creator. When the user opens that file, by double-clicking the mouse on the icon representing the file, the word processor is invoked automatically and the file is loaded, ready to be edited.

The UNIX system is unable to provide such a feature because it uses a crude **magic number** stored at the beginning of some files to indicate roughly the type of the file—executable program, batch file (or **shell script**), postscript file, and so on. Not all files have magic numbers, so system features cannot be based solely on this type of information. UNIX does not record the name of the creating program, either. UNIX does allow file-name-extension hints, but these extensions are not enforced or depended on by the operating system; they are mostly to aid users in determining the type of contents of the file. Extensions

can be used or ignored by a given application, but that is up to the application's programmer.

11.1.4 File Structure

File types also may be used to indicate the internal structure of the file. As mentioned in Section 11.1.3, source and object files have structures that match the expectations of the programs that read them. Further, certain files must conform to a required structure that is understood by the operating system. For example, the operating system may require that an executable file have a specific structure so that it can determine where in memory to load the file and what the location of the first instruction is. Some operating systems extend this idea into a set of system-supported file structures, with sets of special operations for manipulating files with those structures. For instance, DEC's VMS operating system has a file system that supports three defined file structures.

The above discussion brings us to one of the disadvantages of having the operating system support multiple file structures: The resulting size of the operating system is cumbersome. If the operating system defines five different file structures, it needs to contain the code to support these file structures. In addition, every file may need to be definable as one of the file types supported by the operating system. Severe problems may result from new applications that require information structured in ways not supported by the operating system.

For example, assume that a system supports two types of files: text files (composed of ASCII characters separated by a carriage return and line feed) and executable binary files. Now, if we (as users) want to define an encrypted file to protect our contents from being read by unauthorized people, we may find neither file type to be appropriate. The encrypted file is not ASCII text lines, but rather is (apparently) random bits. Although it may appear to be a binary file, it is not executable. As a result, we may have to circumvent or misuse the operating system's file-types mechanism, or abandon our encryption scheme.

Some operating systems impose (and support) a minimal number of file structures. This approach has been adopted in UNIX, MS-DOS, and others. UNIX considers each file to be a sequence of 8-bit bytes; no interpretation of these bits is made by the operating system. This scheme provides maximum flexibility, but little support. Each application program must include its own code to interpret an input file into the appropriate structure. However, all operating systems must support at least one structure—that of an executable file—so that the system is able to load and run programs.

The Macintosh operating system also supports a minimal number of file structures. It expects files to contain two parts: a **resource fork** and a **data fork**. The resource fork contains information of interest to the user. For instance, it holds the labels of any buttons displayed by the program. A foreign user may want to relabel these buttons in his own language, and

the Macintosh operating system provides tools to allow modification of the data in the resource fork. The data fork contains program code or data: the traditional file contents. To accomplish the same task on a UNIX or MS-DOS system, the programmer would need to change and recompile the source code, unless she created her own user-changeable data file. Clearly, it is useful for an operating system to support structures that will be used frequently, and that will save the programmer substantial effort. Too few structures make programming inconvenient, whereas too many cause operating-system bloat and programmer confusion.

11.1.5 Internal File Structure

Internally, locating an offset within a file can be complicated for the operating system. Recall from Chapter 2 that disk systems typically have a well-defined block size determined by the size of a sector. All disk I/O is performed in units of one block (physical record), and all blocks are the same size. It is unlikely that the physical record size will exactly match the length of the desired logical record. Logical records may even vary in length. **Packing** a number of logical records into physical blocks is a common solution to this problem.

For example, the UNIX operating system defines all files to be simply a stream of bytes. Each byte is individually addressable by its offset from the beginning (or end) of the file. In this case, the logical record is 1 byte. The file system automatically packs and unpacks bytes into physical disk blocks—say, 512 bytes per block—as necessary.

The logical record size, physical block size, and packing technique determine how many logical records are in each physical block. The packing can be done either by the user's application program or by the operating system.

In either case, the file may be considered to be a sequence of blocks. All the basic I/O functions operate in terms of blocks. The conversion from logical records to physical blocks is a relatively simple software problem.

Because disk space is always allocated in blocks, some portion of the last block of each file is generally wasted. If each block were 512 bytes, then a file of 1,949 bytes would be allocated four blocks (2,048 bytes); the last 99 bytes would be wasted. The wasted bytes allocated to keep everything in units of blocks (instead of bytes) is *internal fragmentation*. All file systems suffer from internal fragmentation; the larger the block size, the greater the internal fragmentation.

11.2 ■ Access Methods

Files store information. When it is used, this information must be accessed and read into computer memory. The information in the file can be accessed in several ways. Some systems provide only one access method for files. Other

systems, such as those of IBM, support many access methods, and choosing the right one for a particular application is a major design problem.

11.2.1 Sequential Access

The simplest access method is **sequential access**. Information in the file is processed in order, one record after the other. This mode of access is by far the most common; for example, editors and compilers usually access files in this fashion.

The bulk of the operations on a file is reads and writes. A read operation reads the next portion of the file and automatically advances a file pointer, which tracks the I/O location. Similarly, a write appends to the end of the file and advances to the end of the newly written material (the new end of file). Such a file can be reset to the beginning and, on some systems, a program may be able to skip forward or backward n records, for some integer n —perhaps only for $n = 1$. Sequential access is depicted in Figure 11.2. Sequential access is based on a tape model of a file, and works as well on sequential-access devices as it does on random-access ones.

11.2.2 Direct Access

Another method is **direct access** (or **relative access**). A file is made up of fixed-length **logical records** that allow programs to read and write records rapidly in no particular order. The direct-access method is based on a disk model of a file, since disks allow random access to any file block. For direct access, the file is viewed as a numbered sequence of blocks or records. A direct-access file allows arbitrary blocks to be read or written. Thus, we may read block 14, then read block 53, and then write block 7. There are no restrictions on the order of reading or writing for a direct-access file.

Direct-access files are of great use for immediate access to large amounts of information. Databases are often of this type. When a query concerning a particular subject arrives, we compute which block contains the answer, and then read that block directly to provide the desired information.

As a simple example, on an airline-reservation system, we might store all the information about a particular flight (for example, flight 713) in the block

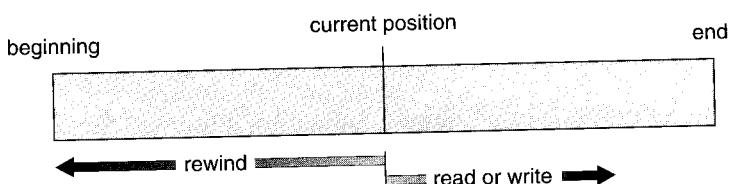


Figure 11.2 Sequential-access file.

identified by the flight number. Thus, the number of available seats for flight 713 is stored in block 713 of the reservation file. To store information about a larger set, such as people, we might compute a hash function on the people's names, or search a small in-memory index to determine a block to read and search.

For the direct-access method, the file operations must be modified to include the block number as a parameter. Thus, we have *read n*, where *n* is the block number, rather than *read next*, and *write n* rather than *write next*. An alternative approach is to retain *read next* and *write next*, as with sequential access, and to add an operation *position file to n*, where *n* is the block number. Then, to effect a *read n*, we would *position to n* and then *read next*.

The block number provided by the user to the operating system is normally a **relative block number**. A relative block number is an index relative to the beginning of the file. Thus, the first relative block of the file is 0, the next is 1, and so on, even though the actual absolute disk address of the block may be 14703 for the first block and 3192 for the second. The use of relative block numbers allows the operating system to decide where the file should be placed (called the *allocation problem*, as discussed in Chapter 12), and helps to prevent the user from accessing portions of the file system that may not be part of his file. Some systems start their relative block numbers at 0; others start at 1.

Given a logical record length *L*, a request for record *N* is turned into an I/O request for *L* bytes starting at location *L * (N - 1)* within the file (assuming first record is *N = 1*). Since logical records are of a fixed size, it is also easy to read, write, or delete a record.

Not all operating systems support both sequential and direct access for files. Some systems allow only sequential file access; others allow only direct access. Some systems require that a file be defined as sequential or direct when it is created; such a file can be accessed only in a manner consistent with its declaration. However, it is easy to simulate sequential access on a direct-access file. If we simply keep a variable *cp* that defines our current position, then we can simulate sequential file operations, as shown in Figure 11.3. On the other hand, it is extremely inefficient and clumsy to simulate a direct-access file on a sequential-access file.

sequential access	implementation for direct access
<i>reset</i>	<i>cp = 0;</i>
<i>read next</i>	<i>read cp;</i> <i>cp = cp+1;</i>
<i>write next</i>	<i>write cp;</i> <i>cp = cp+1;</i>

Figure 11.3 Simulation of sequential access on a direct-access file..

11.2.3 Other Access Methods

Other access methods can be built on top of a direct-access method. These methods generally involve the construction of an index for the file. The **index**, like an index in the back of a book, contains pointers to the various blocks. To find a record in the file, we first search the index, and then use the pointer to access the file directly and to find the desired record.

For example, a retail-price file might list the universal product codes (UPCs) for items, with the associated prices. Each record consists of a 10-digit UPC and a 6-digit price, for a 16-byte record. If our disk has 1,024 bytes per block, we can store 64 records per block. A file of 120,000 records would occupy about 2,000 blocks (2 million bytes). By keeping the file sorted by UPC, we can define an index consisting of the first UPC in each block. This index would have 2,000 entries of 10 digits each, or 20,000 bytes, and thus could be kept in memory. To find the price of a particular item, we can (binary) search the index. From this search, we would know exactly which block contains the desired record and access that block. This structure allows us to search a large file doing little I/O.

With large files, the index file itself may become too large to be kept in memory. One solution is to create an index for the index file. The primary index file would contain pointers to secondary index files, which would point to the actual data items.

For example, IBM's indexed sequential-access method (ISAM) uses a small master index that points to disk blocks of a secondary index. The secondary index blocks point to the actual file blocks. The file is kept sorted on a defined key. To find a particular item, we first make a binary search of the master index, which provides the block number of the secondary index. This block is read in, and again a binary search is used to find the block containing the desired

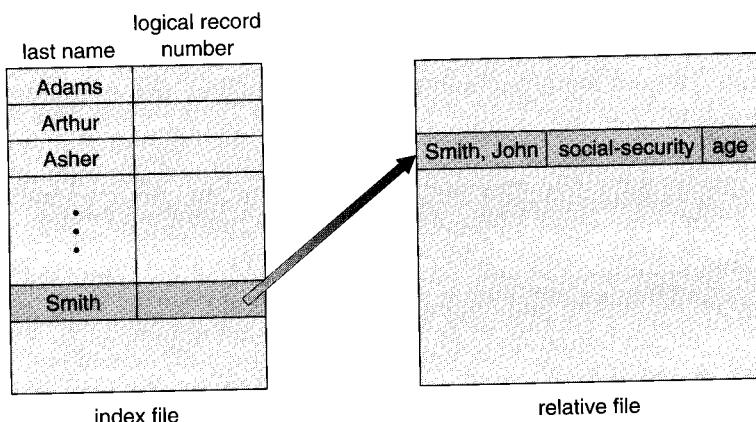


Figure 11.4 Example of index and relative files.

record. Finally, this block is searched sequentially. In this way, any record can be located from its key by at most two direct-access reads. Figure 11.4 shows a similar situation as implemented by VMS index and relative files.

11.3 ■ Directory Structure

The file systems of computers can be extensive. Some systems store millions of files on terabytes of disk. To manage all these data, we need to organize them. This organization is usually done in two parts. First, disks are split into one or more *partitions*, also known as *minidisks* in the IBM world or *volumes* in the PC and Macintosh arenas. Typically, each disk on a system contains at least one partition, which is a low-level structure in which files and directories reside. Sometimes, partitions are used to provide several separate areas within one disk, each treated as a separate storage device, whereas other systems allow partitions to be larger than a disk to group disks into one logical structure. In this way, the user needs to be concerned with only the logical directory and file structure, and can ignore completely the problems of physically allocating space for files. For this reason partitions can be thought of as virtual disks. Partitions can also store multiple operating systems, allowing a system to boot and run more than one.

Second, each partition contains information about files within it. This information is kept in entries in a **device directory** or **volume table of contents**. The device directory (more commonly known simply as a *directory*) records information—such as name, location, size, and type—for all files on that partition. Figure 11.5 shows the typical file-system organization.

The directory can be viewed as a symbol table that translates file names into their directory entries. If we take such a view, we see that the directory

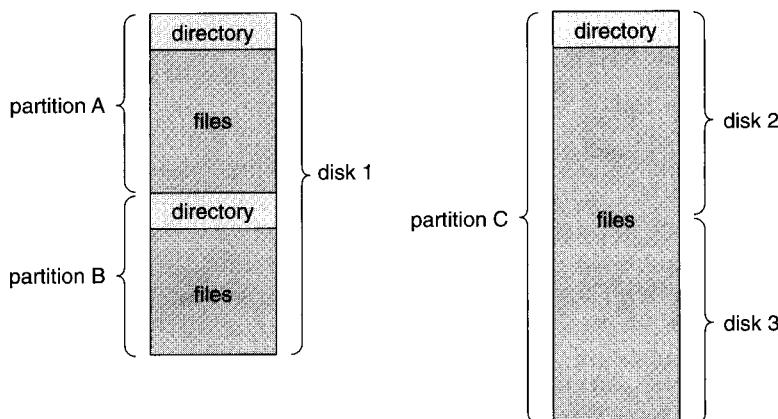


Figure 11.5 A typical file-system organization.

itself can be organized in many ways. We want to be able to insert entries, to delete entries, to search for a named entry, and to list all the entries in the directory. In Chapter 12, we discuss the appropriate data structures that can be used in the implementation of the directory structure. In this section, we examine several schemes for defining the logical structure of the directory system. When considering a particular directory structure, we need to keep in mind the operations that are to be performed on a directory:

- **Search for a file:** We need to be able to search a directory structure to find the entry for a particular file. Since files have symbolic names and similar names may indicate a relationship between files, we may want to be able to find all files whose names match a particular pattern.
- **Create a file:** New files need to be created and added to the directory.
- **Delete a file:** When a file is no longer needed, we want to remove it from the directory.
- **List a directory:** We need to be able to list the files in a directory, and the contents of the directory entry for each file in the list.
- **Rename a file:** Because the name of a file represents its contents to its users, the name must be changeable when the contents or use of the file changes. Renaming a file may also allow its position within the directory structure to be changed.
- **Traverse the file system:** We may wish to access every directory, and every file within a directory structure. For reliability, it is a good idea to save the contents and structure of the entire file system at regular intervals. This saving often consists of copying all files to magnetic tape. This technique provides a backup copy in case of system failure or if the file is simply no longer in use. In this case, the file can be copied to tape, and the disk space of that file released for reuse by another file.

In Sections 11.3.1 through 11.3.5, we describe the most common schemes for defining the logical structure of a directory.

11.3.1 Single-Level Directory

The simplest directory structure is the single-level directory. All files are contained in the same directory, which is easy to support and understand (Figure 11.6).

A single-level directory has significant limitations, however, when the number of files increases or when the system has more than one user. Since all files are in the same directory, they must have unique names. If two users call their data file *test*, then the unique-name rule is violated. For example,

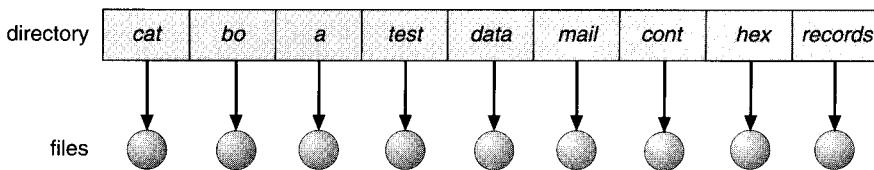


Figure 11.6 Single-level directory.

in one programming class, 23 students called the program for their second assignment *prog2*; another 11 called it *assign2*. Although file names are generally selected to reflect the content of the file, they are often limited in length. The MS-DOS operating system allows only 11-character file names; UNIX allows 255 characters.

Even a single user on a single-level directory may find it difficult to remember the names of all the files, as the number of files increases. It is not uncommon for a user to have hundreds of files on one computer system and an equal number of additional files on another system. In such an environment, keeping track of so many files is a daunting task.

11.3.2 Two-Level Directory

A single-level directory often leads to confusion of file names between different users. The standard solution is to create a *separate* directory for each user.

In the two-level directory structure, each user has her own **user file directory (UFD)**. Each UFD has a similar structure, but lists only the files of a single user. When a user job starts or a user logs in, the system's **master file directory (MFD)** is searched. The MFD is indexed by user name or account number, and each entry points to the UFD for that user (Figure 11.7).

When a user refers to a particular file, only his own UFD is searched. Thus, different users may have files with the same name, as long as all the file names within each UFD are unique.

To create a file for a user, the operating system searches only that user's UFD to ascertain whether another file of that name exists. To delete a file, the operating system confines its search to the local UFD; thus, it cannot accidentally delete another user's file that has the same name.

The user directories themselves must be created and deleted as necessary. A special system program is run with the appropriate user name and account information. The program creates a new UFD and adds an entry for it to the MFD. The execution of this program might be restricted to system administrators. The allocation of disk space for user directories can be handled with the techniques discussed in Chapter 12 for files themselves.

Although the two-level directory structure solves the name-collision problem, it still has disadvantages. This structure effectively isolates one user from another. This isolation is an advantage when the users are completely indepen-

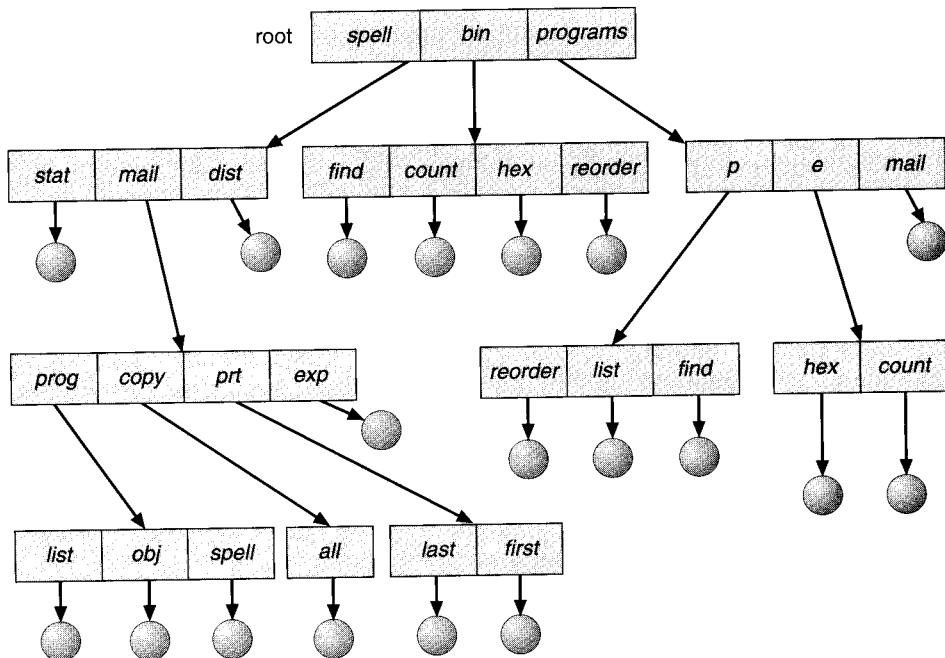


Figure 11.8 Tree-structured directory structure.

some other predefined location) to find an entry for this user (for accounting purposes). In the accounting file is a pointer to (or the name of) the user's initial directory. This pointer is copied to a local variable for this user that specifies the user's initial current directory.

Path names can be of two types: *absolute* path names or *relative* path names. An **absolute path name** begins at the root and follows a path down to the specified file, giving the directory names on the path. A **relative path name** defines a path from the current directory. For example, in the tree-structured file system of Figure 11.8, if the current directory is *root/spell/mail*, then the relative path name *prt/first* refers to the same file as does the absolute path name *root/spell/mail/prt/first*.

Allowing the user to define his own subdirectories permits him to impose a structure on his files. This structure might result in separate directories for files associated with different topics (for example, a subdirectory was created to hold the text of this book) or different forms of information (for example, the directory *programs* may contain source programs; the directory *bin* may store all the binaries).

An interesting policy decision in a tree-structured directory structure is how to handle the deletion of a directory. If a directory is empty, its entry in its containing directory can simply be deleted. However, suppose the directory to be deleted is not empty, but contains several files or subdirectories: One of

two approaches can be taken. Some systems, such as MS-DOS, will not delete a directory unless it is empty. Thus, to delete a directory, the user must first delete all the files in that directory. If any subdirectories exist, this procedure must be applied recursively to them, so that they can be deleted also. This approach may result in a substantial amount of work.

An alternative approach, such as that taken by the UNIX `rm` command, is to provide the option that, when a request is made to delete a directory, all that directory's files and subdirectories are also to be deleted. Either approach is fairly easy to implement; the choice is one of policy. The latter policy is more convenient, but more dangerous, because an entire directory structure may be removed with one command. If that command were issued in error, a large number of files and directories would need to be restored from backup tapes.

With a tree-structured directory system, users can access, in addition to their files, the files of other users. For example, user B can access files of user A by specifying their path names. User B can specify either an absolute or a relative path name. Alternatively, user B could change her current directory to be user A's directory, and access the files by their file names. Some systems also allow users to define their own search paths. In this case, user B could define her search path to be (1) her local directory, (2) the system file directory, and (3) user A's directory, in that order. As long as the name of a file of user A did not conflict with the name of a local file or system file, it could be referred to simply by its name.

A path to a file in a tree-structured directory can be longer than that in a two-level directory. To allow users to access programs without having to remember these long paths, the Macintosh operating system automates the search for executable programs. It maintains a file, called the *Desktop File*, containing the name and location of all executable programs it has seen. When a new hard disk or floppy disk is added to the system, or the network is accessed, the operating system traverses the directory structure, searching for executable programs on the device and recording the pertinent information. This mechanism supports the double-click execution functionality described previously. A double-click on a file causes its creator attribute to be read, and the *Desktop File* to be searched for a match. Once the match is found, the appropriate executable program is started with the clicked-on file as its input. The Microsoft Windows family of operating systems (95, 95, NT, 2000) maintains an extended two-level directory structure, with devices and partitions assigned a drive letter (Section 11.4).

11.3.4 Acyclic-Graph Directories

Consider two programmers who are working on a joint project. The files associated with that project can be stored in a subdirectory, separating them from other projects and files of the two programmers. But since both programmers are equally responsible for the project, both want the subdirectory to be in

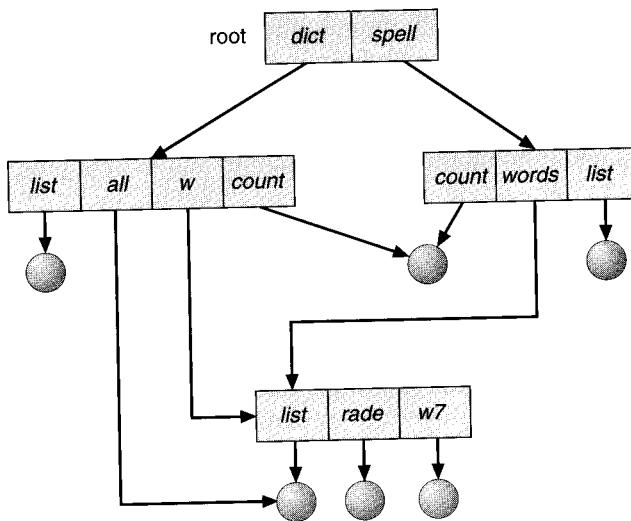


Figure 11.9 Acyclic-graph directory structure.

their own directories. The common subdirectory should be *shared*. A shared directory or file will exist in the file system in two (or more) places at once.

A tree structure prohibits the sharing of files or directories. An **acyclic graph** allows directories to have shared subdirectories and files (Figure 11.9). The *same* file or subdirectory may be in two different directories. An acyclic graph, that is, a graph with no cycles, is a natural generalization of the tree-structured directory scheme.

A shared file (or directory) is not the same as two copies of the file. With two copies, each programmer can view the copy rather than the original, but if one programmer changes the file, the changes will not appear in the other's copy. With a shared file, only *one* actual file exists, so any changes made by one person are immediately visible to the other. Sharing is particularly important for subdirectories; a new file created by one person will automatically appear in all the shared subdirectories.

When people are working as a team, all the files they want to share may be put into one directory. The UFDs of all the team members would each contain this directory of shared files as a subdirectory. Even when there is a single user, his file organization may require that some files be put into different subdirectories. For example, a program written for a particular project should be both in the directory of all programs and in the directory for that project.

Shared files and subdirectories can be implemented in several ways. A common way, exemplified by many of the UNIX systems, is to create a new directory entry called a link. A **link** is effectively a pointer to another file or subdirectory. For example, a link may be implemented as an absolute or relative path name. When a reference to a file is made, we search the directory. If the

directory entry is marked as a link, then the name of the real file (or directory) is given. We *resolve* the link by using the path name to locate the real file. Links are easily identified by their format in the directory entry (or by their having a special type on systems that support types), and are effectively named indirect pointers. The operating system ignores these links when traversing directory trees to preserve the acyclic structure of the system.

Another common approach to implementing shared files is simply to duplicate all information about them in both sharing directories. Thus, both entries are identical and equal. A link is clearly different from the original directory entry; thus, the two are not equal. Duplicate directory entries, however, make the original and the copy indistinguishable. A major problem with duplicate directory entries is maintaining consistency if the file is modified.

An acyclic-graph directory structure is more flexible than is a simple tree structure, but it is also more complex. Several problems must be considered carefully. A file may now have multiple absolute path names. Consequently, distinct file names may refer to the same file. This situation is similar to the aliasing problem for programming languages. If we are trying to traverse the entire file system—to find a file, to accumulate statistics on all files, or to copy all files to backup storage—this problem becomes significant, since we do not want to traverse shared structures more than once.

Another problem involves deletion. When can the space allocated to a shared file be deallocated and reused? One possibility is to remove the file whenever anyone deletes it, but this action may leave dangling pointers to the now-nonexistent file. Worse, if the remaining file pointers contain actual disk addresses, and the space is subsequently reused for other files, these dangling pointers may point into the middle of other files.

In a system where sharing is implemented by symbolic links, this situation is somewhat easier to handle. The deletion of a link does not need to affect the original file; only the link is removed. If the file entry itself is deleted, the space for the file is deallocated, leaving the links dangling. We can search for these links and remove them also, but unless a list of the associated links is kept with each file, this search can be expensive. Alternatively, we can leave the links until an attempt is made to use them. At that time, we can determine that the file of the name given by the link does not exist, and can fail to resolve the link name; the access is treated just like any other illegal file name. (In this case, the system designer should consider carefully what to do when a file is deleted and another file of the same name is created, before a symbolic link to the original file is used.) In the case of UNIX, symbolic links are left when a file is deleted, and it is up to the user to realize that the original file is gone or has been replaced. Microsoft Windows (all flavors) uses the same approach.

Another approach to deletion is to preserve the file until all references to it are deleted. To implement this approach, we must have some mechanism for determining that the last reference to the file has been deleted. We could keep a list of all references to a file (directory entries or symbolic links). When a

link or a copy of the directory entry is established, a new entry is added to the file-reference list. When a link or directory entry is deleted, we remove its entry on the list. The file is deleted when its file-reference list is empty.

The trouble with this approach is the variable and potentially large size of the file-reference list. However, we really do not need to keep the entire list—we need to keep only a count of the *number* of references. A new link or directory entry increments the reference count; deleting a link or entry decrements the count. When the count is 0, the file can be deleted; there are no remaining references to it. The UNIX operating system uses this approach for nonsymbolic links (or **hard links**), keeping a reference count in the file information block (or *inode*, see Appendix A.7.2). By effectively prohibiting multiple references to directories, we maintain an acyclic-graph structure.

To avoid these problems, some systems do not allow shared directories or links. For example, in MS-DOS, the directory structure is a tree structure, rather than an acyclic graph.

11.3.5 General Graph Directory

One serious problem with using an acyclic-graph structure is ensuring that there are no cycles. If we start with a two-level directory and allow users to create subdirectories, a tree-structured directory results. It should be fairly easy to see that simply adding new files and subdirectories to an existing tree-structured directory preserves the tree-structured nature. However, when we add links to an existing tree-structured directory, the tree structure is destroyed, resulting in a simple graph structure (Figure 11.10).

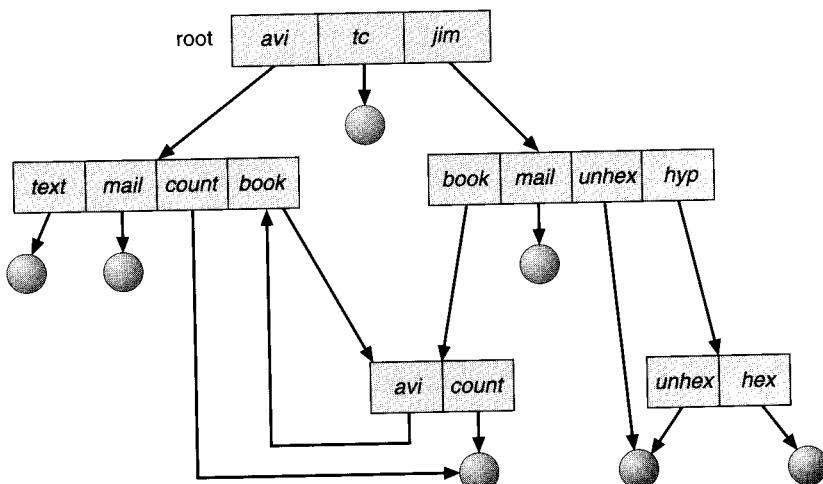


Figure 11.10 General graph directory.

The primary advantage of an acyclic graph is the relative simplicity of the algorithms to traverse the graph and to determine when there are no more references to a file. We want to avoid traversing shared sections of an acyclic graph twice, mainly for performance reasons. If we have just searched a major shared subdirectory for a particular file, without finding it, we want to avoid searching that subdirectory again; the second search would be a waste of time.

If cycles are allowed to exist in the directory, we likewise want to avoid searching any component twice, for reasons of correctness as well as performance. A poorly designed algorithm might result in an infinite loop continually searching through the cycle and never terminating. One solution is arbitrarily to limit the number of directories that will be accessed during a search.

A similar problem exists when we are trying to determine when a file can be deleted. As with acyclic-graph directory structures, a value zero in the reference count means that there are no more references to the file or directory, and the file can be deleted. However, when cycles exist, the reference count may be nonzero, even when it is no longer possible to refer to a directory or file. This anomaly results from the possibility of self-referencing (or a cycle) in the directory structure. In this case, we generally need to use a garbage-collection scheme to determine when the last reference has been deleted and the disk space can be reallocated. Garbage collection involves traversing the entire file system, marking everything that can be accessed. Then, a second pass collects everything that is not marked onto a list of free space. (A similar marking procedure can be used to ensure that a traversal or search will cover everything in the file system once and only once.) Garbage collection for a disk-based file system, however, is extremely time-consuming and is thus seldom attempted.

Garbage collection is necessary only because of possible cycles in the graph. Thus, an acyclic-graph structure is much easier to work with. The difficulty is to avoid cycles as new links are added to the structure. How do we know when a new link will complete a cycle? There are algorithms to detect cycles in graphs; however, they are computationally expensive, especially when the graph is on disk storage. A simpler algorithm in the special case of directories and links is to bypass links during directory traversal. Cycles are avoided and no extra overhead is incurred.

11.4 ■ File-System Mounting

Just as a file must be *opened* before it is used, a file system must be *mounted* before it can be available to processes on the system. More specifically, the directory structure can be built out of multiple partitions, which must be mounted to make them available within the file system name space.

The mount procedure is straightforward. The operating system is given the name of the device, and the location within the file structure at which to attach

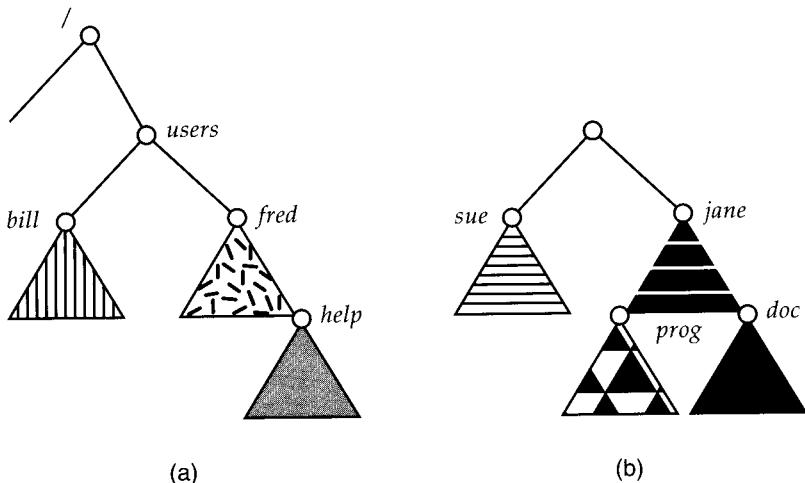


Figure 11.11 File system. (a) Existing. (b) Unmounted partition.

the file system (or **mount point**). Typically, a mount point is an empty directory at which the mounted file system will be attached. For instance, on a UNIX system, a file system containing user's home directories might be mounted as */home*; then, to access the directory structure within that file system, one could precede the directory names with */home*, as in */home/jane*. Mounting that file system under */users* would result in the path name */users/jane* to reach the same directory.

Next, the operating system verifies that the device contains a valid file system. It does so by asking the device driver to read the device directory and verifying that the directory has the expected format. Finally, the operating system notes in its directory structure that a file system is mounted at the specified mount point. This scheme enables the operating system to traverse its directory structure, switching among file systems as appropriate.

To illustrate file mounting, consider the file system depicted in Figure 11.11, where the triangles represent subtrees of directories that are of interest. In Figure 11.11(a), an existing file system is shown, while in Figure 11.11(b), an unmounted partition residing on */device/dsk* is shown. At this point, only the files on the existing file system can be accessed. In Figure 11.12, the effects of the mounting of the partition residing on */device/dsk* over */users* are shown. If the partition is unmounted, the file system is restored to the situation depicted in Figure 11.11.

Systems impose semantics to clarify functionality. For example, a system may disallow a mount over a directory that contains files, or make the mounted file system available at that directory and obscure the directory's existing files until the file system is *unmounted*, terminating the use of the file system and allowing access to the original files in that directory. As another example, a

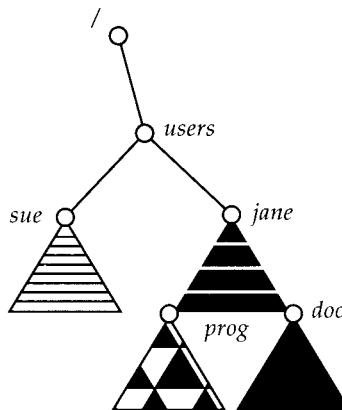


Figure 11.12 Mount point.

system may allow the same file system to be mounted repeatedly, at different mount points, or it may only allow one mount per file system.

Consider the actions of the Macintosh operating system. Whenever the system encounters a disk for the first time (hard disks are found at boot time, floppy disks are seen when they are inserted into the drive), the Macintosh operating system searches for a file system on the device. If it finds one, it automatically mounts the file system at the root level, adding a folder icon on the screen labeled with the name of the file system (as stored in the device directory). The user then is able to click on the icon and thus to display the newly mounted file system.

The Microsoft Windows family of operating systems (95, 98, NT, and 2000) maintains an extended two-level directory structure, with devices and partitions assigned a drive letter. Partitions have a general graph directory structure associated with the drive letter. The path to a specific file is then in the form of *drive-letter:\path\to\file*. These operating systems automatically discover all devices and mount all located file systems at boot time. In some systems, like UNIX, the mount commands are explicit. A system configuration file contains a list of devices and mount points for automatic mounting at boot time, but other mounts may be executed manually.

File system mounting is further discussed in Sections 12.2.2 and A.7.5.

11.5 ■ File Sharing

In the previous sections, we explored the motivation for file sharing and some of the difficulties involved in allowing users to share files. Such file sharing is very desirable for users who want to collaborate and to reduce the effort required to achieve a computing goal. Therefore, user-oriented operating systems must accommodate the need to share files in spite of the inherent difficulties.

In this section, we examine more aspects of file sharing. First is the topic of multiple users and the sharing methods possible. Once multiple users are allowed to share files, the challenge is to extend sharing to multiple file systems, including remote file systems. Finally, there can be several interpretations of conflicting actions occurring on shared files. For instance, if multiple users are writing to the file, should all the writes be allowed to occur, or should the operating system protect the user actions from each other? Consistency semantics is discussed in Section 11.5.3.

11.5.1 Multiple Users

When an operating system accommodates multiple users, the issues of file sharing, file naming, and file protection become preeminent. Given a directory structure that allows files to be shared by users, the system must mediate the file sharing. The system either can allow a user to access the files of other users by default, or it may require that a user specifically grant access to the files. These are the issues of access control and protection, which are covered below.

To implement sharing and protection, the system must maintain more file and directory attributes than on a single-user system. Although there have been many approaches to this topic historically, most systems have evolved to the concepts of file/directory *owner* (or *user*) and *group*. The owner is the user who may change attributes, grant access, and has the most control over the file or directory. The group attribute of a file is used to define a subset of users who may share access to the file. For example, the owner of a file on a UNIX system may issue all operations on a file, while members of the file's group may execute one subset of those operations, and all other users may execute another subset of operations. Exactly which operations can be executed by group members and other users is definable by the file's owner. More details on permission attributes are included in the next section.

Most systems implement owner attributes by managing a list of user names and associated **user identifiers** (**user IDs**). In Windows NT parlance, this is a **Security ID (SID)**. These numerical IDs are unique, one per user. When a user logs in to the system, the authentication stage determines the appropriate user ID for the user. That user ID is associated with all of the user's processes and threads. When they need to be user readable, they are translated back to the user name via the user name list. Likewise, group functionality can be implemented as a system-wide list of group names and **group identifiers**. Every user can be in one or more groups, depending upon operating system design decisions. The user's group IDs are also included in every associated process and thread.

The owner and group IDs of a given file or directory are stored with the other file attributes. When a user requests an operation on a file, the user ID can be compared to the owner attribute to determine if the requesting user is the owner of the file. Likewise, the group IDs can be compared. The result indicates

which permissions are applicable. The system then applies those permissions to the requested operation, and allows or denies it.

The user information within a process can be used for other purposes as well. One process may attempt to interact with another process, and user information can dictate the result, based on the design of the operating system. For example, a process may attempt to terminate, background, or lower the priority of another process. If the owner of each process is the same, then the command may succeed, or else it may be denied. It may also be allowed to succeed if it is owned by the privileged user.

Many systems have multiple local file systems, including partitions of a single disk or multiple partitions on multiple attached disks. In these cases, the ID checking and permission matching are straightforward, once the file systems are mounted.

11.5.2 Remote File Systems

The advent of networks (Chapter 15) allowed communication between remote computers. Networking allows the sharing of resources spread within a campus or even around the world. One obvious resource to share is data, in the form of files. Through the evolution of network and file technology, file-sharing methods have changed. In the first implemented method, users manually transfer files between machines via programs like `ftp`. The second major method is a **distributed file system (DFS)** in which remote directories are visible from the local machine. In some ways, the third method, the **World Wide Web**, is a reversion to the first. A browser is needed to gain access to the remote files, and separate operations (essentially a wrapper for `ftp`) are used to transfer files.

`ftp` is used for both anonymous and authenticated access. **Anonymous access** allows a user to transfer files without having an account on the remote system. The World Wide Web uses anonymous file exchange almost exclusively. DFS involve a much tighter integration between the machine that is accessing the remote files and the machine providing the files. This integration adds complexity, which we describe in this section.

11.5.2.1 The Client-Server Model

Remote file systems allow a computer to mount one or more file systems from one or more remote machines. In this case, the machine containing the files is the *server*, and the machine wanting access to the files is the *client*. The client-server relationship is common with networked machines. Generally, the server declares that a resource is available to clients and specifies exactly which resource (in this case, which files) and exactly which clients. Files are usually specified on a partition or subdirectory level. A server can serve multiple clients, and a client can use multiple servers, depending on the implementation details of a given client-server facility.

Client identification is more difficult. Clients can be specified by their network name or other identifier, such as *IP address*, but these can be **spoofed** (or imitated). An unauthorized client can spoof the server into deciding that it is authorized, and the unauthorized client could be allowed access. More secure solutions include secure authentication of the client to the server via encrypted keys. Unfortunately, with security comes many challenges, including ensuring compatibility of the client and server (they must use the same encryption algorithms) and secure key exchanges (intercepted keys could again allow unauthorized client access). These problems are difficult enough that, most commonly, unsecure authentication methods are used. In the case of UNIX and its network file system (NFS), authentication is via the client networking information, by default. In this scheme, the user IDs must match on the client and server. If not, the server will be unable to determine access rights to files.

Consider the example of a user who has the ID of 1000 on the client and 2000 on the server. A request from the client to the server for a specific file will not be handled appropriately, as the server will determine if user 1000 has access to the file, rather than the *real* user ID of 2000. Access would be granted or denied based on incorrect authentication information. The server must trust the client to present the correct user ID. The NFS protocols allow many-to-many relationships. That is, many servers can provide files to many clients. In fact, a given machine can be both a server to other NFS clients and a client of other NFS servers.

Once the remote file system is mounted, file operation requests are sent on the behalf of the user, across the network, to the server, via the DFS protocol. Typically, a file open request is sent along with the ID of the requesting user. The server then applies the standard access checks to determine if the user has credentials to access the file in the mode requested. The request is either allowed or denied. If it is allowed, a file handle is returned to the client application, and the application then may perform read, write, and other operations on the file. The client closes the file when access is completed. The operating system may apply semantics similar to those for a local file system mount, or may have different semantics.

11.5.2.2 Distributed Information Systems

To ease the management of client-server services, **distributed information systems**, also known as **distributed naming services**, have been devised to provide a unified access to the information needed for remote computing. **Domain name system (DNS)** provides host-name-to-network-address translations for the entire Internet (including the World Wide Web). Before DNS was invented and became widespread, files containing the same information were sent via e-mail or *f tp* between all networked hosts. This methodology was not scalable. DNS is further discussed in 15.4.1.

Other distributed information systems provide *user name/password/user ID/group ID* space for a distributed facility. UNIX systems have had a wide

variety of distributed information methods. Sun Microsystems introduced *yellow pages* (since renamed to **network information service (NIS)**), and most of the industry adopted its use. It centralizes storage of user names, host names, printer information, and the like. Unfortunately, it uses unsecure authentication methods, including sending user passwords unencrypted (in *clear text*) and identifying hosts by IP address. Sun's NIS+ is a much more secure replacement for NIS, but is also much more complicated and has not been widely adopted.

In the case of Microsoft networks (**CIFS**), network information is used in conjunction with user authentication (user name and password) to create a **network login** that the server uses to decide whether to allow or deny access to a requested file system. For this authentication to be valid, the user names must match between the machines (as with NFS). Microsoft uses two distributed-naming structures to provide a single namespace for users. The older naming technology is **domains**. The newer technology, available in Windows 2000 and beyond, is **active directory**. Once established, the distributed-naming facility is used by all clients and servers to authenticate users.

The industry is moving toward **lightweight directory-access protocol (LDAP)** as a secure, distributed naming mechanism. In fact, active directory is based on LDAP. Sun Microsystems' Solaris 8 allows LDAP to be used for user authentication as well as system-wide retrieval of information such as available printers. If the convergence of the use of LDAP succeeds, then one distributed LDAP directory will be used by an organization to store all user and resource information for all computers within that organization. The result would be **secure single sign-on** for users, who would enter their authentication information once for access to all computers within the organization. It would also ease systems-administration efforts by combining, into one location, information that is currently scattered in various files on each system or in differing distributed information services.

11.5.2.3 Failure Modes

Local file systems can fail for a variety of reasons, including failure of the disk containing the file system, corruption of the directory structure or other disk management information (collectively called **metadata**), disk-controller failure, cable failure, or host adapter failure. User or systems-administrator failure can also cause files to be lost, or entire directories or partitions to be deleted. Many of these failures would cause a host to crash and an error condition to be displayed, and require human intervention to repair.

Some failures do not cause loss of data or loss of availability of data. **Redundant arrays of inexpensive disks (RAID)** can prevent the loss of a disk from resulting in the loss of data. RAID is covered further in Section 14.5.

Remote file systems have more failure modes. By nature of the complexity of network systems and the required interactions between remote machines, many more problems can interfere with the proper operation of remote file systems. In the case of networks, the network can be interrupted between

the two hosts. This could be due to hardware failure or misconfiguration, or networking implementation issues at any of the involved hosts. Although some networks have built-in resiliency, including multiple paths between each host, many do not. Any single failure could interrupt the flow of DFS commands.

Consider a client in the midst of using a remote file system. It has remote file systems mounted and may have files open from the remote host; among other activities, it may be performing directory lookups to open files, reading or writing data to files, and closing files. Now consider a partitioning of the network, a crash of the server, or even a scheduled shutdown of that server, such that the remote file system is no longer reachable. This scenario is rather common, so it would not be appropriate for the client to act as it would in the case of a loss of a local file system.

Rather, the system could either terminate all operations to the lost server, or delay operations until the server is again reachable. This failure semantics is defined and implemented as part of the remote file system protocol. Termination of all operations can result in users losing data, and patience. Most DFS protocols either enforce or allow delaying of file-system operations to remote hosts, with the hope that the remote host will become available again.

For this kind of recovery from failure, some kind of *state* information may be maintained on both the client and server. If the server has crashed, but must recognize that it had exported file systems, remotely mounted them, and opened certain files, NFS takes a simple approach, implementing a **stateless** DFS. In essence, it assumes that a client request for a file read or write would not have occurred unless the file system had been remotely mounted and the file had been previously open. The NFS protocol carries all the information needed to locate the appropriate file and perform the requested operation on a file. Likewise, it does not track which clients have its exported partitions mounted, again assuming that if a request comes it, it must be legitimate. While this stateless approach makes NFS resilient and rather easy to implement, it makes it unsecure. For example, forged read or write requests could be allowed by an NFS server even though the requisite mount request and permission check have not take place.

11.5.3 Consistency Semantics

Consistency semantics is an important criterion for evaluating any file system that supports file sharing. It is a characterization of the system that specifies the semantics of multiple users accessing a shared file simultaneously. In particular, these semantics should specify when modifications of data by one user are observable by other users. The semantics are typically implemented as code with the file system.

Consistency semantics are directly related to the process synchronization algorithms of Chapter 7. However, the complex algorithms of that chapter tend not to be implemented in the case of file I/O because of the great latencies and

slow transfer rates of disks and networks. For example, performing an atomic transaction to a remote disk could involve several network communications or several disk reads and writes, or both. Systems that attempt such a full set of functionalities tend to perform poorly. A successful implementation of complex sharing semantics can be found in the Andrew File System.

For the following discussion, we assume that a series of file accesses (that is, reads and writes) attempted by a user to the same file is always enclosed between the `open` and `close` operations. The series of accesses between the `open` and `close` operations is a **file session**. To illustrate the concept, we sketch several prominent examples of consistency semantics.

11.5.4 UNIX Semantics

The UNIX file system (Chapter 16) uses the following consistency semantics:

- Writes to an open file by a user are visible immediately to other users that have this file open at the same time.
- One mode of sharing allows users to share the pointer of current location into the file. Thus, the advancing of the pointer by one user affects all sharing users. Here, a file has a single image that interleaves all accesses, regardless of their origin.

In the UNIX semantics a file is associated with a single physical image that is accessed as an exclusive resource. Contention for this single image results in user processes being delayed.

11.5.5 Session Semantics

The Andrew file system (AFS) (Chapter 16) uses the following consistency semantics:

- Writes to an open file by a user are not visible immediately to other users that have the same file open simultaneously.
- Once a file is closed, the changes made to it are visible only in sessions starting later. Already open instances of the file do not reflect these changes.

According to these semantics, a file may be associated temporarily with several (possibly different) images at the same time. Consequently, multiple users are allowed to perform both read and write accesses concurrently on their image of the file, without delay. Almost no constraints are enforced on scheduling accesses.

11.5.6 Immutable-Shared-Files Semantics

A unique approach is that of **immutable shared files**. Once a file is declared as *shared* by its creator, it cannot be modified. An immutable file has two

key properties: Its name may not be reused and its contents may not be altered. Thus, the name of an immutable file signifies that the contents of the file are fixed, rather than the file being a container for variable information. The implementation of these semantics in a distributed system (Chapter 16) is simple, because the sharing is disciplined (read-only).

11.6 ■ Protection

When information is kept in a computer system, we want to keep it safe from physical damage (*reliability*) and improper access (*protection*).

Reliability is generally provided by duplicate copies of files. Many computers have systems programs that automatically (or through computer-operator intervention) copy disk files to tape at regular intervals (once per day or week or month) to maintain a copy should a file system be accidentally destroyed. File systems can be damaged by hardware problems (such as errors in reading or writing), power surges or failures, head crashes, dirt, temperature extremes, and vandalism. Files may be deleted accidentally. Bugs in the file-system software can also cause file contents to be lost. Reliability is covered in more detail in Chapter 14.

Protection can be provided in many ways. For a small single-user system, we might provide protection by physically removing the floppy disks and locking them in a desk drawer or file cabinet. In a multiuser system, however, other mechanisms are needed.

11.6.1 Types of Access

The need to protect files is a direct result of the ability to access files. Systems that do not permit access to the files of other users do not need protection. Thus, we could provide complete protection by prohibiting access. Alternatively, we could provide free access with no protection. Both approaches are too extreme for general use. What is needed is **controlled access**.

Protection mechanisms provide controlled access by limiting the types of file access that can be made. Access is permitted or denied depending on several factors, one of which is the type of access requested. Several different types of operations may be controlled:

- **Read:** Read from the file.
- **Write:** Write or rewrite the file.
- **Execute:** Load the file into memory and execute it.
- **Append:** Write new information at the end of the file.
- **Delete:** Delete the file and free its space for possible reuse.
- **List:** List the name and attributes of the file.

Other operations, such as renaming, copying, or editing the file, may also be controlled. For many systems, however, these higher-level functions may be implemented by a system program that makes lower-level system calls. Protection is provided at only the lower level. For instance, copying a file may be implemented simply by a sequence of read requests. In this case, a user with read access can also cause the file to be copied, printed, and so on.

Many protection mechanisms have been proposed. Each scheme has advantages and disadvantages and must be appropriate for its intended application. A small computer system that is used by only a few members of a research group may not need the same types of protection as will a large corporate computer that is used for research, finance, and personnel operations. A complete treatment of the protection problem is deferred to Chapter 18.

11.6.2 Access Control

The most common approach to the protection problem is to make access dependent on the identity of the user. Various users may need different types of access to a file or directory. The most general scheme to implement identity-dependent access is to associate with each file and directory an **access-control list (ACL)** specifying the user name and the types of access allowed for each user. When a user requests access to a particular file, the operating system checks the access list associated with that file. If that user is listed for the requested access, the access is allowed. Otherwise, a protection violation occurs, and the user job is denied access to the file.

This approach has the advantage of enabling complex access methodologies. The main problem with access lists is their length. If we want to allow everyone to read a file, we must list all users with read access. This technique has two undesirable consequences:

- Constructing such a list may be a tedious and unrewarding task, especially if we do not know in advance the list of users in the system.
- The directory entry, previously of fixed size, now needs to be of variable size, resulting in more complicated space management.

These problems can be resolved by use of a condensed version of the access list.

To condense the length of the access control list, many systems recognize three classifications of users in connection with each file:

- **Owner:** The user who created the file is the owner.
- **Group:** A set of users who are sharing the file and need similar access is a group, or work group.
- **Universe:** All other users in the system constitute the universe.

The most common recent approach is to combine access control lists with the more general (and easier to implement) owner, group, and universe access-control scheme that was described above. For example, Solaris 2.6 and beyond uses the three categories of access by default, but allows access control lists to be added to specific files and directories when more fine-grained access control is desired.

As an example, consider a person, Sara, who is writing a new book. She has hired three graduate students (Jim, Dawn, and Jill) to help with the project. The text of the book is kept in a file named *book*. The protection associated with this file is as follows:

- Sara should be able to invoke all operations on the file.
- Jim, Dawn, and Jill should be able only to read and write the file; they should not be allowed to delete the file.
- All other users should be able to read, but not write, the file. (Sara is interested in letting as many people as possible read the text so that she can obtain appropriate feedback.)

To achieve such a protection, we must create a new group, say *text*, with members Jim, Dawn, and Jill. The name of the group *text* must be then associated with the file *book*, and the access right must be set in accordance with the policy we have outlined.

Now consider a visitor to whom Sara would like to grant temporary access to chapter 1. The visitor cannot be added to the *text* group because that grants to the visitor access to all chapters. Because files can only be in one group, another group cannot be added to chapter 1. With the addition of access-control-list functionality, the visitor can be added to the access control list of chapter 1.

For this scheme to work properly, permissions and access lists must be controlled tightly. This control can be accomplished in several ways. For example, in the UNIX system, groups can be created and modified by only the manager of the facility (or by any superuser). Thus, this control is achieved through human interaction. In the VMS system, the owner of the file can create and modify this list. Access lists are discussed further in Section 18.4.2.

With the more limited protection classification, only three fields are needed to define protection. Each field is often a collection of bits, each of which either allows or prevents the access associated with it. For example, the UNIX system defines three fields of 3 bits each—*rwx*, where *r* controls read access, *w* controls write access, and *x* controls execution. A separate field is kept for the file owner, for the file's group, and for all other users. In this scheme, 9 bits per file are needed to record protection information. Thus, for our example, the protection fields for the file *book* are as follows: For the owner Sara, all 3 bits are set; for the group *text*, the *r* and *w* bits are set; and for the universe, only the *r* bit is set.

One difficulty in combining approaches comes in the user interface. Users must be able to tell when the optional ACL permissions are set on a file. In the Solaris example, a “+” appends the regular permissions, as in:

```
19 -rw-r--r--+ 1 jim staff 130 May 25 22:13 file1
```

A separate set of commands `setfacl` and `getfacl` are used to manage the ACLs. Another difficulty is assigning precedence when permission and ACLs conflict. For example, if Joe is in a file’s group, which has read permission, but the file has an ACL granting Joe read and write permissions, should a write by Joe be granted or denied? Solaris gives ACLs permission (as they are more fine-grained and are not assigned by default). This follows the general rule that specificity should have priority.

11.6.3 Other Protection Approaches

Another approach to the protection problem is to associate a password with each file. Just as access to the computer system is often controlled by a password, access to each file can be controlled by a password. If the passwords are chosen randomly and changed often, this scheme may be effective in limiting access to a file to only those users who know the password. This scheme, however, has several disadvantages. First, the number of passwords that a user needs to remember may become large, making the scheme impractical. Secondly, if only one password is used for all the files, then, once it is discovered, all files are accessible. Some systems (for example, TOPS-20) allow a user to associate a password with a subdirectory, rather than with an individual file, to deal with this problem. The IBM VM/CMS operating system allows three passwords for a minidisk—one each for read, write, and multiwrite access. Thirdly, commonly, only one password is associated with all of the user’s files. Thus, protection is on an all-or-nothing basis. To provide protection on a more detailed level, we must use multiple passwords.

Limited file protection is also currently available on single user systems, such as MS-DOS and Macintosh operating system. These operating systems, when originally designed, essentially ignored the protection problem. However, since these systems are now being placed on networks where file sharing and communication are necessary, protection mechanisms must be **retrofitted** into the operating system. Designing a feature into a new operating system is almost always easier than adding a feature to an existing one. Such updates are usually less effective and are not seamless.

In a multilevel directory structure, we need to protect not only individual files, but also collections of files in a subdirectory; that is, we need to provide a mechanism for directory protection. The directory operations that must be protected are somewhat different from the file operations. We want to control the creation and deletion of files in a directory. In addition, we probably want

to control whether a user can determine the existence of a file in a directory. Sometimes, knowledge of the existence and name of a file may be significant in itself. Thus, listing the contents of a directory must be a protected operation. Therefore, if a path name refers to a file in a directory, the user must be allowed access to both the directory and the file. In systems where files may have numerous path names (such as acyclic or general graphs), a given user may have different access rights to a file, depending on the path name used.

11.6.4 An Example: UNIX

In the UNIX system, directory protection is handled similarly to file protection. That is, associated with each subdirectory are three fields—owner, group, and universe—each consisting of the 3 bits `rwx`. Thus, a user can list the content of a subdirectory only if the `r` bit is set in the appropriate field. Similarly, a user can change his current directory to another current directory (say `foo`) only if the `x` bit associated with the `foo` subdirectory is set in the appropriate field.

A sample directory listing from a UNIX environment is shown in Figure 11.13. The first field describes the file or directory's protection. A `d` as the first character indicates a subdirectory. Also shown are the number of links to the file, the owner's name, the group's name, the size of the file in unit of bytes, the creation date, and finally the file's name (with optional extension).

11.7 ■ Summary

A file is an abstract data type defined and implemented by the operating system. It is a sequence of logical records. A logical record may be a byte, a line (fixed or variable length), or a more complex data item. The operating system may specifically support various record types or may leave that support to the application program.

The major task for the operating system is to map the logical file concept onto physical storage devices such as magnetic tape or disk. Since the physical record size of the device may not be the same as the logical record size, it may

<code>-rw-rw-r-</code>	1 pbg	staff	31200	Sep 3 08:30	intro.ps
<code>drwx---</code>	5 pbg	staff	512	Jul 8 09:33	private/
<code>drwxrwxr-x</code>	2 pbg	staff	512	Jul 8 09:35	doc/
<code>drwxrwx--</code>	2 pbg	student	512	Aug 3 14:13	student-proj/
<code>-rw-r-r-</code>	1 pbg	staff	9423	Feb 24 1999	program.c
<code>-rwxr-xr-x</code>	1 pbg	staff	20471	Feb 24 200	program
<code>drwx-x-x</code>	4 pbg	faculty	512	Jul 31 10:31	lib/
<code>drwx---</code>	3 pbg	staff	1024	Aug 29 06:52	mail/
<code>drwxrwxrwx</code>	3 pbg	staff	512	Jul 8 09:35	test/

Figure 11.13 A sample directory listing.

be necessary to block logical records into physical records. Again, this task may be supported by the operating system or left for the application program.

Each device in a file system keeps a volume table of contents or device directory listing the location of the files on the device. In addition, it is useful to create directories to allow files to be organized. A single-level directory in a multiuser system causes naming problems, since each file must have a unique name. A two-level directory solves this problem by creating a separate directory for each user. Each user has her own directory, containing her own files. The directory lists the files by name, and includes such information as the file's location on the disk, length, type, owner, time of creation, time of last use, and so on.

The natural generalization of a two-level directory is a tree-structured directory. A tree-structured directory allows a user to create subdirectories to organize his files. Acyclic-graph directory structures allow subdirectories and files to be shared, but complicate searching and deletion. A general graph structure allows complete flexibility in the sharing of files and directories, but sometimes requires garbage collection to recover unused disk space.

Disk are segmented into one or more partitions, each containing a file system or left “raw”. File systems may be mounted into the system’s naming structures to make them available. The naming scheme varies by operating system. Once mounted, the files within the partition are available for use. File systems may be unmounted to disable access or for maintenance.

File sharing depends on the semantics provided by the system. Files may have multiple readers, multiple writers, or limits on the sharing. Distributed file systems allow client hosts to mount partitions or directories from servers, as long as they can access each other across a network. Remote file systems have challenges in reliability, performance, and security. Distributed information systems maintain user, host, and access information such that clients and servers share state information to manage use and access.

Since files are the main information-storage mechanism in most computer systems, file protection is needed. Access to files can be controlled separately for each type of access—read, write, execute, append, delete, list directory, and so on. File protection can be provided by passwords, by access lists, or by special ad hoc techniques.

■ Exercises

- 11.1 Consider a file system where a file can be deleted and its disk space reclaimed while links to that file still exist. What problems may occur if a new file is created in the same storage area or with the same absolute path name? How can these problems be avoided?
- 11.2 Some systems automatically delete all user files when a user logs off or a job terminates, unless the user explicitly requests that they be kept; other

systems keep all files unless the user explicitly deletes them. Discuss the relative merits of each approach.

- 11.3 Why do some systems keep track of the type of a file, while others leave it to the user or simply do not implement multiple file types? Which system is “better”?
- 11.4 Similarly, some systems support many types of structures for a file’s data, while others simply support a stream of bytes. What are the advantages and disadvantages?
- 11.5 What are the advantages and disadvantages of recording the name of the creating program with the file’s attributes (as is done in the Macintosh operating system)?
- 11.6 Could you simulate a multilevel directory structure with a single-level directory structure in which arbitrarily long names can be used? If your answer is yes, explain how you can do so, and contrast this scheme with the multilevel directory scheme. If your answer is no, explain what prevents your simulation’s success. How would your answer change if file names were limited to seven characters?
- 11.7 Explain the purpose of the open and close operations.
- 11.8 Some systems automatically open a file when it is referenced for the first time, and close the file when the job terminates. Discuss the advantages and disadvantages of this scheme as compared to the more traditional one, where the user has to open and close the file explicitly.
- 11.9 Give an example of an application in which data in a file should be accessed in the following order:
 - a. Sequentially
 - b. Randomly
- 11.10 Some systems provide file sharing by maintaining a single copy of a file; other systems maintain several copies, one for each of the users sharing the file. Discuss the relative merits of each approach.
- 11.11 In some systems, a subdirectory can be read and written by an authorized user, just as ordinary files can be.
 - a. Describe the protection problems that could arise.
 - b. Suggest a scheme for dealing with each of the protection problems you named in part a.
- 11.12 Consider a system that supports 5000 users. Suppose that you want to allow 4990 of these users to be able to access one file.

- a. How would you specify this protection scheme in UNIX?
 - b. Could you suggest another protection scheme that can be used more effectively for this purpose than the scheme provided by UNIX?
- 11.13 Researchers have suggested that, instead of having an access list associated with each file (specifying which users can access the file, and how), we should have a **user control list** associated with each user (specifying which files a user can access, and how). Discuss the relative merits of these two schemes.

Bibliographical Notes

General discussions concerning file systems were offered by Grosshans [1986]. Golden and Pechura [1986] described the structure of microcomputer file systems. Database systems and their file structures were described in full in Silberschatz et al. [2001].

A multilevel directory structure was first implemented on the MULTICS system (Organick [1972]). Most operating systems now implement multilevel directory structures. These include UNIX (Ritchie and Thompson [1974]), the Apple Macintosh operating system (Apple [1991]), and MS-DOS (Microsoft [1991]).

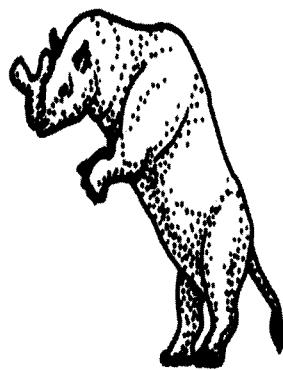
The MS-DOS file system was described in Norton and Wilton [1988]. That of VAX VMS was covered in Kenah et al. [1988] and DEC [1981]. The Network File System (NFS), designed by Sun Microsystems, allows directory structures to be spread across networked computer systems. Discussions concerning NFS were presented in Sandberg et al. [1985], Sandberg [1987], and Sun [1990]. NFS is fully described in Chapter 16. The immutable-shared-files semantics was described by Schroeder et al. [1985].

DNS was first proposed by Su [1982] and has gone through several revision since, with Mockapetris [1987] adding several major features. Most recently, Eastlake [1999] proposing security extensions to let DNS hold security keys.

LDAP, also known as X.509, is a derivative subset of the X.500 distributed directory protocol. It was defined by Yeong et al. [1995] and has been implemented on many operating systems. Interesting research is ongoing in the area of file-system interfaces, in particular, on issues relating to file naming and attributes. For example, the Plan 9 operating system from Bell Laboratories (Lucent Technology) makes all objects look like file systems. Thus, to display a list of processes on a system, a user simply lists the contents of the */proc* directory. Similarly, to display the time of day, a user needs only to type the file */dev/time*.

Chapter 12

FILE-SYSTEM IMPLEMENTATION



As we saw in Chapter 11, the file system provides the mechanism for on-line storage and access to file contents, including data and programs. The file system resides permanently on *secondary storage*, which is designed to hold a large amount of data permanently. This chapter is primarily concerned with issues surrounding file storage and access on the most common secondary-storage medium, the disk. We explore ways to structure file use, to allocate disk space, to recover freed space, to track the locations of data, and to interface other parts of the operating system to secondary storage. Performance issues are considered throughout the chapter.

12.1 ■ File-System Structure

Disks provide the bulk of secondary storage on which a file system is maintained. They have two characteristics that make them a convenient medium for storing multiple files:

1. They can be rewritten in place; it is possible to read a block from the disk, to modify the block, and to write it back into the same place.
2. They can access directly any given block of information on the disk. Thus, it is simple to access any file either sequentially or randomly, and switching from one file to another requires only moving the read-write heads and waiting for the disk to rotate.

We discuss disk structure in great detail in Chapter 14.

Rather than transferring a byte at a time, to improve I/O efficiency, I/O transfers between memory and disk are performed in units of *blocks*. Each block is one or more sectors. Depending on the disk drive, sectors vary from 32 bytes to 4,096 bytes; usually, they are 512 bytes.

To provide an efficient and convenient access to the disk, the operating system imposes one or more **file systems** to allow the data to be stored, located, and retrieved easily. A file system poses two quite different design problems. The first problem is defining how the file system should look to the user. This task involves defining a file and its attributes, the operations allowed on a file, and the directory structure for organizing files. The second problem is creating algorithms and data structures to map the logical file system onto the physical secondary-storage devices.

The file system itself is generally composed of many different levels. The structure shown in Figure 12.1 is an example of a layered design. Each level in the design uses the features of lower levels to create new features for use by higher levels.

The lowest level, the *I/O control*, consists of **device drivers** and interrupt handlers to transfer information between the main memory and the disk system. A device driver can be thought of as a translator. Its input consists of high-level commands such as “retrieve block 123.” Its output consists of low-level, hardware-specific instructions that are used by the hardware controller, which interfaces the I/O device to the rest of the system. The device driver usually writes specific bit patterns to special locations in the I/O controller’s memory to tell the controller on which device location to act and what actions

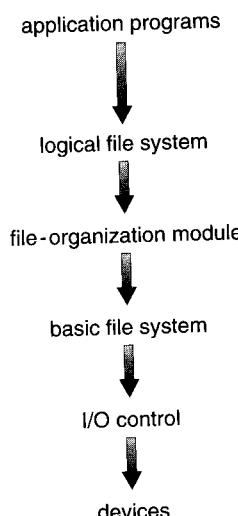


Figure 12.1 Layered file system.

to take. The details of device drivers and the I/O infrastructure are covered in Chapter 13.

The **basic file system** needs only to issue generic commands to the appropriate device driver to read and write physical blocks on the disk. Each physical block is identified by its numeric disk address (for example, drive 1, cylinder 73, track 2, sector 10).

The **file-organization module** knows about files and their logical blocks, as well as physical blocks. By knowing the type of file allocation used and the location of the file, the file-organization module can translate logical block addresses to physical block addresses for the basic file system to transfer. Each file's logical blocks are numbered from 0 (or 1) through N , whereas the physical blocks containing the data usually do not match the logical numbers, so a translation is needed to locate each block. The file-organization module also includes the free-space manager, which tracks unallocated blocks and provides these blocks to the file-organization module when requested.

Finally, the **logical file system** manages metadata information. Metadata includes all of the file-system structure, excluding the actual *data* (or contents of the files). The logical file system manages the directory structure to provide the file-organization module with the information the latter needs, given a symbolic file name. It maintains file structure via file control blocks. A **file control block (FCB)** contains information about the file, including ownership, permissions, and location of the file contents. The logical file system is also responsible for protection and security, as was discussed in Chapter 11 and will be further discussed in Chapter 18.

Many implemented file systems currently exist. Most operating systems support more than one file system. For example, most CD-ROMs are written in the *High Sierra* format, which is a standard format agreed upon by CD-ROM manufacturers. Without such a standard, there would be little or no interoperability between systems trying to use CD-ROMs. Aside from removable media file systems, each operating system has one (or more) disk-based file system. UNIX uses the **UNIX file system (UFS)** as a base. Windows NT supports disk file-system formats of FAT, FAT32 and NTFS (or Windows NT File System), as well as CD-ROM, DVD, and floppy-disk file-system formats. By using a layered structure for file-system implementation, duplication of code is minimized. The I/O control and sometimes the basic file system code can be used by multiple file systems. Each file system may then have its own logical file system and file-organization modules.

12.2 ■ File-System Implementation

As was described in Section 11.1.2, operating systems implement `open` and `close` systems calls for processes to request access to file contents. In this

section, we delve into the structures and operations used to implement file-system operations.

12.2.1 Overview

Several on-disk and in-memory structures are used to implement a file system. These vary depending on the operating system and the file system, but some general principles apply. On-disk, the file system may contain information about how to boot an operating system stored there, the total number of blocks, the number and location of free blocks, the directory structure, and individual files. Many of these structures are detailed throughout the remainder of this chapter.

The on-disk structures include:

- A **boot control block** can contain information needed by the system to boot an operating from that partition. If the disk does not contain an operating system, this block can be empty. It is typically the first block of a partition. In UFS, this is called the **boot block**; in NTFS, it is the **partition boot sector**.
- A **partition control block** contains partition details, such as the number of blocks in the partition, size of the blocks, free-block count and free-block pointers, and free FCB count and FCB pointers. In UFS this is called a **superblock**; in NTFS, it is the **Master File Table**.
- A directory structure is used to organize the files
- An FCB contains many of the file's details, including file permissions, ownership, size, and location of the data blocks. In UFS this is called the **inode**. In NTFS, this information is actually stored within the Master File Table, which uses a relational database structure, with a row per file.

The in-memory information is used for both file-system management and performance improvement via caching. The structures can include:

- An in-memory partition table containing information about each mounted partition.
- An in-memory directory structure that holds the directory information of recently accessed directories. (For directories at which partitions are mounted, it can contain a pointer to the partition table.)
- The **system-wide open-file table** contains a copy of the FCB of each open file, as well as other information.
- The **per-process open-file table** contains a pointer to the appropriate entry in the system-wide open-file table, as well as other information.

To create a new file, an application program calls the logical file system. The logical file system knows the format of the directory structures. To create a new file, it allocates a new FCB, reads the appropriate directory into memory, updates it with the new file name and FCB, and writes it back to the disk. A typical FCB is shown in Figure 12.2.

Some operating systems, including UNIX, treat a directory exactly as a file—one with a type field indicating that it is a directory. Other operating systems, including Windows NT, implement separate system calls for files and directories and treat directories as entities separate from files. No matter the larger structural issues, the logical file system can call the file-organization module to map the directory I/O into disk-block numbers, which are passed on to the basic file system and I/O control system. The file-organization module also allocates blocks for storage of the file's data.

Now that a file has been created, it can be used for I/O. First, though, it must be *opened*. The open call passes a file name to the file system. When a file is opened, the directory structure is searched for the given file name. Parts of the directory structure are usually cached in memory to speed directory operations. Once the file is found, the FCB is copied into a system-wide open-file table in memory. This table not only stores the FCB, but also has entries for a count of the number of processes that have the file open.

Next, an entry is made in the per-process open-file table, with a pointer to the entry in the system-wide open-file table and some other fields. These other fields can include a pointer to the current location in the file (for the next `read` or `write` operation) and the access mode in which the file is open. The open call returns a pointer to the appropriate entry in the per-process file-system table. All file operations are then performed via this pointer. The file name may not be part of the open-file table, as the system has no use for it once the appropriate FCB is located on disk. The name given to the index varies.

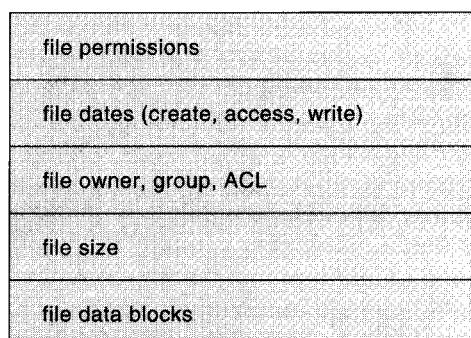


Figure 12.2 A typical file control block.

UNIX systems refer to it as a **file descriptor**; Windows 2000 refers to it as a **file handle**. Consequently, as long as the file is not closed, all file operations are done on the open-file table.

When a process closes the file, the per-process table entry is removed, and the system-wide entry's open count is decremented. When all users that have opened the file close it, the updated file information is copied back to the disk-based directory structure and the system-wide open-file table entry is removed.

In reality, the open system call first searches the system-wide open-file table to see if the file is already in use by another process. If it is, a per-process open-file table entry is created pointing to the existing system-wide open-file table. This algorithm can save substantial overhead when opening files that are already open.

Some systems complicate this scheme even further by using the file system as an interface to other system aspects, such as networking. For example, in UFS, the system-wide open-file table holds the inodes and other information for files and directories. It also holds similar information for network connections and devices. In this way, one mechanism can be used for multiple system aspects.

The caching aspects of these structures should not be overlooked. Using this scheme, all information about an open file, except for its actual data blocks, is in memory. The BSD UNIX system is typical in its use of caches wherever disk I/O can be saved. Its average cache hit rate of 85 percent shows that these techniques are well worth implementing. The BSD UNIX system is described fully in Appendix A

The operating structures of a file-system implementation are summarized in Figure 12.3.

12.2.2 Partitions and Mounting

The layout of a disk can have many variations, depending on the operating system. A disk can be sliced into multiple partitions, or a partition can span multiple disks. The former is discussed here, while the latter is more appropriately considered a form of RAID and is covered in Section 14.5.

Each partition can either be “raw,” containing no file system, or “cooked,” containing a file system. **Raw disk** is used where no file system is appropriate. UNIX swap space can use a raw partition, as it uses its own format on disk and does not use a file system. Likewise, some databases use raw disk and format the data to suit their needs. Raw disk can also hold information needed by disk RAID systems, such as bit maps indicating which blocks are mirrored and which have changed and need to be mirrored. Similarly, raw disk can contain a miniature database holding RAID configuration information, such as which disks are members of each RAID set. Raw disk use is further discussed in Section 14.3.1.

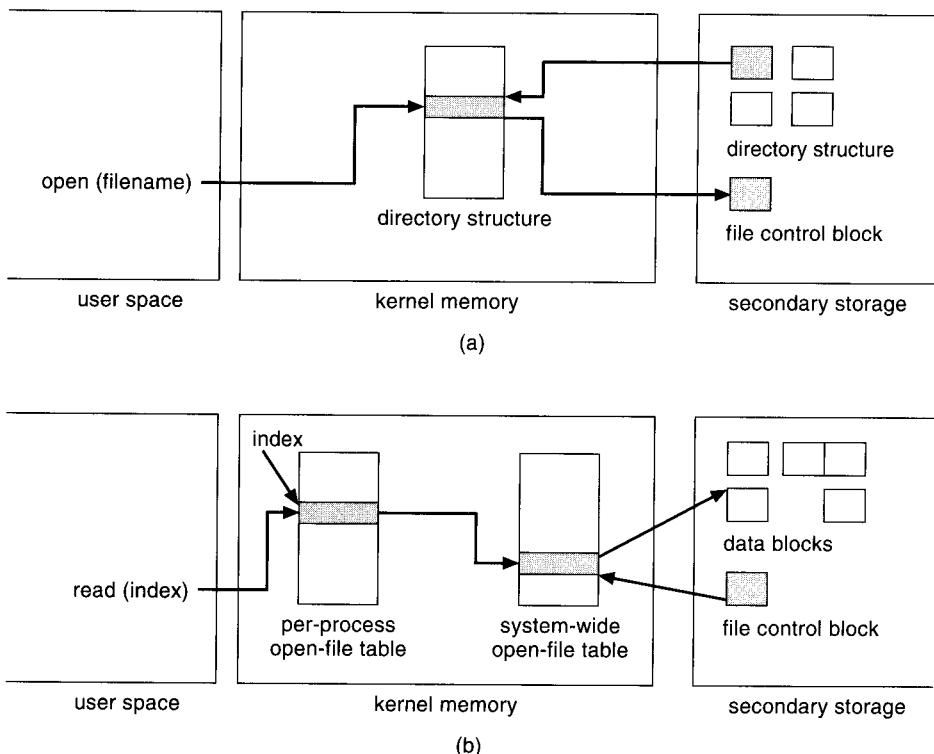


Figure 12.3 In-memory file-system structures. (a) File open. (b) File read.

Boot information can be stored in a separate partition. Again, it has its own format, because at boot time the system does not have file-system device drivers loaded and therefore cannot interpret the file-system format. Rather, it is usually a sequential series of blocks, loaded as an image into memory. Execution of the image starts at a predefined location, such as the first byte. This boot image can contain more than the instructions for how to boot a specific operating system. For instance, PCs and other systems can be **dual-booted**. Multiple operating systems can be installed on such a system. How does the system know which one to boot? A boot loader that understands multiple file systems and multiple operating systems can occupy the boot space. Once loaded, it can boot one of the operating systems available on the disk. The disk can have multiple partitions, each containing a different type of file system and a different operating system.

The **root partition**, which contains the operating-system kernel and potentially other system files, is mounted at boot time. Other partitions can be automatically mounted at boot or manually mounted later, depending on the operating system. As part of a successful mount operation, the operating sys-

tem verifies that the device contains a valid file system. It does so by asking the device driver to read the device directory and verifying that the directory has the expected format. If the format is invalid, the partition must have its consistency checked and possibly corrected, either with or without user intervention. Finally, the operating system notes in its in-memory **mount table** structure that a file system is mounted, and the type of the file system. The details of this function depend on the operating system. Microsoft Windows-based systems mount each partition in a separate name space, denoted by a letter and a colon. To record that a file system is mounted at *f:*, for example, the operating system places a pointer to the file system in a field of the device structure corresponding to *f:*. When a process specifies the driver letter, the operating system finds the appropriate file-system pointer and traverses the directory structures on that device to find the specified file or directory.

On UNIX, file systems can be mounted at any directory. This is implemented by setting a flag in the in-memory copy of the inode for that directory. The flag indicates that the directory is a mount point. A field then points to an entry in the mount table, indicating which device is mounted there. The mount table entry contains a pointer to the superblock of the file system on that device. This scheme enables the operating system to traverse its directory structure, switching among file systems as appropriate.

12.2.3 Virtual File Systems

While the previous section makes it clear that modern operating systems must support concurrently multiple types of file systems, we now need to discuss some implementation details. How does an operating system allow multiple types of file systems to be integrated into a directory structure? How can users seamlessly move between file-system types as they navigate the file-system space?

An obvious but suboptimal method of implementing multiple types of file systems is to write directory and file routines for each type. Rather, most operating systems, including UNIX, use object-oriented techniques to simplify, organize, and modularize the implementation. The use of these methods allows very dissimilar file-system types to be implemented within the same structure, including network file systems, such as NFS. Users can access files that are contained within multiple file systems on the local disk, or even on file systems available across the network.

Data structures and procedures are used to isolate the basic system call functionality from the implementation details. Thus, the file-system implementation consists of three major layers; it is depicted schematically in Figure 12.4. The first layer is the file-system interface, based on the `open`, `read`, `write`, and `close` calls, and file descriptors.

The second layer is called the **Virtual File System (VFS)** layer; it serves two important functions:

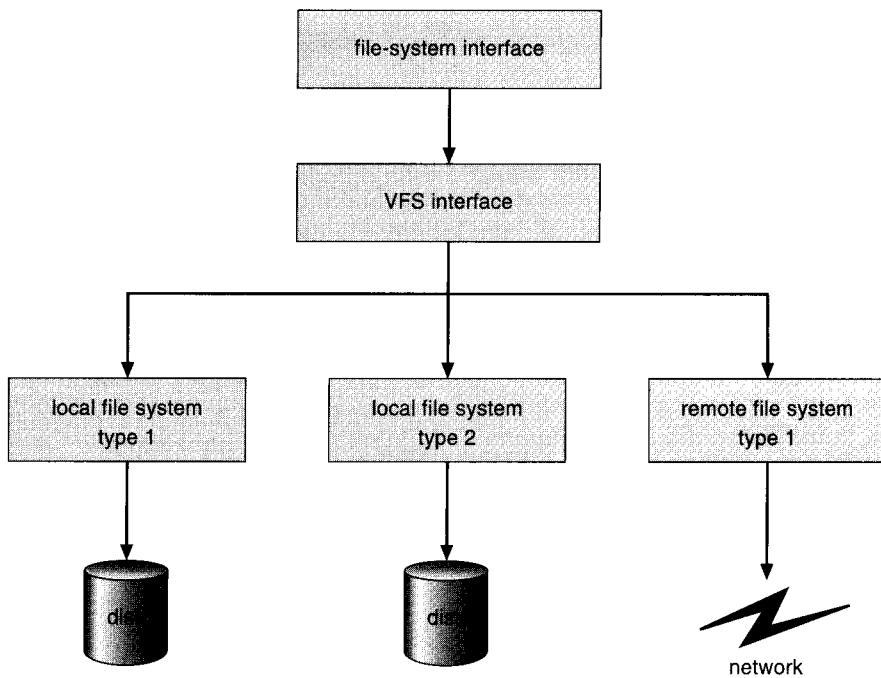


Figure 12.4 Schematic view of a virtual file system.

1. It separates file-system-generic operations from their implementation by defining a clean VFS interface. Several implementations for the VFS interface may coexist on the same machine, allowing transparent access to different types of file systems mounted locally.
2. The VFS is based on a file-representation structure, called a **vnode**, that contains a numerical designator for a network-wide unique file. (UNIX inodes are unique within only a single file system.) This network-wide uniqueness is required for support of network file systems. The kernel maintains one vnode structure for each active node (file or directory).

Thus, the VFS distinguishes local files from remote ones, and local files are further distinguished according to their file-system types.

The VFS activates file-system-specific operations to handle local requests according to their file-system types, and even calls the NFS protocol procedures for remote requests. File handles are constructed from the relevant vnodes and are passed as arguments to these procedures. The layer implementing the file-system type, or remote file system protocol, is the bottom layer of the architecture. An illustration of VFS operation is found in Section 12.9.

12.3 ■ Directory Implementation

The selection of directory-allocation and directory-management algorithms has a large effect on the efficiency, performance, and reliability of the file system. Therefore, you need to understand the tradeoffs involved in these algorithms.

12.3.1 Linear List

The simplest method of implementing a directory is to use a linear list of file names with pointers to the data blocks. A linear list of directory entries requires a linear search to find a particular entry. This method is simple to program but time-consuming to execute. To create a new file, we must first search the directory to be sure that no existing file has the same name. Then, we add a new entry at the end of the directory. To delete a file, we search the directory for the named file, then release the space allocated to it. To reuse the directory entry, we can do one of several things. We can mark the entry as unused (by assigning it a special name, such as an all-blank name, or with a used–unused bit in each entry), or we can attach it to a list of free directory entries. A third alternative is to copy the last entry in the directory into the freed location, and to decrease the length of the directory. A linked list can also be used to decrease the time to delete a file.

The real disadvantage of a linear list of directory entries is the linear search to find a file. Directory information is used frequently, and users would notice a slow implementation of access to it. In fact, many operating systems implement a software cache to store the most recently used directory information. A cache hit avoids constantly rereading the information from disk. A sorted list allows a binary search and decreases the average search time. However, the requirement that the list must be kept sorted may complicate creating and deleting files, since we may have to move substantial amounts of directory information to maintain a sorted directory. A more sophisticated tree data structure, such as a B-tree, might help here. An advantage of the sorted list is that a sorted directory listing can be produced without a separate sort step.

12.3.2 Hash Table

Another data structure that has been used for a file directory is a **hash table**. In this method, a linear list stores the directory entries, but a hash data structure is also used. The hash table takes a value computed from the file name and returns a pointer to the file name in the linear list. Therefore, it can greatly decrease the directory search time. Insertion and deletion are also fairly straightforward, although some provision must be made for **collisions**—situations where two file names hash to the same location. The major difficulties with a hash table are its generally fixed size and the dependence of the hash function on that size.

For example, assume that we make a linear-probing hash table that holds 64 entries. The hash function converts file names into integers from 0 to 63, for

instance, by using the remainder of a division by 64. If we later try to create a 65th file, we must enlarge the directory hash table—say, to 128 entries. As a result, we need a new hash function that must map file names to the range 0 to 127, and we must reorganize the existing directory entries to reflect their new hash-function values. Alternately, a chained-overflow hash table can be used. Each hash entry can be a linked list instead of an individual value, and we can resolve collisions by adding the new entry to the linked list. Lookups may be somewhat slowed, because searching for a name might require stepping through a linked list of colliding table entries, but this is likely to be much faster than a linear search through the entire directory.

12.4 ■ Allocation Methods

The direct-access nature of disks allows us flexibility in the implementation of files. In almost every case, many files will be stored on the same disk. The main problem is how to allocate space to these files so that disk space is utilized effectively and files can be accessed quickly. Three major methods of allocating disk space are in wide use: contiguous, linked, and indexed. Each method has advantages and disadvantages. Some systems (such as Data General's RDOS for its Nova line of computers) support all three. More commonly, a system will use one particular method for all files.

12.4.1 Contiguous Allocation

The **contiguous-allocation** method requires each file to occupy a set of contiguous blocks on the disk. Disk addresses define a linear ordering on the disk. With this ordering, assuming that only one job is accessing the disk, accessing block $b + 1$ after block b normally requires no head movement. When head movement is needed (from the last sector of one cylinder to the first sector of the next cylinder), it is only one track. Thus, the number of disk seeks required for accessing contiguously allocated files is minimal, as is seek time when a seek is finally needed. The IBM VM/CMS operating system uses contiguous allocation because it provides such good performance.

Contiguous allocation of a file is defined by the disk address and length (in block units) of the first block. If the file is n blocks long and starts at location b , then it occupies blocks $b, b + 1, b + 2, \dots, b + n - 1$. The directory entry for each file indicates the address of the starting block and the length of the area allocated for this file (Figure 12.5).

Accessing a file that has been allocated contiguously is easy. For sequential access, the file system remembers the disk address of the last block referenced and, when necessary, reads the next block. For direct access to block i of a file that starts at block b , we can immediately access block $b + i$. Thus, both sequential and direct access can be supported by contiguous allocation.

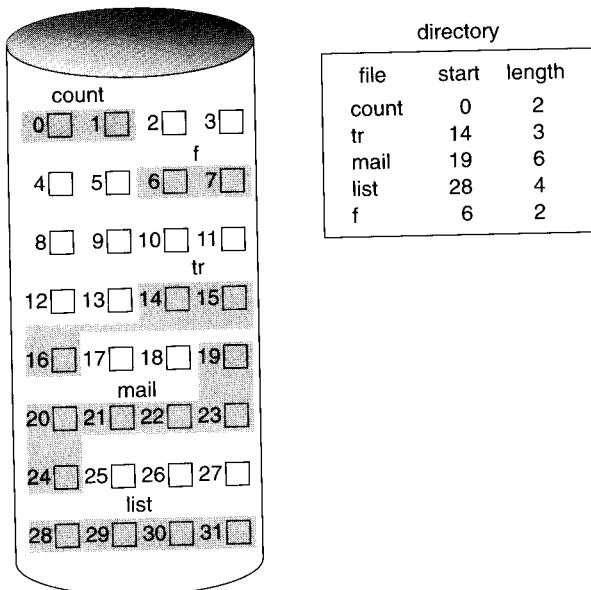


Figure 12.5 Contiguous allocation of disk space.

Contiguous allocation has some problems, however. One difficulty is finding space for a new file. The implementation of the free-space-management system, discussed in Section 12.5, determines how this task is accomplished. Any management system can be used, but some are slower than others.

The contiguous disk-space-allocation problem can be seen to be a particular application of the general **dynamic storage-allocation** problem discussed in Section 9.3, which is how to satisfy a request of size n from a list of free holes. First fit and best fit are the most common strategies used to select a free hole from the set of available holes. Simulations have shown that both first fit and best fit are more efficient than worst fit in terms of both time and storage utilization. Neither first fit nor best fit is clearly best in terms of storage utilization, but first fit is generally faster.

These algorithms suffer from the problem of **external fragmentation**. As files are allocated and deleted, the free disk space is broken into little pieces. External fragmentation exists whenever free space is broken into chunks. It becomes a problem when the largest contiguous chunk is insufficient for a request; storage is fragmented into a number of holes, no one of which is large enough to store the data. Depending on the total amount of disk storage and the average file size, external fragmentation may be a minor or a major problem.

Some older microcomputer systems used contiguous allocation on floppy disks. To prevent loss of significant amounts of disk space to external fragmentation, the user had to run a repacking routine that copied the entire file system

onto another floppy disk or onto a tape. The original floppy disk was then freed completely, creating one large contiguous free space. The routine then copied the files back onto the floppy disk by allocating contiguous space from this one large hole. This scheme effectively **compacts** all free space into one contiguous space, solving the fragmentation problem. The cost of this compaction is time. The time cost is particularly severe for large hard disks that use contiguous allocation, where compacting all the space may take hours and may be necessary on a weekly basis. During this **down time**, normal system operation generally cannot be permitted, so such compaction is avoided at all costs on production machines.

Another problem with contiguous allocation is determining how much space is needed for a file. When the file is created, the total amount of space it will need must be found and allocated. How does the creator (program or person) know the size of the file to be created? In some cases, this determination may be fairly simple (copying an existing file, for example); in general, however, the size of an output file may be difficult to estimate.

If we allocate too little space to a file, we may find that the file cannot be extended. Especially with a best-fit allocation strategy, the space on both sides of the file may be in use. Hence, we cannot make the file larger in place. Two possibilities then exist. First, the user program can be terminated, with an appropriate error message. The user must then allocate more space and run the program again. These repeated runs may be costly. To prevent them, the user will normally overestimate the amount of space needed, resulting in considerable wasted space.

The other possibility is to find a larger hole, copy the contents of the file to the new space, and release the previous space. This series of actions may be repeated as long as space exists, although it can be time-consuming. However, in this case, the user never needs to be informed explicitly about what is happening; the system continues despite the problem, although more and more slowly.

Even if the total amount of space needed for a file is known in advance, preallocation may be inefficient. A file that grows slowly over a long period (months or years) must be allocated enough space for its final size, even though much of that space may be unused for a long time. The file, therefore, has a large amount of internal fragmentation.

To minimize these drawbacks, some operating systems use a modified contiguous-allocation scheme, in which a contiguous chunk of space is allocated initially, and then, when that amount is not large enough, another chunk of contiguous space, an **extent**, is added to the initial allocation. The location of a file's blocks is then recorded as a location and a block count, plus a link to the first block of the next extent. On some systems, the owner of the file can set the extent size, but this setting results in inefficiencies if the owner is incorrect. Internal fragmentation can still be a problem if the extents are too large, and external fragmentation can be a problem as extents of varying sizes

are allocated and deallocated. The commercial Veritas File System uses extents to optimize performance. It is a high-performance replacement for the standard UFS.

12.4.2 Linked Allocation

Linked allocation solves all problems of contiguous allocation. With linked allocation, each file is a linked list of disk blocks; the disk blocks may be scattered anywhere on the disk. The directory contains a pointer to the first and last blocks of the file. For example, a file of five blocks might start at block 9, continue at block 16, then block 1, block 10, and finally block 25 (Figure 12.6). Each block contains a pointer to the next block. These pointers are not made available to the user. Thus, if each block is 512 bytes, and a disk address (the pointer) requires 4 bytes, then the user sees blocks of 508 bytes.

To create a new file, we simply create a new entry in the directory. With linked allocation, each directory entry has a pointer to the first disk block of the file. This pointer is initialized to *nil* (the end-of-list pointer value) to signify an empty file. The size field is also set to 0. A write to the file causes a free block to be found via the free-space-management system, and this new block is then written to, and is linked to the end of the file. To read a file, we simply read blocks by following the pointers from block to block. There is no external fragmentation with linked allocation, and any free block on the free-space list

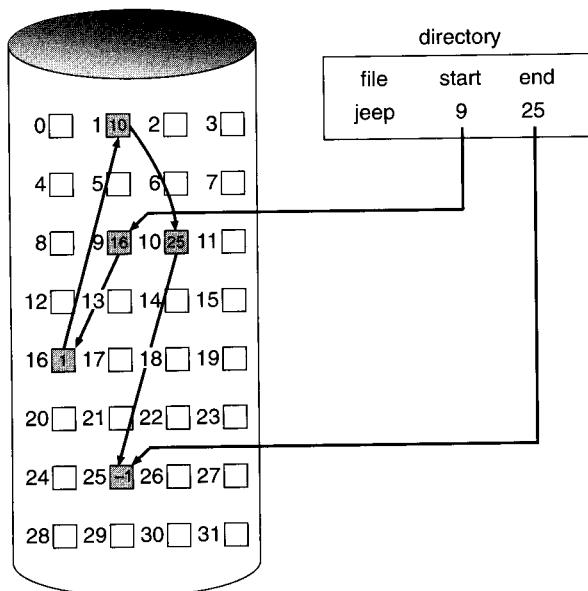


Figure 12.6 Linked allocation of disk space.

can be used to satisfy a request. The size of a file does not need to be declared when that file is created. A file can continue to grow as long as free blocks are available. Consequently, it is never necessary to compact disk space.

Linked allocation does have disadvantages, however. The major problem is that it can be used effectively only for sequential-access files. To find the i th block of a file, we must start at the beginning of that file, and follow the pointers until we get to the i th block. Each access to a pointer requires a disk read, and sometimes a disk seek. Consequently, it is inefficient to support a direct-access capability for linked allocation files.

Another disadvantage to linked allocation is the space required for the pointers. If a pointer requires 4 bytes out of a 512-byte block, then 0.78 percent of the disk is being used for pointers, rather than for information. Each file requires slightly more space than it would otherwise.

The usual solution to this problem is to collect blocks into multiples, called **clusters**, and to allocate the clusters rather than blocks. For instance, the file system may define a cluster as 4 blocks, and operate on the disk in only cluster units. Pointers then use a much smaller percentage of the file's disk space. This method allows the logical-to-physical block mapping to remain simple, but improves disk throughput (fewer disk head seeks) and decreases the space needed for block allocation and free-list management. The cost of this approach is an increase in internal fragmentation, because more space is wasted if a cluster is partially full than when a block is partially full. Clusters can be used to improve the disk-access time for many other algorithms, so they are used in most operating systems.

Yet another problem of linked allocation is reliability. Since the files are linked together by pointers scattered all over the disk, consider what would happen if a pointer were lost or damaged. A bug in the operating-system software or a disk hardware failure might result in picking up the wrong pointer. This error could result in linking into the free-space list or into another file. Partial solutions are to use doubly linked lists or to store the file name and relative block number in each block; however, these schemes require even more overhead for each file.

An important variation on the linked allocation method is the use of a **file-allocation table (FAT)**. This simple but efficient method of disk-space allocation is used by the MS-DOS and OS/2 operating systems. A section of disk at the beginning of each partition is set aside to contain the table. The table has one entry for each disk block, and is indexed by block number. The FAT is used much as is a linked list. The directory entry contains the block number of the first block of the file. The table entry indexed by that block number then contains the block number of the next block in the file. This chain continues until the last block, which has a special end-of-file value as the table entry. Unused blocks are indicated by a 0 table value. Allocating a new block to a file is a simple matter of finding the first 0-valued table entry, and replacing the previous end-of-file value with the address of the new block. The 0 is then

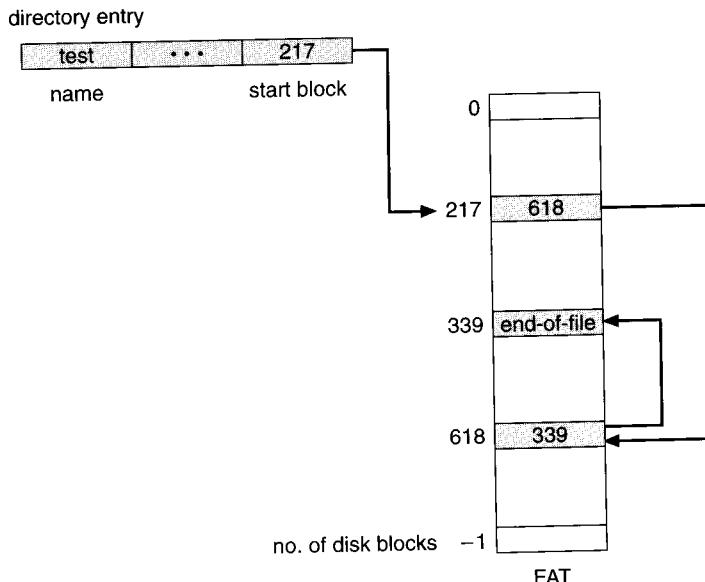


Figure 12.7 File-allocation table.

replaced with the end-of-file value. An illustrative example is the FAT structure of Figure 12.7 for a file consisting of disk blocks 217, 618, and 339.

The FAT allocation scheme can result in a significant number of disk head seeks, unless the FAT is cached. The disk head must move to the start of the partition to read the FAT and find the location of the block in question, then move to the location of the block itself. In the worst case, both moves occur for each of the blocks. A benefit is that random access time is improved, because the disk head can find the location of any block by reading the information in the FAT.

12.4.3 Indexed Allocation

Linked allocation solves the external-fragmentation and size-declaration problems of contiguous allocation. However, in the absence of a FAT, linked allocation cannot support efficient direct access, since the pointers to the blocks are scattered with the blocks themselves all over the disk and need to be retrieved in order. **Indexed allocation** solves this problem by bringing all the pointers together into one location: the **index block**.

Each file has its own index block, which is an array of disk-block addresses. The *i*th entry in the index block points to the *i*th block of the file. The directory contains the address of the index block (Figure 12.8). To read the *i*th block, we

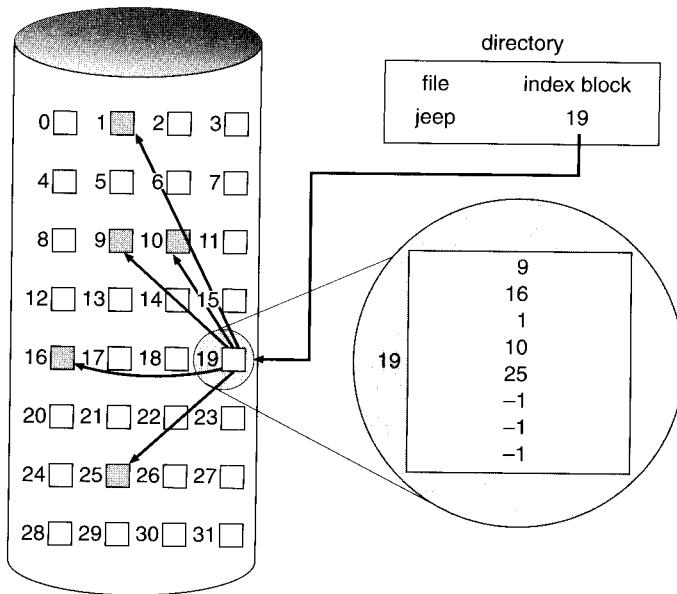


Figure 12.8 Indexed allocation of disk space.

use the pointer in the i th index-block entry to find and read the desired block. This scheme is similar to the paging scheme described in Chapter 9.

When the file is created, all pointers in the index block are set to *nil*. When the i th block is first written, a block is obtained from the free-space manager, and its address is put in the i th index-block entry.

Indexed allocation supports direct access, without suffering from external fragmentation, because any free block on the disk may satisfy a request for more space.

Indexed allocation does suffer from wasted space. The pointer overhead of the index block is generally greater than the pointer overhead of linked allocation. Consider a common case in which we have a file of only one or two blocks. With linked allocation, we lose the space of only one pointer per block (one or two pointers). With indexed allocation, an entire index block must be allocated, even if only one or two pointers will be non-*varnil*.

This point raises the question of how large the index block should be. Every file must have an index block, so we want the index block to be as small as possible. If the index block is too small, however, it will not be able to hold enough pointers for a large file, and a mechanism will have to be available to deal with this issue:

- **Linked scheme:** An index block is normally one disk block. Thus, it can be read and written directly by itself. To allow for large files, we may link together several index blocks. For example, an index block might contain a

small header giving the name of the file, and a set of the first 100 disk-block addresses. The next address (the last word in the index block) is *nil* (for a small file) or is a pointer to another index block (for a large file).

- **Multilevel index:** A variant of the linked representation is to use a first-level index block to point to a set of second-level index blocks, which in turn point to the file blocks. To access a block, the operating system uses the first-level index to find a second-level index block, and that block to find the desired data block. This approach could be continued to a third or fourth level, depending on the desired maximum file size. With 4,096-byte blocks, we could store 1,024 4-byte pointers in an index block. Two levels of indexes allow 1,048,576 data blocks, which allows a file of up to 4 GB.
- **Combined scheme:** Another alternative, used in the UFS, is to keep the first, say, 15 pointers of the index block in the file's inode. The first 12 of these pointers point to **direct blocks**; that is, they contain addresses of blocks that contain data of the file. Thus, the data for small (no more than 12 blocks) files do not need a separate index block. If the block size is 4 KB, then up to 48 KB of data may be accessed directly. The next 3 pointers point to **indirect blocks**. The first indirect block pointer is the address of a **single indirect block**. The single indirect block is an index block, containing not data, but rather the addresses of blocks that do contain data. Then there is a **double indirect block** pointer, which contains the address of a block that contains the addresses of blocks that contain pointers to the actual data blocks. The last pointer would contain the address of a **triple indirect block**. Under this method, the number of blocks that can be allocated to a file exceeds the amount of space addressable by the 4-byte file pointers used by many operating systems. A 32-bit file pointer reaches only 2^{32} bytes, or 4 GB. Many UNIX implementations, including Solaris and IBM's AIX, now support up to 64-bit file pointers. Pointers of this size allow files and file systems to be terabytes in size. An inode is shown in Figure 12.9.

Indexed-allocation schemes suffer from some of the same performance problems as does linked allocation. Specifically, the index blocks can be cached in memory, but the data blocks may be spread all over a partition.

12.4.4 Performance

The allocation methods that we have discussed vary in their storage efficiency and data-block access times. Both are important criteria in selecting the proper method or methods for an operating system to implement.

Before selecting an allocation method, we need to determine how the systems will be used. A system with mostly sequential access should use a method different from that for a system with mostly random access. For any type of access, contiguous allocation requires only one access to get a disk

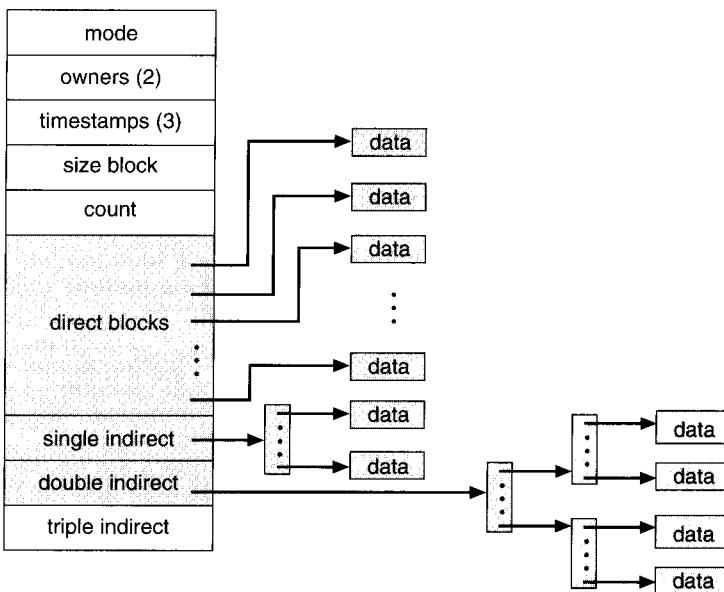


Figure 12.9 The UNIX inode.

block. Since we can easily keep the initial address of the file in memory, we can calculate immediately the disk address of the i th block (or the next block) and read it directly.

For linked allocation, we can also keep the address of the next block in memory and read it directly. This method is fine for sequential access; for direct access, however, an access to the i th block might require i disk reads. This problem indicates why linked allocation should not be used for an application requiring direct access.

As a result, some systems support direct-access files by using contiguous allocation and sequential access by linked allocation. For these systems, the type of access to be made must be declared when the file is created. A file created for sequential access will be linked and cannot be used for direct access. A file created for direct access will be contiguous and can support both direct access and sequential access, but its maximum length must be declared when it is created. In this case, the operating system must have appropriate data structures and algorithms to support *both* allocation methods. Files can be converted from one type to another by the creation of a new file of the desired type, into which the contents of the old file are copied. The old file may then be deleted, and the new file renamed.

Indexed allocation is more complex. If the index block is already in memory, then the access can be made directly. However, keeping the index block in memory requires considerable space. If this memory space is not available, then

we may have to read first the index block and then the desired data block. For a two-level index, two index-block reads might be necessary. For an extremely large file, accessing a block near the end of the file would require reading in all the index blocks to follow the pointer chain before the needed data block finally could be read. Thus, the performance of indexed allocation depends on the index structure, on the size of the file, and on the position of the block desired.

Some systems combine contiguous allocation with indexed allocation by using contiguous allocation for small files (up to three or four blocks), and automatically switching to an indexed allocation if the file grows large. Since most files are small, and contiguous allocation is efficient for small files, average performance can be quite good.

For instance, the version of the UNIX operating system from Sun Microsystems was changed in 1991 to improve performance in the file-system allocation algorithm. The performance measurements indicated that the maximum disk throughput on a typical workstation (12-MIPS SPARCstation1) took 50 percent of the CPU and produced a disk bandwidth of only 1.5 MB per second. To improve performance, Sun made changes to allocate space in clusters of size 56 KB whenever possible. (56 KB was the maximum size of a DMA transfer on Suns at that time.) This allocation reduced external fragmentation, and thus seek and latency times. In addition, the disk-reading routines were optimized to read in these large clusters. The inode structure was left unchanged. These changes, plus the use of read-ahead and free-behind (discussed in Section 12.6.2), resulted in 25 percent less CPU being used for substantially improved throughput.

Many other optimizations are possible and are in use. Given the disparity between CPU and disk speed, it is not unreasonable to add thousands of extra instructions to the operating system to save just a few disk-head movements. Furthermore, this disparity is increasing over time, to the point where hundreds of thousands of instructions reasonably could be used to optimize head movements.

12.5 ■ Free-Space Management

Since disk space is limited, we need to reuse the space from deleted files for new files, if possible. (Write-once optical disks only allow one write to any given sector, and thus such reuse is not physically possible.) To keep track of free disk space, the system maintains a **free-space list**. The free-space list records all *free* disk blocks—those not allocated to some file or directory. To create a file, we search the free-space list for the required amount of space, and allocate that space to the new file. This space is then removed from the free-space list. When a file is deleted, its disk space is added to the free-space list. The free-space list, despite its name, might not be implemented as a list, as we shall discuss.

12.5.1 Bit Vector

Frequently, the free-space list is implemented as a **bit map** or **bit vector**. Each block is represented by 1 bit. If the block is free, the bit is 1; if the block is allocated, the bit is 0.

For example, consider a disk where blocks 2, 3, 4, 5, 8, 9, 10, 11, 12, 13, 17, 18, 25, 26, and 27 are free, and the rest of the blocks are allocated. The free-space bit map would be

00111100111110001100000011100000 ...

The main advantage of this approach is its relatively simplicity and efficiency in finding the first free block, or n consecutive free blocks on the disk. Indeed, many computers supply bit-manipulation instructions that can be used effectively for that purpose. For example, the Intel family starting with the 80386 and the Motorola family starting with the 68020 (processors that have powered PCs and Macintosh systems, respectively) have instructions that return the offset in a word of the first bit with the value 1. In fact, the Apple Macintosh operating system uses the bit-vector method to allocate disk space. To find the first free block, the Macintosh operating system checks sequentially each word in the bit map to see whether that value is not 0, since a 0-valued word has all 0 bits and represents a set of allocated blocks. The first non-0 word is scanned for the first 1 bit, which is the location of the first free block. The calculation of the block number is

$$(\text{number of bits per word}) \times (\text{number of 0-value words}) + \text{offset of first 1 bit}.$$

Again, we see hardware features driving software functionality. Unfortunately, bit vectors are inefficient unless the entire vector is kept in main memory (and is written to disk occasionally for recovery needs). Keeping it in main memory is possible for smaller disks, such as on microcomputers, but not for larger ones. A 1.3-GB disk with 512-byte blocks would need a bit map of over 332 KB to track its free blocks. Clustering the blocks in groups of four reduces this number to over 83 KB per disk.

12.5.2 Linked List

Another approach to free-space management is to link together all the free disk blocks, keeping a pointer to the first free block in a special location on the disk and caching it in memory. This first block contains a pointer to the next free disk block, and so on. In our example (Section 12.5.1), we would keep a pointer to block 2, as the first free block. Block 2 would contain a pointer to block 3, which would point to block 4, which would point to block 5, which would point to block 8, and so on (Figure 12.10). However, this scheme is not efficient; to traverse the list, we must read each block, which requires substantial I/O

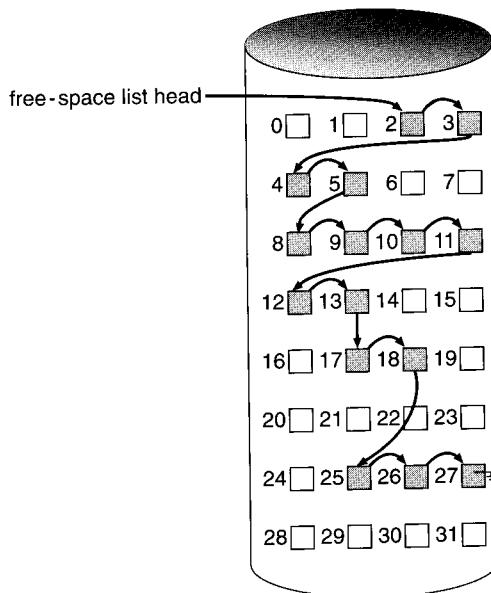


Figure 12.10 Linked free space list on disk.

time. Fortunately, traversing the free list is not a frequent action. Usually, the operating system simply needs a free block so that it can allocate that block to a file, so the first block in the free list is used. The FAT method incorporates free-block accounting into the allocation data structure. No separate method is needed.

12.5.3 Grouping

A modification of the free-list approach is to store the addresses of n free blocks in the first free block. The first $n-1$ of these blocks are actually free. The last block contains the addresses of another n free blocks, and so on. The importance of this implementation is that the addresses of a large number of free blocks can be found quickly, unlike in the standard linked-list approach.

12.5.4 Counting

Another approach is to take advantage of the fact that, generally, several contiguous blocks may be allocated or freed simultaneously, particularly when space is allocated with the contiguous-allocation algorithm or through clustering. Thus, rather than keeping a list of n free disk addresses, we can keep the address of the first free block and the number n of free contiguous blocks that follow the first block. Each entry in the free-space list then consists of a disk address and a count. Although each entry requires more space than would

a simple disk address, the overall list will be shorter, as long as the count is generally greater than 1.

12.6 ■ Efficiency and Performance

Now that we have discussed the block-allocation and directory-management options, we can further consider their effect on performance and efficient disk use. Disks tend to be a major bottleneck in system performance, since they are the slowest main computer component. In this section, we discuss a variety of techniques used to improve the efficiency and performance of secondary storage.

12.6.1 Efficiency

The efficient use of disk space is heavily dependent on the disk allocation and directory algorithms in use. For instance, UNIX inodes are preallocated on a partition. Even an “empty” disk has a percentage of its space lost to inodes. However, by preallocating the inodes and spreading them across the partition, we improve the file system’s performance. This improved performance is a result of the UNIX allocation and free-space algorithms, which try to keep a file’s data blocks near that file’s inode block to reduce seek time.

As another example, let us reconsider the clustering scheme discussed in Section 12.4, which aids in file-seek and file-transfer performance at the cost of internal fragmentation. To reduce this fragmentation, BSD UNIX varies the cluster size as a file grows. Large clusters are used where they can be filled, and small clusters are used for small files and the last cluster of a file. This system is described in Appendix A.

The types of data normally kept in a file’s directory (or inode) entry also require consideration. Commonly, a “last write date” is recorded to supply information to the user and to determine whether the file needs to be backed up. Some systems also keep a “last access date,” so that a user can determine when the file was last read. The result of keeping this information is that, whenever the file is read, a field in the directory structure must be written to. This change requires the block to be read into memory, a section changed, and the block written back out to disk, because operations on disks occur only in block (or cluster) chunks. So, any time a file is opened for reading, its directory entry must be read and written as well. This requirement can be inefficient for frequently accessed files, so we must weigh its benefit against its performance cost when designing a file system. Generally, *every* data item associated with a file needs to be considered for its effect on efficiency and performance.

As an example, consider how efficiency is affected by the size of the pointers used to access data. Most systems use either 16- or 32-bit pointers throughout the operating system. These pointer sizes limit the length of a file to either 2^{16} (64 KB) or 2^{32} bytes (4 GB). Some systems implement 64-bit pointers to increase

this limit to 2^{64} bytes, which is a very large number indeed. However, 64-bit pointers take more space to store, and in turn make the allocation and free-space-management methods (linked lists, indexes, and so on) use more disk space.

One of the difficulties in choosing a pointer size, or indeed any fixed allocation size within an operating system, is planning for the effects of changing technology. Consider that the IBM PC XT had a 10-MB hard drive and an MS-DOS file system that could support only 32 MB. (Each FAT entry was 12 bits, pointing to an 8-KB cluster.) As disk capacities increased, larger disks had to be split into 32-MB partitions, because the file system could not track blocks beyond 32 MB. As hard disks of over 100-MB capacities became common, the disk data structures and algorithms in MS-DOS had to be modified to allow larger file systems. (Each FAT entry was expanded to 16 bits, and later to 32 bits.) The initial file-system decisions were made for efficiency reasons; however, with the advent of MS-DOS Version 4, millions of computer users were inconvenienced when they had to switch to the new, larger file system.

As another example, consider the evolution of Sun's Solaris operating system. Originally, many data structures were of fixed lengths, allocated at system startup. These structures included the process table and the open-file table. When the process table became full, no more processes could be created. When the file table became full, no more files could be opened. The system would fail to provide services to the users. These table sizes could be increased only by recompiling the kernel and rebooting the system. Since the release of Solaris 2, almost all kernel structures are allocated dynamically, eliminating these artificial limits on system performance. Of course, the algorithms that manipulate these tables are more complicated, and the operating system is a little slower because it must dynamically allocate and deallocate table entries, but that price is the usual one for more functional generality.

12.6.2 Performance

Once the basic file-system algorithms are selected, we can still improve performance in several ways. As noted in Chapter 2, most disk controllers include local memory to form an on-board **cache** that is sufficiently large to store entire tracks at a time. Once a seek is performed, the track is read into the disk cache starting at the sector under the disk head (alleviating latency time). The disk controller then transfers any sector requests to the operating system. Once blocks make it from the disk controller into main memory, the operating system may cache the blocks there.

Some systems maintain a separate section of main memory for a **disk cache**, where blocks are kept under the assumption that they will be used again shortly. Other systems cache file data using a **page cache**. The page cache uses virtual-memory techniques to cache file data as pages rather than as file-system-oriented blocks. Caching file data using virtual addresses is far more

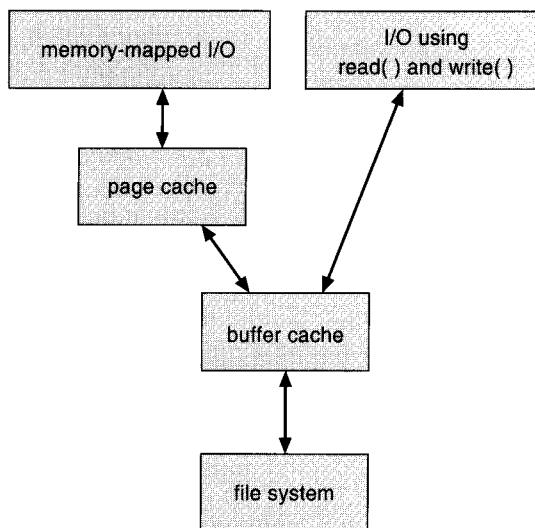


Figure 12.11 I/O without a unified buffer cache.

efficient than caching through physical disk blocks. Several systems, including Solaris, some new Linux releases, and Windows NT and 2000, use page caching to cache both process pages and file data. This is known as **unified virtual memory**. Solaris uses both a block cache and a page cache. The block cache is used for file-system metadata (such as inodes) and the page cache is used for all file-system data.

Some versions of UNIX provide a **unified buffer cache**. Consider the two alternatives of opening and accessing a file. One approach is to use memory mapping (Section 10.3.2), the second is to use the standard system calls `read` and `write`. Without a unified buffer cache, we have a situation similar to Figure 12.11. In this instance, the `read` and `write` system calls go through the buffer cache. The memory mapping call requires using two caches—the page cache and buffer cache. A memory mapping proceeds by reading in disk blocks from the file system and storing them in the buffer cache. Because the virtual memory system cannot interface with the buffer cache, the contents of the file in the buffer cache must be copied into the page cache. This situation is known as **double caching** and requires caching file-system data twice. Not only is it wasteful of memory, but it wastes significant CPU and I/O cycles due to the extra data movement within system memory. Also, inconsistencies between the two caches can result in corrupt files. By providing a unified buffer cache, both memory mapping and the `read` and `write` system calls use the same page cache. This has the benefit of avoiding double caching and it allows the virtual memory system to manage file-system data. The unified buffer cache is shown in Figure 12.12.

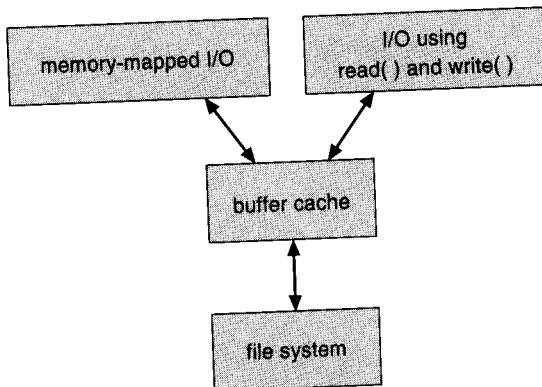


Figure 12.12 I/O using a unified buffer cache.

Regardless of whether we are caching disk blocks or pages, LRU seems a reasonable general-purpose algorithm for block or page replacement. However, the evolution of the Solaris page-caching algorithms reveals the difficulty in choosing an algorithm. Solaris allows processes and the page cache to share unused memory. Prior to Solaris 2.5.1, there was no distinction between allocating pages to a process or the page cache. As a result, a system performing many I/O operations uses most of the available memory for caching pages. Because of the high rates of I/O, the page scanner (Section 10.7.2) reclaims pages from processes—rather than the page cache—when free memory runs low. Solaris 2.6 and Solaris 7 optionally implemented *priority paging*, in which the page scanner gives priority to process pages over the page cache. Solaris 8 added a fixed limit between process pages and file-system page cache, preventing either from forcing the other out of memory.

The page cache, the file system, and the disk drivers have some interesting interactions. When data are written to a disk file, the pages are buffered in the cache, and the disk driver sorts its output queue according to disk address. These two actions allow the disk driver to minimize disk-head seeks and to write data at times optimized for disk rotation. Unless synchronous writes are required, a process writing to disk simply writes into the cache, and the system asynchronously writes the data to disk when convenient. The user process sees very fast writes. When data are read from a disk file, the block I/O system does some read-ahead; however, writes are much nearer to asynchronous than are reads. Thus, output to the disk through the file system is often faster than is input for large transfers, counter to intuition.

Synchronous writes occur in the order in which the disk subsystem receives them, and the writes are not buffered. Thus the calling routine must wait for the data to reach the disk drive before it can proceed. **Asynchronous writes** are done the majority of the time. In an asynchronous write the data is stored in the cache and returns control to the caller. Metadata writes, among

others, can be synchronous. Operating systems frequently include a flag in the open system call to allow a process to request that writes be performed synchronously. For example, databases use this feature for atomic transactions, to assure that data reaches stable storage in the required order.

Some systems optimize their page cache by using different replacement algorithms, depending on the access type of the file. A file being read or written sequentially should not have its pages replaced in LRU order, because the most recently used page will be used last, or perhaps never again. Instead, sequential access may be optimized by techniques known as free-behind and read-ahead. **Free-behind** removes a page from the buffer as soon as the next page is requested. The previous pages are not likely to be used again and waste buffer space. With **read-ahead**, a requested page and several subsequent pages are read and cached. These pages are likely to be requested after the current page is processed. Retrieving this data from the disk in one transfer and caching it saves a considerable amount of time. A track cache on the controller does not eliminate the need for read-ahead on a multiprogrammed system, because of the high latency and overhead of many small transfers from the track cache to main memory.

Another method of using main memory to improve performance is common on PCs. A section of memory is set aside and treated as a **virtual disk** (or **RAM disk**). In this case, a RAM-disk device driver accepts all the standard disk operations but performs those operations on the memory section, instead of on a disk. All disk operations can then be executed on this RAM disk and, except for the lightning-fast speed, users will not notice a difference. Unfortunately, RAM disks are useful only for temporary storage, since a power failure or a reboot of the system will usually erase them. Commonly, temporary files such as intermediate compiler files are stored there.

The difference between a RAM disk and a disk cache is that the contents of the RAM disk are totally user controlled, whereas those of the disk cache are under the control of the operating system. For instance, a RAM disk will stay empty until the user (or programs, at a user's direction) creates files there. Figure 12.13 shows the possible caching locations in a system.

12.7 ■ Recovery

Since files and directories are kept both in main memory and on disk, care must be taken to ensure that system failure does not result in loss of data or in data inconsistency.

12.7.1 Consistency Checking

As discussed in Section 12.3, part of the directory information is kept in main memory (or cache) to speed up access. The directory information in main

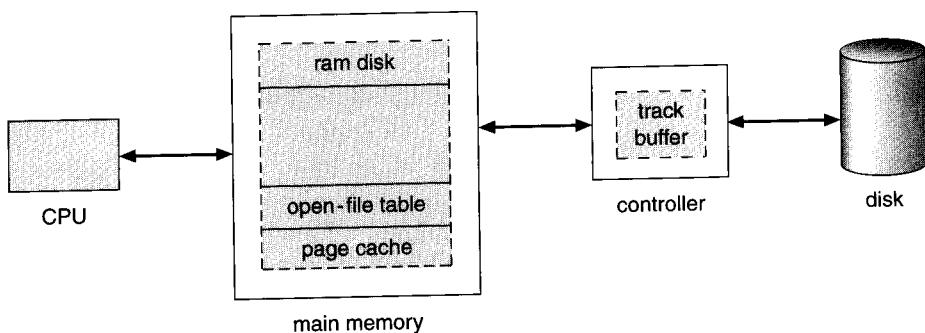


Figure 12.13 Various disk-caching locations.

memory is generally more up to date than is the corresponding information on the disk, because the write of cached directory information to disk does not necessarily occur as soon as the update takes place.

Consider the possible effect of a computer crash. In this case, the table of opened files is generally lost, and with it any changes in the directories of opened files. This event can leave the file system in an inconsistent state: The actual state of some files is not as described in the directory structure. Frequently, a special program is run at reboot time to check for and correct disk inconsistencies.

The **consistency checker** compares the data in the directory structure with the data blocks on disk, and tries to fix any inconsistencies it finds. The allocation and free-space-management algorithms dictate what types of problems the checker can find, and how successful it will be in fixing them. For instance, if linked allocation is used and there is a link from any block to its next block, then the entire file can be reconstructed from the data blocks, and the directory structure can be recreated. The loss of a directory entry on an indexed allocation system could be disastrous, because the data blocks have no knowledge of one another. For this reason, UNIX caches directory entries for reads, but any data write that results in space allocation, or other metadata changes, is done synchronously, before the corresponding data blocks are written.

12.7.2 Backup and Restore

Because magnetic disks sometimes fail, care must be taken to ensure that the data are not lost forever. To this end, system programs can be used to **back up** data from disk to another storage device, such as a floppy disk, magnetic tape, or optical disk. Recovery from the loss of an individual file, or of an entire disk, may then be a matter of **restoring** the data from backup.

To minimize the copying needed, we can use information from each file's directory entry. For instance, if the backup program knows when the last backup of a file was done, and the file's last write date in the directory indicates

that the file has not changed since that date, then the file does not need to be copied again. A typical backup schedule may then be as follows:

- **Day 1:** Copy to a backup medium all files from the disk. This is called a **full backup**.
- **Day 2:** Copy to another medium all files changed since day 1. This is an **incremental backup**.
- **Day 3:** Copy to another medium all files changed since day 2.
- •
•
- **Day N:** Copy to another medium all files changed since day $N - 1$. Then go back to Day 1.

The new cycle can have its backup written over the previous set, or onto a new set of backup media. In this manner, we can restore an entire disk by starting restores with the full backup, and continuing through each of the incremental backups. Of course, the larger N is, the more tapes or disks need to be read for a complete restore. An added advantage of this backup cycle is that we can restore any file accidentally deleted during the cycle by retrieving the deleted file from the backup of the previous day. The length of the cycle is a compromise between the amount of backup medium needed and the number of days back from which a restore can be done.

A user may notice that a particular file is missing or corrupted long after the damage was done. For this reason, we usually plan to take a full backup from time to time that will be saved “forever,” rather than reusing that backup medium. It is a good idea to store these permanent backups far away from the regular backups to protect against hazard, such as a fire that destroys the computer and all the backups too. And if the backup cycle reuses media, one must take care not to reuse the media too many times—if the media wear out, it might not be possible to restore any data from the backups.

12.8 ■ Log-Structured File System

Frequently in computer science, algorithms and technologies transition from their original use to other applicable areas. Such is the case with the database log-based-recovery algorithms described in Section 7.9.2. These logging algorithms have been applied successfully to the problem of consistency checking. The resulting implementations are known as **log-based transaction-oriented (or journaling) file systems**.

Recall that on-disk file-system data structures—such as the directory structures, free-block pointers, free FCB pointers—can become inconsistent due to a system crash. Before the use of log-based techniques in operating systems, changes were usually applied to these structures in place. A typical operation, such as file create, can involve many structural changes within the file system on the disk. Directory structures are modified, FCBs are allocated, data blocks are allocated, and the free counts for all of these blocks are decreased. Those changes can be interrupted by a crash, with the result that the structures are inconsistent. For example, the free FCB count might indicate that an FCB had been allocated, but the directory structure might not point to the FCB. The FCB would be lost were it not for the consistency-check phase.

There are several problems with the approach of allowing the structures to break and repairing them on recovery. One is that the inconsistency may be irreparable. The consistency check may not be able to recover the structures, with the resulting loss of files and even entire directories. Consistency checking can require human intervention to resolve conflicts, and that is inconvenient if no human is available. The system can remain unavailable until the human tells the system how to proceed. Consistency checking also takes system and clock time. Terabytes of data can take hours of clock time to check.

The solution to this problem is to apply log-based-recovery techniques to file-system metadata updates. NTFS and the Veritas File System both use this method, and it is an option to UFS on Solaris 7 and beyond. In fact, it is becoming common on many operating systems.

Fundamentally, all metadata changes are written sequentially to a log. Each set of operations that perform a specific task is a **transaction**. Once the changes are written to this log, they are considered to be committed, and the system call can return to the user process, allowing it to continue execution. Meanwhile, these log entries are replayed across the actual file-system structures. As the changes are made, a pointer is updated to indicate which actions have completed and which are still incomplete. When an entire committed transaction is completed, it is removed from the log file, which is actually a circular buffer. The log may be in a separate section of the file system, or could even be on a separate disk spindle. It is more efficient, but more complex, to have it under separate read/write heads, thereby decreasing head contention and seek times.

If the system crashes, there will zero or more transactions in the log file. Those transactions were never completed to the file system even though they were committed by the operating system, so they must be completed. The transactions can be executed from the pointer until the work is complete, and the file-system structures remain consistent. The only problem occurs when a transaction has been aborted. That is, it was not committed before the system crashed. Any changes from those transactions that were applied to the file system must be undone, again preserving the consistency of the file system.

This recovery is all that is needed after a crash, eliminating all problems with consistency checking.

A side benefit of using logging on disk metadata updates is that those updates proceed much faster than when they are applied directly to the on-disk data structures. The reason for this improvement is found in the performance advantage of sequential I/O over random I/O. The costly synchronous random metadata writes are turned into much less costly synchronous sequential writes to the log-structured file systems logging area. Those changes in turn are replayed asynchronously via random writes to the appropriate structures. The overall result is a significant gain in performance of metadata-oriented operations, such as file creation and deletion.

12.9 ■ NFS

Network file systems are commonplace. They typically integrate with the overall directory structure and interface of the client system. NFS is a good example of a widely used, well-implemented client–server network file system. Here, we use it as an example to explore the implementation details of network file systems.

NFS is both an implementation and a specification of a software system for accessing remote files across LANs (or even WANs). NFS is part of ONC+, which most UNIX vendors and some PC operating systems are supporting. The implementation described here is part of the Solaris operating system, which is a modified version of UNIX SVR4, running on Sun workstations and other hardware. It uses either the TCP or UDP/IP protocol (depending on the interconnecting network). The specification and the implementation are intertwined in our description of NFS. Whenever detail is needed, we refer to the Sun implementation; whenever the description is general, it applies to the specification also.

12.9.1 Overview

NFS views a set of interconnected workstations as a set of independent machines with independent file systems. The goal is to allow some degree of sharing among these file systems (on explicit request) in a transparent manner. Sharing is based on a client–server relationship. A machine may be, and often is, both a client and a server. Sharing is allowed between any pair of machines, rather than with only dedicated server machines. To ensure machine independence, sharing of a remote file system affects only the client machine and no other machine.

So that a remote directory will be accessible in a transparent manner from a particular machine—say, from *M1*—a client of that machine has to carry out a mount operation first. The semantics of the operation are that a remote directory is mounted over a directory of a local file system. Once the mount

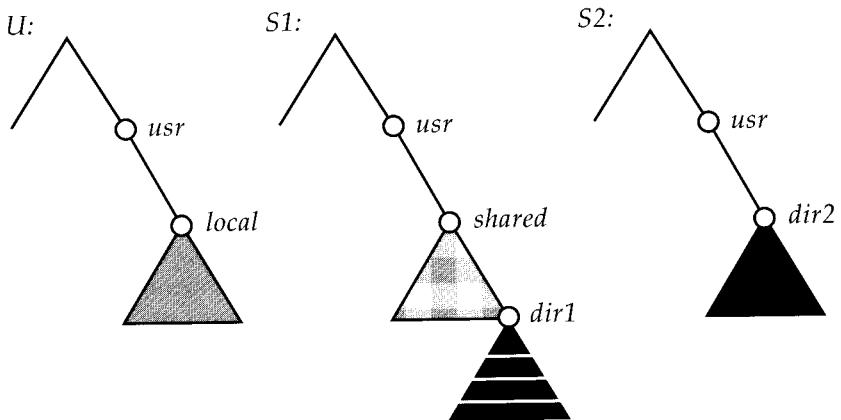


Figure 12.14 Three independent file systems.

operation is completed, the mounted directory looks like an integral subtree of the local file system, replacing the subtree descending from the local directory. The local directory becomes the name of the root of the newly mounted directory. Specification of the remote directory as an argument for the mount operation is done in a non-transparent manner; the location (or host name) of the remote directory has to be provided. However, from then on, users on machine M_1 can access files in the remote directory in a totally transparent manner.

To illustrate file mounting, consider the file system depicted in Figure 12.14, where the triangles represent subtrees of directories that are of interest. The figure shows three independent file systems of machines named U , S_1 , and S_2 . At this point, at each machine, only the local files can be accessed. In Figure 12.15(a), the effects of the mounting of $S_1:/usr/shared$ over $U:/usr/local$ are shown. This figure depicts the view users on U have of their file system. Observe that they can access any file within the $dir1$ directory, for instance, using the prefix $/usr/local/dir1$ on U after the mount is complete. The original directory $/usr/local$ on that machine is no longer visible.

Subject to access-rights accreditation, potentially any file system, or any directory within a file system, can be mounted remotely on top of any local directory. Diskless workstations can even mount their own roots from servers.

Cascading mounts are also permitted in some NFS implementations. That is, a file system can be mounted over another file system that is remotely mounted, not local. A machine is affected by only those mounts that it has itself invoked.

By mounting a remote file system, the client does not gain access to other file systems that were, by chance, mounted over the former file system. Thus, the mount mechanism does not exhibit a transitivity property. In Figure 12.15(b), we illustrate cascading mounts by continuing our previous example.

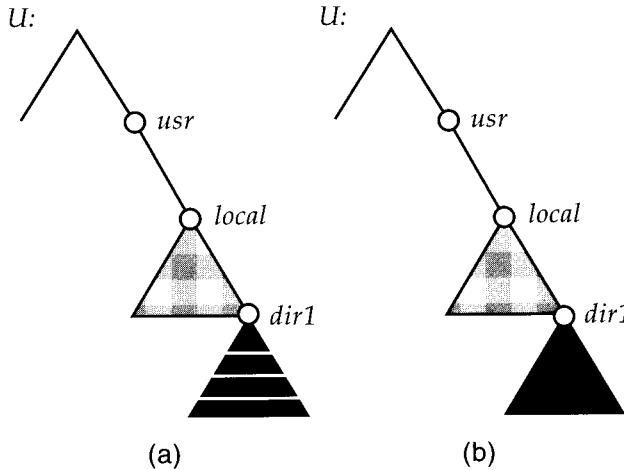


Figure 12.15 Mounting in NFS. (a) Mounts. (b) Cascading mounts.

The figure shows the result of mounting $S2:/usr/dir2$ over $U:/usr/local/dir1$, which is already remotely mounted from $S1$. Users can access files within $dir2$ on U using the prefix $/usr/local/dir1$. If a shared file system is mounted over a user's home directories on all machines in a network, a user can log in to any workstation and get his home environment. This property permits **user mobility**.

One of the design goals of NFS was to operate in a heterogeneous environment of different machines, operating systems, and network architectures. The NFS specification is independent of these media and thus encourages other implementations. This independence is achieved through the use of RPC primitives built on top of an External Data Representation (XDR) protocol used between two implementation-independent interfaces. Hence, if the system consists of heterogeneous machines and file systems that are properly interfaced to NFS, file systems of different types can be mounted both locally and remotely.

The NFS specification distinguishes between the services provided by a mount mechanism and the actual remote-file-access services. Accordingly, two separate protocols are specified for these services: a mount protocol, and a protocol for remote file accesses, the **NFS protocol**. The protocols are specified as sets of RPCs. These RPCs are the building blocks used to implement transparent remote file access.

12.9.2 The Mount Protocol

The **mount protocol** establishes the initial logical connection between a server and a client. In Sun's implementation, each machine has a server process, outside the kernel, performing the protocol functions.

A mount operation includes the name of the remote directory to be mounted and the name of the server machine storing it. The mount request is mapped to the corresponding RPC and is forwarded to the mount server running on the specific server machine. The server maintains an **export list**—the */etc/dfs/dfstab* in Solaris, which can be edited only by a superuser—which specifies local file systems that it exports for mounting, along with names of machines that are permitted to mount them. The specification can also include access rights, such as read only. To simplify the maintenance of export lists and mount tables, a distributed naming scheme can be used to hold this information and make it available to appropriate clients.

Recall that any directory within an exported file system can be mounted remotely by an accredited machine. Hence, a component unit is such a directory. When the server receives a mount request that conforms to its export list, it returns to the client a **file handle** that serves as the key for further accesses to files within the mounted file system. The file handle contains all the information that the server needs to distinguish an individual file it stores. In UNIX terms, the file handle consists of a file-system identifier and an inode number to identify the exact mounted directory within the exported file system.

The server also maintains a list of the client machines and the corresponding currently mounted directories. This list is used mainly for administrative purposes—for instance, for notifying all clients that the server is going down. Addition and deletion of entries in this list are the only ways that the server state can be affected by the mount protocol.

Usually, a system has a static mounting preconfiguration that is established at boot time (*/etc/vfstab* in Solaris); however, this layout can be modified. In addition to the actual mount procedure, the mount protocol includes several other procedures, such as unmount and return export list.

12.9.3 The NFS Protocol

The NFS protocol provides a set of RPCs for remote file operations. The procedures support the following operations:

- Searching for a file within a directory
- Reading a set of directory entries
- Manipulating links and directories
- Accessing file attributes
- Reading and writing files

These procedures can be invoked only after a file handle for the remotely mounted directory has been established.

The omission of open and close operations is intentional. A prominent feature of NFS servers is that they are *stateless*. Servers do not maintain information about their clients from one access to another. No parallels to UNIX's open-files table or file structures exist on the server side. Consequently, each request has to provide a full set of arguments, including a unique file identifier and an absolute offset inside the file for the appropriate operations. The resulting design is robust; no special measures need to be taken to recover a server after a crash. File operations need to be idempotent for this purpose. Every NFS request has a sequence number, allowing the server to determine if a request is duplicated or if any are missing.

Maintaining the list of clients that we mentioned seems to violate the statelessness of the server. However, this list is not essential for the correct operation of the client or the server, and hence it does not need to be restored after a server crash. Consequently, it might include inconsistent data and is treated as only a hint.

A further implication of the stateless-server philosophy and a result of the synchrony of an RPC is that modified data (including indirection and status blocks) must be committed to the server's disk before results are returned to the client. That is, a client can cache write blocks, but when it flushes them to the server, it assumes that they have reached the server's disks. The server must write all NFS data synchronously. Thus, a server crash and recovery will be invisible to a client; all blocks that the server is managing for the client will be intact. The consequent performance penalty can be large, because the advantages of caching are lost. Performance can be increased by using storage with its own nonvolatile cache (usually battery-backed-up memory). The disk controller acknowledges the disk write when the write is stored in the nonvolatile cache. In essence the host sees a very fast synchronous write. These blocks remain intact even after system crash, and are written from this stable storage to disk periodically.

A single NFS write procedure call is guaranteed to be atomic, and also is not intermixed with other write calls to the same file. The NFS protocol, however, does not provide concurrency-control mechanisms. A `write` system call may be broken down into several RPC writes, because each NFS write or read call can contain up to 8 KB of data and UDP packets are limited to 1,500 bytes. As a result, two users writing to the same remote file may get their data intermixed. The claim is that, because lock management is inherently stateful, a service outside the NFS should provide locking (and Solaris does). Users are advised to coordinate access to shared files using mechanisms outside the scope of NFS.

NFS is integrated into the operating system via a VFS. As an illustration of the architecture, let us trace how an operation on an already open remote file is handled (follow the example in Figure 12.16). The client initiates the operation by a regular system call. The operating-system layer maps this call to a VFS operation on the appropriate vnode. The VFS layer identifies the file as a remote one and invokes the appropriate NFS procedure. An RPC call is made to the

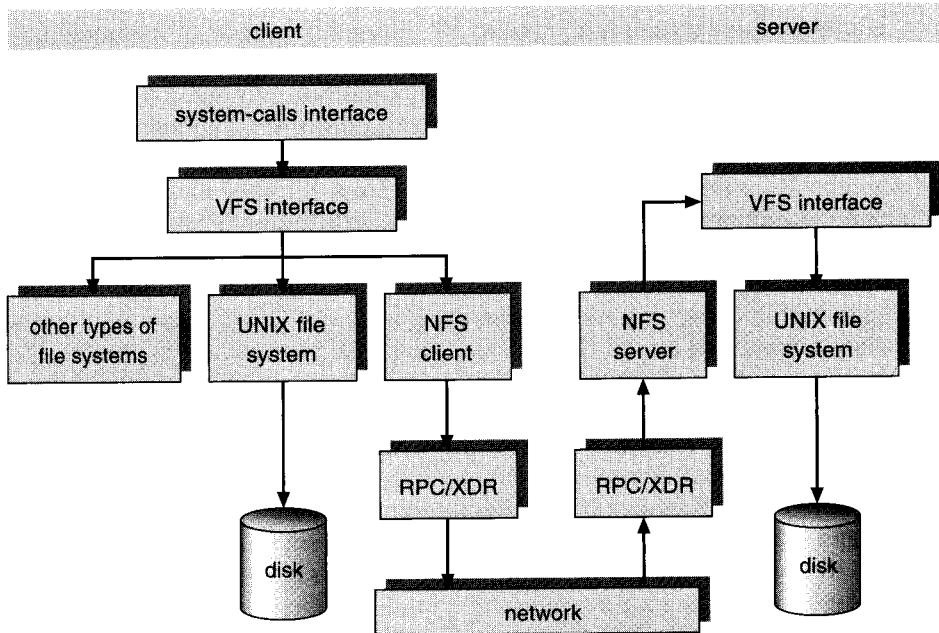


Figure 12.16 Schematic view of the NFS architecture.

NFS service layer at the remote server. This call is reinjected to the VFS layer on the remote system, which finds that it is local and invokes the appropriate file-system operation. This path is retraced to return the result. An advantage of this architecture is that the client and the server are identical; thus, a machine may be a client, or a server, or both.

The actual service on each server is performed by several kernel processes that provide a temporary substitute to a lightweight process (or threads) mechanism.

12.9.4 Path-Name Translation

Path-name translation is done by breaking the path into component names and performing a separate NFS lookup call for every pair of component name and directory vnode. Once a mount point is crossed, every component lookup causes a separate RPC to the server. This expensive path-name-traversal scheme is needed, since each client has a unique layout of its logical name space, dictated by the mounts it performed. It would have been much more efficient to hand a server a path name and to receive a target vnode once a mount point was encountered. At any point, however, there can be another mount point for the particular client of which the stateless server is unaware.

So that lookup is fast, a directory name lookup cache on the client side holds the vnodes for remote directory names. This cache speeds up references to files with the same initial path name. The directory cache is discarded when attributes returned from the server do not match the attributes of the cached vnode.

Recall that mounting a remote file system on top of another already mounted remote file system (cascading mount) is allowed in some implementations of NFS. However, a server cannot act as an intermediary between a client and another server. Instead, a client must establish a direct client–server connection with the second server by directly mounting the desired directory. When a client has a cascading mount, more than one server can be involved in a path-name traversal. However, each component lookup is performed between the original client and some server. Therefore, when a client does a lookup on a directory on which the server has mounted a file system, the client sees the underlying directory, instead of the mounted directory.

12.9.5 Remote Operations

With the exception of opening and closing files, there is almost a one-to-one correspondence between the regular UNIX system calls for file operations and the NFS protocol RPCs. Thus, a remote file operation can be translated directly to the corresponding RPC. Conceptually, NFS adheres to the remote-service paradigm, but in practice buffering and caching techniques are employed for the sake of performance. No direct correspondence exists between a remote operation and an RPC. Instead, file blocks and file attributes are fetched by the RPCs and are cached locally. Future remote operations use the cached data, subject to consistency constraints.

There are two caches: the file-attribute (inode-information) cache and the file-blocks cache. On a file open, the kernel checks with the remote server whether to fetch or revalidate the cached attributes. The cached file blocks are used only if the corresponding cached attributes are up to date. The attribute cache is updated whenever new attributes arrive from the server. Cached attributes are, by default, discarded after 60 seconds. Both read-ahead and delayed-write techniques are used between the server and the client. Clients do not free delayed-write blocks until the server confirms that the data have been written to disk. In contrast to the system used in Sprite, delayed-write is retained even when a file is opened concurrently, in conflicting modes. Hence, UNIX semantics are not preserved.

Tuning the system for performance makes it difficult to characterize the consistency semantics of NFS. New files created on a machine may not be visible elsewhere for 30 seconds. It is indeterminate whether writes to a file at one site are visible to other sites that have this file open for reading. New opens of that file observe only the changes that have already been flushed to the server. Thus, NFS provides neither strict emulation of UNIX semantics, nor the

session semantics of Andrew. In spite of these drawbacks, the utility and high performance of the mechanism makes it the most widely used multivendor distributed system in operation.

12.10 ■ Summary

The file system resides permanently on secondary storage, which is designed to hold a large amount of data permanently. The most common secondary-storage medium is the disk.

Physical disks may be segmented into partitions to control media use and to allow multiple, possibly varying, file systems per spindle. These file systems are mounted onto a logical file system architecture to make them available for use. File systems are often implemented in a layered or modular structure. The lower levels deal with the physical properties of storage devices. Upper levels deal with symbolic file names and logical properties of files. Intermediate levels map the logical file concepts into physical device properties.

Every file-system type can have different structures and algorithms. A VFS layer allows the upper layers to deal with each file-system type uniformly. Even remote file systems can be integrated into the system's directory structure and acted on by standard system calls via the VFS interface.

The various files can be allocated space on the disk in three ways: through contiguous, linked, or indexed allocation. Contiguous allocation can suffer from external fragmentation. Direct access is very inefficient with linked allocation. Indexed allocation may require substantial overhead for its index block. These algorithms can be optimized in many ways. Contiguous space may be enlarged through extents to increase flexibility and to decrease external fragmentation. Indexed allocation can be done in clusters of multiple blocks to increase throughput and to reduce the number of index entries needed. Indexing in large clusters is similar to contiguous allocation with extents.

Free-space allocation methods also influence the efficiency of use of disk space, the performance of the file system, and the reliability of secondary storage. The methods used include bit vectors and linked lists. Optimizations include grouping, counting, and the FAT, which places the linked list in one contiguous area.

The directory-management routines must consider efficiency, performance, and reliability. A hash table is the most frequently used method; it is fast and efficient. Unfortunately, damage to the table or a system crash could result in the directory information not corresponding to the disk's contents. A consistency checker—a systems program such as `fsck` in UNIX, or `chkdsk` in MS-DOS—can be used to repair the damage. Operating-system backup tools allow disk data to be copied to tape, to recover from data or even disk loss due to hardware failure, operating system bug, or user error.

Network file systems, such as NFS, use client–server methodology to allow users to access files and directories from remote machines as if they were on local file systems. System calls on the client are translated into network protocols, and retranslated into file-system operations on the server. Networking and multiple-client access create challenges in the areas of data consistency and performance.

Due to the fundamental role that file systems play in system operation, their performance and reliability is crucial. Techniques such as log structures and caching help improve the performance, while log structures and RAID improve reliability.

■ Exercises

- 12.1** Consider a file currently consisting of 100 blocks. Assume that the FCB (and the index block, in the case of indexed allocation) is already in memory. Calculate how many disk I/O operations are required for contiguous, linked, and indexed (single-level) allocation strategies, if, for one block, the following conditions hold. In the contiguous-allocation case, assume that there is no room to grow in the beginning, but room to grow in the end. Assume that the block information to be added is stored in memory.
- The block is added at the beginning.
 - The block is added in the middle.
 - The block is added at the end.
 - The block is removed from the beginning.
 - The block is removed from the middle.
 - The block is removed from the end.
- 12.2** Consider a system where free space is kept in a free-space list.
- Suppose that the pointer to the free-space list is lost. Can the system reconstruct the free-space list? Explain your answer.
 - Suggest a scheme to ensure that the pointer is never lost as a result of memory failure.
- 12.3** What problems could occur if a system allowed a file system to be mounted simultaneously at more than one location?
- 12.4** Why must the bit map for file allocation be kept on mass storage, rather than in main memory?

- 12.5 Consider a system that supports the strategies of contiguous, linked, and indexed allocation. What criteria should be used in deciding which strategy is best utilized for a particular file?
- 12.6 Consider a file system on a disk that has both logical and physical block sizes of 512 bytes. Assume that the information about each file is already in memory. For each of the three allocation strategies (contiguous, linked, and indexed), answer these questions:
- How is the logical-to-physical address mapping accomplished in this system? (For the indexed allocation, assume that a file is always less than 512 blocks long.)
 - If we are currently at logical block 10 (the last block accessed was block 10) and want to access logical block 4, how many physical blocks must be read from the disk?
- 12.7 One problem with contiguous allocation is that the user must preallocate enough space for each file. If the file grows to be larger than the space allocated for it, special actions must be taken. One solution is to define a file structure consisting of an initial contiguous area (of a specified size). If this area is filled, the operating system automatically defines an overflow area that is linked to the initial contiguous area. If the overflow area is filled, another overflow area is allocated. Compare this implementation of a file with the standard contiguous and linked implementations.
- 12.8 Fragmentation on a storage device could be eliminated by recompaction of the information. Typical disk devices do not have relocation or base registers (such as are used when memory is to be compacted), so how can we relocate files? Give three reasons why recompacting and relocation of files are often avoided.
- 12.9 How do caches help improve performance? Why do systems not use more or larger caches if they are so useful?
- 12.10 In what situations would using memory as a RAM disk be more useful than using it as a disk cache?
- 12.11 Why is it advantageous to the user for an operating system to dynamically allocate its internal tables? What are the penalties to the operating system for doing so?
- 12.12 Explain why logging metadata updates ensures recovery of a file system after a file system crash.
- 12.13 Explain how the VFS layer allows an operating system easily to support multiple types of file systems.

12.14 Consider the following backup scheme:

- **Day 1:** Copy to a backup medium all files from the disk.
- **Day 2:** Copy to another medium all files changed since day 1.
- **Day 3:** Copy to another medium all files changed since day 1.

This contrasts to the schedule given in Section 12.7.2 by having all subsequent backups copy all files modified since the first full backup. What are the benefits of this system over the one in Section 12.7.2? What are the drawbacks? Are restore operations made easier or more difficult? Explain your answer.

Bibliographical Notes

The Apple Macintosh disk-space management scheme was discussed in Apple [1987] and Apple [1991]. The MS-DOS FAT system was explained in Norton and Wilton [1988], and the OS/2 description is found in Iacobucci [1988]. These operating systems use the Motorola MC68000 family (Motorola [1989a]) and the Intel 8086 (Intel [1985b], Intel [1985a], Intel [1986], Intel [1990]) CPUs, respectively. IBM allocation methods were described in Deitel [1990]. The internals of the BSD UNIX system were covered in full in McKusick et al. [1996]. McVoy and Kleiman [1991] presented optimizations to these methods made in SunOS.

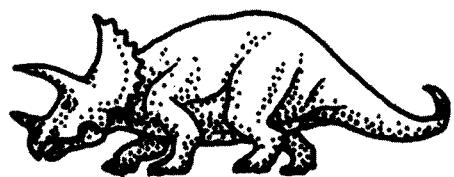
Disk file allocation based on the buddy system was discussed by Koch [1987]. A file-organization scheme that guarantees retrieval in one access was discussed by Larson and Kajla [1984].

Disk caching was discussed by McKeon [1985] and Smith [1985]. Caching in the experimental Sprite operating system was described in Nelson et al. [1988]. General discussions concerning mass-storage technology were offered by Chi [1982] and Hoagland [1985]. Folk and Zoellick [1987] covered the gamut of file structures. Silvers [2000] discusses implementing the page cache in the NetBSD operating system.

The Network File System (NFS) is discussed in Sandberg et al. [1985], Sandberg [1987], Sun [1990], and Callaghan [2000]. NFS and the UNIX File System (UFS) are described in Vahalia [1996] and Mauro and McDougall [2001]. The Windows NT file system, NTFS, is described in Solomon [1998]. The Ext2 file system used in Linux is described in Bovet and Cesati [2001].

Part Four

I/O SYSTEMS

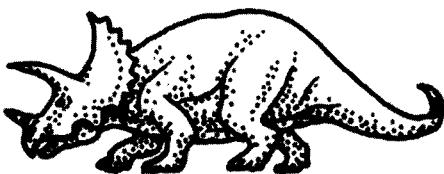


The devices that attach to a computer vary in many aspects. Some devices transfer a character or a block of characters at a time. Some can be accessed only sequentially, others randomly. Some transfer data synchronously, others asynchronously. Some are dedicated, some shared. They can be read-only or read-write. They vary greatly in speed. In many ways they are also the slowest major component of the computer.

Because of all this device variation, the operating system needs to provide a wide range of functionality to applications, to allow them to control all aspects of the devices. One key goal of an operating system's I/O subsystem is to provide the simplest interface possible to the rest of the system. Because devices are a performance bottleneck, another key is to optimize I/O for maximum concurrency. We initially describe the myriad variations of I/O devices and the ways in which operating systems control them. Afterwards we discuss the more complicated I/O devices used for secondary and tertiary storage, and we explain the special attention that operating systems must give them.

Chapter 13

I/O SYSTEMS



The two main jobs of a computer are I/O and processing. In many cases, the main job is I/O and the processing is merely incidental. For instance, when we browse a web page or edit a file, our immediate interest is to read or enter some information, not to compute an answer.

The role of the operating system in computer I/O is to manage and control I/O operations and I/O devices. Although related topics appear in other chapters, here we bring together the pieces to paint a complete picture of I/O. First, we describe the basics of I/O hardware, because the nature of the hardware interface places requirements on the internal facilities of the operating system. Next, we discuss the I/O services provided by the operating system, and the embodiment of these services in the application I/O interface. Then, we explain how the operating system bridges the gap between the hardware interface and the application interface. We also discuss the UNIX System V STREAMS mechanism, which enables an application to assemble pipelines of driver code dynamically. Finally, we discuss the performance aspects of I/O, and the principles of operating-system design that improve the I/O performance.

13.1 ■ Overview

The control of devices connected to the computer is a major concern of operating-system designers. Because I/O devices vary so widely in their function and speed (consider a mouse, a hard disk, and a CD-ROM jukebox), a

variety of methods are needed to control them. These methods form the *I/O subsystem* of the kernel, which separates the rest of the kernel from the complexity of managing I/O devices.

I/O-device technology exhibits two conflicting trends. On one hand, we see increasing standardization of software and hardware interfaces. This trend helps us to incorporate improved device generations into existing computers and operating systems. On the other hand, we see an increasingly broad variety of I/O devices. Some new devices are so unlike previous devices that it is a challenge to incorporate them into our computers and operating systems. This challenge is met by a combination of hardware and software techniques. The basic I/O hardware elements, such as ports, buses, and device controllers, accommodate a wide variety of I/O devices. To encapsulate the details and oddities of different devices, the kernel of an operating system is structured to use device-driver modules. The **device drivers** present a uniform device-access interface to the I/O subsystem, much as system calls provide a standard interface between the application and the operating system.

13.2 ■ I/O Hardware

Computers operate a great many kinds of devices. Most fit into the general categories of storage devices (disks, tapes), transmission devices (network cards, modems), and human-interface devices (screen, keyboard, mouse). Other devices are more specialized, such as the steering of a military fighter jet or a space shuttle. In these aircraft, a human gives input to the flight computer via a joystick, and the computer sends output commands that cause motors to move rudders, flaps, and thrusters.

Despite the incredible variety of I/O devices, we need only a few concepts to understand how the devices are attached, and how the software can control the hardware.

A device communicates with a computer system by sending signals over a cable or even through the air. The device communicates with the machine via a connection point (or **port**), for example, a serial port. If one or more devices use a common set of wires, the connection is called a *bus*. A **bus** is a set of wires and a rigidly defined protocol that specifies a set of messages that can be sent on the wires. In terms of the electronics, the messages are conveyed by patterns of electrical voltages applied to the wires with defined timings. When device *A* has a cable that plugs into device *B*, and device *B* has a cable that plugs into device *C*, and device *C* plugs into a port on the computer, this arrangement is called a **daisy chain**. A daisy chain usually operates as a bus.

Buses are used widely in computer architecture. Figure 13.1 shows a typical PC bus structure. This figure shows a **PCI bus** (the common PC system bus) that connects the processor–memory subsystem to the fast devices, and an **expansion bus** that connects relatively slow devices such as the keyboard and

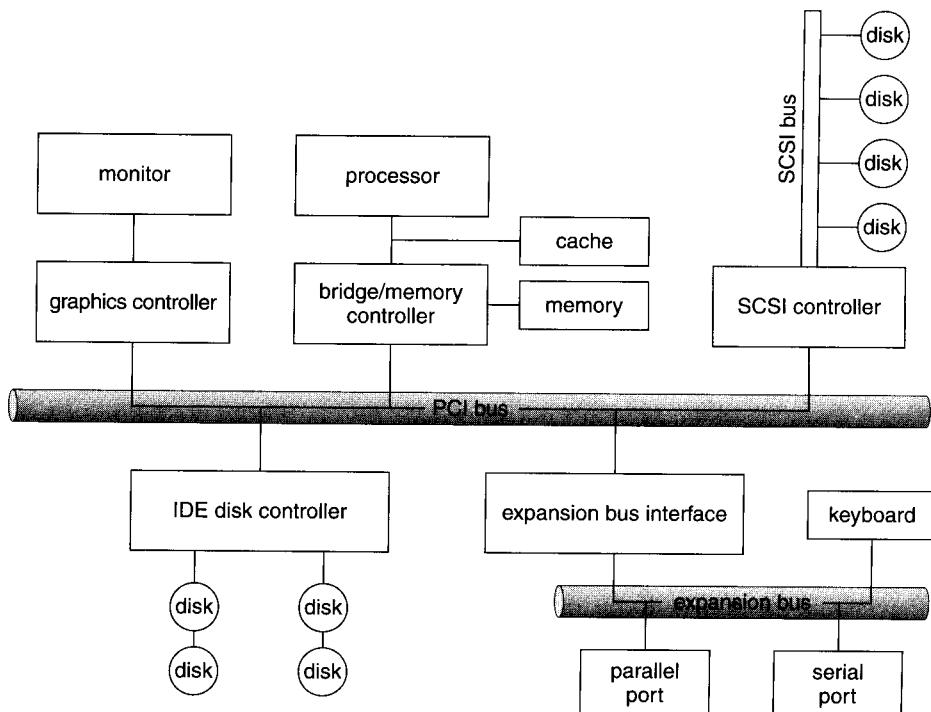


Figure 13.1 A typical PC bus structure.

serial and parallel ports. In the upper-right portion of the figure, four disks are connected together on a SCSI bus plugged into a SCSI controller.

A **controller** is a collection of electronics that can operate a port, a bus, or a device. A serial-port controller is a simple device controller. It is a single chip (or portion of a chip) in the computer that controls the signals on the wires of a serial port. By contrast, a SCSI bus controller is not simple. Because the SCSI protocol is complex, the SCSI bus controller is often implemented as a separate circuit board (or a **host adapter**) that plugs into the computer. It typically contains a processor, microcode, and some private memory to enable it to process the SCSI protocol messages. Some devices have their own built-in controllers. If you look at a disk drive, you will see a circuit board attached to one side. This board is the disk controller. It implements the disk side of the protocol for some kind of connection, SCSI or IDE, for instance. It has microcode and a processor to do many tasks, such as bad-sector mapping, prefetching, buffering, and caching.

How can the processor give commands and data to a controller to accomplish an I/O transfer? The short answer is that the controller has one or more registers for data and control signals. The processor communicates with the controller by reading and writing bit patterns in these registers. One way that

this communication can occur is through the use of special I/O instructions that specify the transfer of a byte or word to an I/O port address. The I/O instruction triggers bus lines to select the proper device and to move bits into or out of a device register. Alternatively, the device controller can support **memory-mapped** I/O. In this case, the device-control registers are mapped into the address space of the processor. The CPU executes I/O requests using the standard data-transfer instructions to read and write the device-control registers.

Some systems use both techniques. For instance, PCs use I/O instructions to control some devices and memory-mapped I/O to control others. Figure 13.2 shows the usual PC I/O port addresses. The graphics controller has I/O ports for basic control operations, but the controller has a large memory-mapped region to hold screen contents. The process sends output to the screen by writing data into the memory-mapped region. The controller generates the screen image based on the contents of this memory. This technique is simple to use. Moreover, writing millions of bytes to the graphics memory is faster than issuing millions of I/O instructions. But the ease of writing to a memory-mapped I/O controller is offset by a disadvantage. Because a common type of software fault is a write through an incorrect pointer to an unintended region of memory, a memory-mapped device register is vulnerable to accidental modification. Of course, protected memory helps to reduce this risk.

An I/O port typically consists of four registers, called the *status*, *control*, *data-in*, and *data-out* registers.

I/O address range (hexadecimal)	device
000-00F	DMA controller
020-021	interrupt controller
040-043	timer
200-20F	game controller
2F8-2FF	serial port (secondary)
320-32F	hard-disk controller
378-37F	parallel port
3D0-3DF	graphics controller
3F0-3F7	diskette-drive controller
3F8-3FF	serial port (primary)

Figure 13.2 Device I/O port locations on PCs (partial).

- The *status* register contains bits that can be read by the host. These bits indicate states such as whether the current command has completed, whether a byte is available to be read from the data-in register, and whether there has been a device error.
- The *control* register can be written by the host to start a command or to change the mode of a device. For instance, a certain bit in the *control* register of a serial port chooses between full-duplex and half-duplex communication, another enables parity checking, a third bit sets the word length to 7 or 8 bits, and other bits select one of the speeds supported by the serial port.
- The *data-in* register is read by the host to get input.
- The *data-out* register is written by the host to send output.

The data registers are typically 1 to 4 bytes. Some controllers have FIFO chips that can hold several bytes of input or output data to expand the capacity of the controller beyond the size of the data register. A FIFO chip can hold a small burst of data until the device or host is able to receive those data.

13.2.1 Polling

The complete protocol for interaction between the host and a controller can be intricate, but the basic *handshaking* notion is simple. We explain handshaking by an example. We assume that 2 bits are used to coordinate the producer-consumer relationship between the controller and the host. The controller indicates its state through the *busy* bit in the *status* register. (Recall that to *set* a bit means to write a 1 into the bit, and to *clear* a bit means to write a 0 into it.) The controller sets the *busy* bit when it is busy working, and clears the *busy* bit when it is ready to accept the next command. The host signals its wishes via the *command-ready* bit in the *command* register. The host sets the *command-ready* bit when a command is available for the controller to execute. For this example, the host writes output through a port, coordinating with the controller by handshaking as follows.

1. The host repeatedly reads the *busy* bit until that bit becomes clear.
2. The host sets the *write* bit in the *command* register and writes a byte into the *data-out* register.
3. The host sets the *command-ready* bit.
4. When the controller notices that the *command-ready* bit is set, it sets the *busy* bit.
5. The controller reads the *command* register and sees the *write* command. It reads the *data-out* register to get the byte, and does the I/O to the device.

6. The controller clears the *command-ready* bit, clears the *error* bit in the status register to indicate that the device I/O succeeded, and clears the *busy* bit to indicate that it is finished.

This loop is repeated for each byte.

In step 1, the host is **busy-waiting** or **polling**: It is in a loop, reading the *status* register over and over until the *busy* bit becomes clear. If the controller and device are fast, this method is a reasonable one. But if the wait may be long, the host should probably switch to another task. How then does the host know when the controller has become idle? For some devices, the host must service the device quickly, or data will be lost. For instance, when data are streaming in on a serial port or from a keyboard, the small buffer on the controller will overflow and data will be lost if the host waits too long before returning to read the bytes.

In many computer architectures, three CPU-instruction cycles are sufficient to poll a device: *read* a device register, *logical-and* to extract a status bit, and *branch* if not zero. Clearly, the basic polling operation is efficient. But polling becomes inefficient when it is attempted repeatedly, yet rarely finds a device to be ready for service, while other useful CPU processing remains undone. In such instances, it may be more efficient to arrange for the hardware controller to notify the CPU when the device becomes ready for service, rather than to require the CPU to poll repeatedly for an I/O completion. The hardware mechanism that enables a device to notify the CPU is called an **interrupt**.

13.2.2 Interrupts

The basic interrupt mechanism works as follows. The CPU hardware has a wire called the **interrupt-request** line that the CPU senses after executing every instruction. When the CPU detects that a controller has asserted a signal on the interrupt request line, the CPU saves a small amount of state, such as the current value of the instruction pointer, and jumps to the **interrupt-handler** routine at a fixed address in memory. The interrupt handler determines the cause of the interrupt, performs the necessary processing, and executes a **return from interrupt** instruction to return the CPU to the execution state prior to the interrupt. We say that the device controller *raises* an interrupt by asserting a signal on the interrupt request line, the CPU *catches* the interrupt and *dispatches* to the interrupt handler, and the handler *clears* the interrupt by servicing the device. Figure 13.3 summarizes the interrupt-driven I/O cycle.

This basic interrupt mechanism enables the CPU to respond to an asynchronous event, such as a device controller becoming ready for service. In a modern operating system, we need more sophisticated interrupt-handling features. First, we need the ability to defer interrupt handling during critical processing. Second, we need an efficient way to dispatch to the proper interrupt handler for a device, without first polling all the devices to see which one raised the interrupt. Third, we need multilevel interrupts, so that the operating system

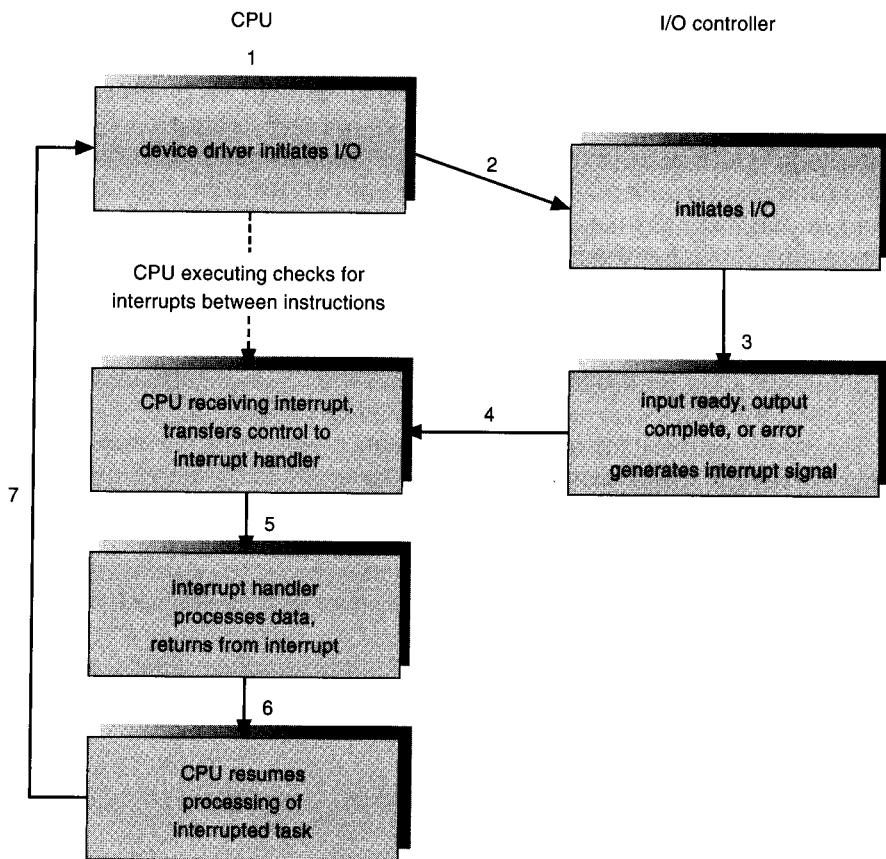


Figure 13.3 Interrupt-driven I/O cycle.

can distinguish between high- and low-priority interrupts, and can respond with the appropriate degree of urgency. In modern computer hardware, these three features are provided by the CPU and by the **interrupt-controller** hardware.

Most CPUs have two interrupt request lines. One is the **nonmaskable interrupt**, which is reserved for events such as unrecoverable memory errors. The second interrupt line is **maskable**: It can be turned off by the CPU before the execution of critical instruction sequences that must not be interrupted. The maskable interrupt is used by device controllers to request service.

The interrupt mechanism accepts an **address**—a number that selects a specific interrupt-handling routine from a small set. In most architectures, this address is an offset in a table called the **interrupt vector**. This vector contains the memory addresses of specialized interrupt handlers. The purpose of a vectored interrupt mechanism is to reduce the need for a single interrupt

handler to search all possible sources of interrupts to determine which one needs service. In practice, however, computers have more devices (and hence, interrupt handlers) than they have address elements in the interrupt vector. A common way to solve this problem is to use the technique of interrupt chaining, in which each element in the interrupt vector points to the head of a list of interrupt handlers. When an interrupt is raised, the handlers on the corresponding list are called one by one, until one is found that can service the request. This structure is a compromise between the overhead of a huge interrupt table and the inefficiency of a dispatching to a single interrupt handler.

Figure 13.4 illustrates the design of the interrupt vector for the Intel Pentium processor. The events from 0 to 31, which are nonmaskable, are used to signal various error conditions. The events from 32 to 255, which are maskable, are used for purposes such as device-generated interrupts.

The interrupt mechanism also implements a system of **interrupt priority levels**. This mechanism enables the CPU to defer the handling of low-priority interrupts without masking off all interrupts, and makes it possible for a high-priority interrupt to preempt the execution of a low-priority interrupt.

A modern operating system interacts with the interrupt mechanism in several ways. At boot time, the operating system probes the hardware buses to determine what devices are present, and installs the corresponding interrupt handlers into the interrupt vector. During I/O, the various device controllers

vector number	description
0	divide error
1	debug exception
2	null interrupt
3	breakpoint
4	INTO-detected overflow
5	bound range exception
6	invalid opcode
7	device not available
8	double fault
9	coprocessor segment overrun (reserved)
10	invalid task state segment
11	segment not present
12	stack fault
13	general protection
14	page fault
15	(Intel reserved, do not use)
16	floating-point error
17	alignment check
18	machine check
19–31	(Intel reserved, do not use)
32–255	maskable interrupts

Figure 13.4 Intel Pentium processor event-vector table.

raise interrupts when they are ready for service. These interrupts signify that output has completed, or that input data are available, or that a failure has been detected. The interrupt mechanism is also used to handle a wide variety of **exceptions**, such as dividing by zero, accessing a protected or nonexistent memory address, or attempting to execute a privileged instruction from user mode. The events that trigger interrupts have a common property: They are occurrences that induce the CPU to execute an urgent, self-contained routine.

An operating system has other good uses for an efficient hardware mechanism that saves a small amount of processor state, and then calls a privileged routine in the kernel. For example, many operating systems use the interrupt mechanism for virtual-memory paging. A page fault is an exception that raises an interrupt. The interrupt suspends the current process and jumps to the page-fault handler in the kernel. This handler saves the state of the process, moves the process to the wait queue, performs page-cache management, schedules an I/O operation to fetch the page, schedules another process to resume execution, and then returns from the interrupt.

Another example is found in the implementation of system calls. A *system call* is a function called by an application to invoke a kernel service. The system call checks the arguments given by the application, builds a data structure to convey the arguments to the kernel, and then executes a special instruction called a **software interrupt** (or a **trap**). This instruction has an operand that identifies the desired kernel service. When the system call executes the trap instruction, the interrupt hardware saves the state of the user code, switches to supervisor mode, and dispatches to the kernel routine that implements the requested service. The trap is given a relatively low interrupt priority compared to those assigned to device interrupts—executing a system call on behalf of an application is less urgent than servicing a device controller before its FIFO queue overflows and loses data.

Interrupts can also be used to manage the flow of control within the kernel. For example, consider the processing required to complete a disk read. One step is to copy data from kernel space to the user buffer. This copying is time consuming but not urgent—it should not block other high-priority interrupt handling. Another step is to start the next pending I/O for that disk drive. This step has higher priority: If the disks are to be used efficiently, we need to start the next I/O as soon as the previous one completes. Consequently, a *pair* of interrupt handlers implements the kernel code that completes a disk read. The high-priority handler records the I/O status, clears the device interrupt, starts the next pending I/O, and raises a low-priority interrupt to complete the work. Later, when the CPU is not occupied with high-priority work, the low-priority interrupt will be dispatched. The corresponding handler completes the user-level I/O by copying data from kernel buffers to the application space, and then by calling the scheduler to place the application on the ready queue.

A threaded kernel architecture is well suited to implement multiple interrupt priorities and to enforce the precedence of interrupt handling over back-

ground processing in kernel and application routines. We illustrate this point with the Solaris kernel. In Solaris, interrupt handlers are executed as kernel threads. A range of high priorities is reserved for these threads. These priorities give interrupt handlers precedence over application code and kernel housekeeping, and implement the priority relationships among interrupt handlers. The priorities cause the Solaris thread scheduler to preempt low-priority interrupt handlers in favor of higher-priority ones, and the threaded implementation enables multiprocessor hardware to run several interrupt handlers concurrently. We describe the interrupt architecture of UNIX and Windows NT in Appendix A and Chapters 21, respectively.

In summary, interrupts are used throughout modern operating systems to handle asynchronous events and to trap to supervisor-mode routines in the kernel. To enable the most urgent work to be done first, modern computers use a system of interrupt priorities. Device controllers, hardware faults, and system calls all raise interrupts to trigger kernel routines. Because interrupts are used so heavily for time-sensitive processing, efficient interrupt handling is required for good system performance.

13.2.3 Direct Memory Access

For a device that does large transfers, such as a disk drive, it seems wasteful to use an expensive general-purpose processor to watch status bits and to feed data into a controller register 1 byte at a time—a process termed **programmed I/O (PIO)**. Many computers avoid burdening the main CPU with PIO by offloading some of this work to a special-purpose processor called a **direct-memory-access (DMA)** controller. To initiate a DMA transfer, the host writes a DMA command block into memory. This block contains a pointer to the source of a transfer, a pointer to the destination of the transfer, and a count of the number of bytes to be transferred. The CPU writes the address of this command block to the DMA controller, then goes on with other work. The DMA controller proceeds to operate the memory bus directly, placing addresses on the bus to perform transfers without the help of the main CPU. A simple DMA controller is a standard component in PCs, and **bus-mastering** I/O boards for the PC usually contain their own high-speed DMA hardware.

Handshaking between the DMA controller and the device controller is performed via a pair of wires called DMA-request and DMA-acknowledge. The device controller places a signal on the DMA-request wire when a word of data is available for transfer. This signal causes the DMA controller to seize the memory bus, to place the desired address on the memory-address wires, and to place a signal on the DMA-acknowledge wire. When the device controller receives the DMA-acknowledge signal, it transfers the word of data to memory, and removes the DMA-request signal.

When the entire transfer is finished, the DMA controller interrupts the CPU. This process is depicted in Figure 13.5. When the DMA controller seizes the

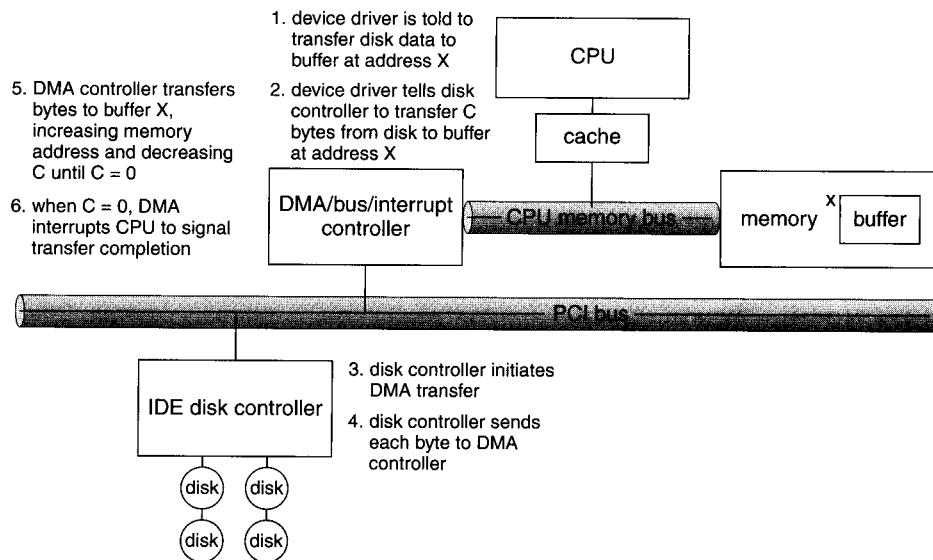


Figure 13.5 Steps in a DMA transfer.

memory bus, the CPU is momentarily prevented from accessing main memory, although it can still access data items in its primary and secondary cache. Although this **cycle stealing** can slow down the CPU computation, offloading the data-transfer work to a DMA controller generally improves the total system performance. Some computer architectures use physical memory addresses for DMA, but others perform **direct virtual-memory access (DVMA)**, using virtual addresses that undergo virtual- to physical-memory address translation. DVMA can perform a transfer between two memory-mapped devices without the intervention of the CPU or the use of main memory.

On protected-mode kernels, the operating system generally prevents processes from issuing device commands directly. This discipline protects data from access-control violations, and also protects the system from erroneous use of device controllers that could cause a system crash. Instead, the operating system exports functions that a sufficiently privileged process can use to access low-level operations on the underlying hardware. On kernels without memory protection, processes can access device controllers directly. This direct access can be used to obtain high performance, since it can avoid kernel communication, context switches, and layers of kernel software. Unfortunately, it interferes with system security and stability. The trend in general-purpose operating systems is to protect memory and devices, so that the system can try to guard against erroneous or malicious applications.

Although the hardware aspects of I/O are complex when considered at the level of detail of electronics-hardware designers, the concepts that we have just

described are sufficient to understand many I/O aspects of operating systems. Let's review the main concepts:

- A bus
- A controller
- An I/O port and its registers
- The handshaking relationship between the host and a device controller
- The execution of this handshaking in a polling loop or via interrupts
- The offloading of this work to a DMA controller for large transfers

We gave a basic example of the handshaking that takes place between a device controller and the host in Section 13.2. In reality, the wide variety of available devices poses a problem for operating-system implementers. Each kind of device has its own set of capabilities, control-bit definitions, and protocol for interacting with the host—and they are all different. How can the operating system be designed so that new devices can be attached to the computer without the operating system being rewritten? Also, when the devices vary so widely, how can the operating system give a convenient, uniform I/O interface to applications?

13.3 ■ Application I/O Interface

In this section, we discuss structuring techniques and interfaces for the operating system that enable I/O devices to be treated in a standard, uniform way. We explain, for instance, how an application can open a file on a disk without knowing what kind of disk it is, and how new disks and other devices can be added to a computer without the operating system being disrupted.

Like other complex software-engineering problems, the approach here involves abstraction, encapsulation, and software layering. Specifically, we can abstract away the detailed differences in I/O devices by identifying a few general kinds. Each general kind is accessed through a standardized set of functions—an **interface**. The differences are encapsulated in kernel modules called device drivers that internally are custom tailored to each device, but that export one of the standard interfaces. Figure 13.6 illustrates how the I/O-related portions of the kernel are structured in software layers.

The purpose of the device-driver layer is to hide the differences among device controllers from the I/O subsystem of the kernel, much as the I/O system calls encapsulate the behavior of devices in a few generic classes that hide hardware differences from applications. Making the I/O subsystem independent of the hardware simplifies the job of the operating-system developer. It also benefits the hardware manufacturers. They either design new devices to be

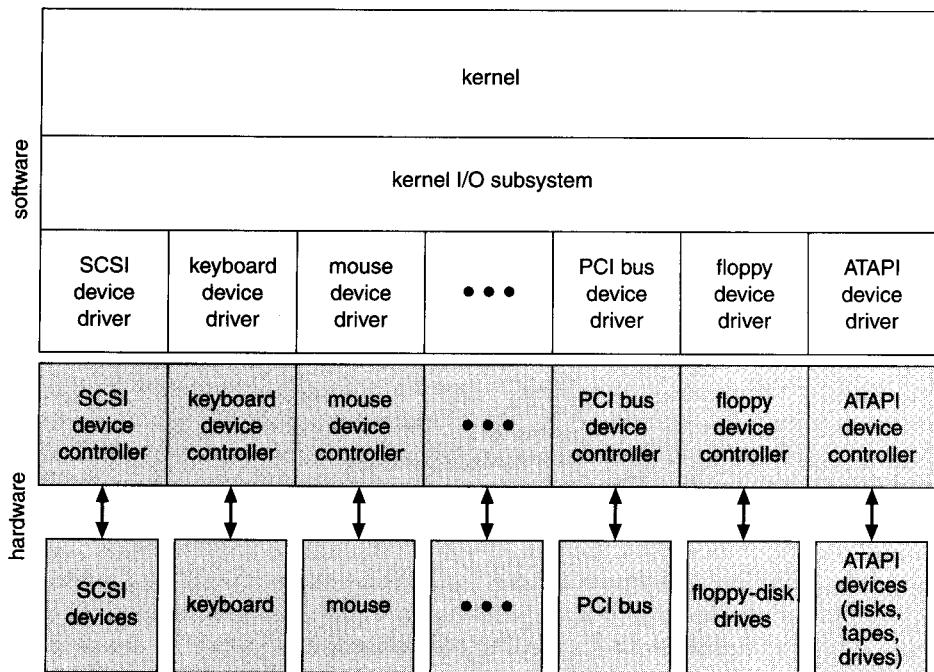


Figure 13.6 A kernel I/O structure.

compatible with an existing host controller interface (such as SCSI-2), or they write device drivers to interface the new hardware to popular operating systems. Thus, new peripherals can be attached to a computer without waiting for the operating-system vendor to develop support code.

Unfortunately for device-hardware manufacturers, each type of operating system has its own standards for the device-driver interface. A given device may ship with multiple device drivers—for instance, drivers for MS-DOS, Windows 95/98, Windows NT/2000, and Solaris. Devices vary in many dimensions, as illustrated in Figure 13.7.

- **Character-stream or block:** A character-stream device transfers bytes one by one, whereas a block device transfers a block of bytes as a unit.
- **Sequential or random-access:** A sequential device transfers data in a fixed order determined by the device, whereas the user of a random-access device can instruct the device to seek to any of the available data storage locations.
- **Synchronous or asynchronous:** A synchronous device is one that performs data transfers with predictable response times. An asynchronous device exhibits irregular or unpredictable response times.

aspect	variation	example
data-transfer mode	character block	terminal disk
access method	sequential random	modem CD-ROM
transfer schedule	synchronous asynchronous	tape keyboard
sharing	dedicated sharable	tape keyboard
device speed	latency seek time transfer rate delay between operations	
I/O direction	read only write only read-write	CD-ROM graphics controller disk

Figure 13.7 Characteristics of I/O devices.

- **Sharable or dedicated:** A sharable device can be used concurrently by several processes or threads; a dedicated device cannot.
- **Speed of operation:** Device speeds range from a few bytes per second to a few gigabytes per second.
- **Read-write, read only, or write only:** Some devices perform both input and output, but others support only one data direction.

For the purpose of application access, many of these differences are hidden by the operating system, and the devices are grouped into a few conventional types. The resulting styles of device access have been found to be useful and broadly applicable. Although the exact system calls may differ across operating systems, the device categories are fairly standard. The major access conventions include block I/O, character-stream I/O, memory-mapped file access, and network sockets. Operating systems also provide special system calls to access a few additional devices, such as a time-of-day clock and a timer. Some operating systems provide a set of system calls for graphical display, video, and audio devices.

Most operating systems also have an **escape** (or **back-door**) that transparently passes arbitrary commands from an application to a device driver. In UNIX, this system call is `ioctl()` (for I/O control). The `ioctl()` system call enables an application to access any functionality that can be implemented by any device driver, without the need to invent a new system call. The `ioctl()`

system call has three arguments. The first is a file descriptor that connects the application to the driver by referring to a hardware device managed by that driver. The second is an integer that selects one of the commands implemented in the driver. The third is a pointer to an arbitrary data structure in memory, thus enabling the application and driver to communicate any necessary control information or data.

13.3.1 Block and Character Devices

The **block-device** interface captures all the aspects necessary for accessing disk drives and other block-oriented devices. The expectation is that the device understands commands such as `read()` and `write()`, and, if it is a random-access device, it has a `seek()` command to specify which block to transfer next. Applications normally access such a device through a file-system interface. The operating system itself, and special applications such as database-management systems, may prefer to access a block device as a simple linear array of blocks. This mode of access is sometimes called **raw I/O**. We can see that `read()`, `write()`, and `seek()` capture the essential behaviors of block-storage devices, so that applications are insulated from the low-level differences among those devices.

Memory-mapped file access can be layered on top of block-device drivers. Rather than offering read and write operations, a memory-mapped interface provides access to disk storage via an array of bytes in main memory. The system call that maps a file into memory returns the virtual-memory address of an array of characters that contains a copy of the file. The actual data transfers are performed only when needed to satisfy access to the memory image. Because the transfers are handled by the same mechanism as that used for demand-paged virtual-memory access, memory-mapped I/O is efficient. Memory mapping is also convenient for programmers—access to a memory-mapped file is as simple as reading and writing to memory. Operating systems that offer virtual memory commonly use the mapping interface for kernel services. For instance, to execute a program, the operating system maps the executable into memory, and then transfers control to the entry address of the executable. The mapping interface is also commonly used for kernel access to swap space on disk.

A keyboard is an example of a device that is accessed through a **character-stream** interface. The basic system calls in this interface enable an application to `get()` or `put()` one character. On top of this interface, libraries can be built that offer line-at-a-time access, with buffering and editing services (for example, when a user types a backspace, the preceding character is removed from the input stream). This style of access is convenient for input devices such as keyboards, mice, and modems, which produce data for input “spontaneously”—that is, at times that cannot necessarily be predicted by the application. This

access style is also good for output devices such as printers or audio boards, which naturally fit the concept of a linear stream of bytes.

13.3.2 Network Devices

Because the performance and addressing characteristics of network I/O differ significantly from those of disk I/O, most operating systems provide a network I/O interface that is different from the `read()`-`write()`-`seek()` interface used for disks. One interface available in many operating systems, including UNIX and Windows NT, is the network **socket** interface.

Think of a wall socket for electricity: Any electrical appliance can be plugged in. By analogy, the system calls in the socket interface enable an application to create a socket, to connect a local socket to a remote address (which plugs this application into a socket created by another application), to listen for any remote application to plug into the local socket, and to send and receive packets over the connection. To support the implementation of servers, the socket interface also provides a function called `select()` that manages a set of sockets. A call to `select()` returns information about which sockets have a packet waiting to be received, and which sockets have room to accept a packet to be sent. The use of `select()` eliminates the polling and busy waiting that would otherwise be necessary for network I/O. These functions encapsulate the essential behaviors of networks, greatly facilitating the creation of distributed applications that can use any underlying network hardware and protocol stack.

Many other approaches to interprocess communication and network communication have been implemented. For instance, Windows NT provides one interface to the network interface card, and a second interface to the network protocols (Section 21.6). In UNIX, which has a long history as a proving ground for network technology, we find half-duplex pipes, full-duplex FIFOs, full-duplex STREAMS, message queues, and sockets. Information on UNIX networking is given in Section A.9.

13.3.3 Clocks and Timers

Most computers have hardware clocks and timers that provide three basic functions:

- Give the current time
- Give the elapsed time
- Set a timer to trigger operation X at time T

These functions are used heavily by the operating system, and also by time-sensitive applications. Unfortunately, the system calls that implement these functions are not standardized across operating systems.

The hardware to measure elapsed time and to trigger operations is called a **programmable interval timer**. It can be set to wait a certain amount of time and then to generate an interrupt. It can be set to do this operation once, or to repeat the process, to generate periodic interrupts. The scheduler uses this mechanism to generate an interrupt that will preempt a process at the end of its time slice. The disk I/O subsystem uses it to invoke the flushing of dirty cache buffers to disk periodically, and the network subsystem uses it to cancel operations that are proceeding too slowly because of network congestion or failures. The operating system may also provide an interface for user processes to use timers. The operating system can support more timer requests than the number of timer hardware channels by simulating virtual clocks. To do so, the kernel (or the timer device driver) maintains a list of interrupts wanted by its own routines and by user requests, sorted in earliest-time-first order. It sets the timer for the earliest time. When the timer interrupts, the kernel signals the requester, and reloads the timer with the next earliest time.

On many computers, the interrupt rate generated by the ticking of the hardware clock is between 18 and 60 ticks per second. This resolution is coarse, since a modern computer can execute hundreds of millions of instructions per second. The precision of triggers is limited by the coarse resolution of the timer, together with the overhead of maintaining virtual clocks. And, if the timer ticks are used to maintain the system time-of-day clock, the system clock can drift. In most computers, the hardware clock is constructed from a high-frequency counter. In some computers, the value of this counter can be read from a device register, in which case the counter can be considered to be a high-resolution clock. Although this clock does not generate interrupts, it offers accurate measurements of time intervals.

13.3.4 Blocking and Nonblocking I/O

Another aspect of the system-call interface relates to the choice between blocking I/O and nonblocking (or asynchronous) I/O. When an application issues a **blocking** system call, the execution of the application is suspended. The application is moved from the operating system's run queue to a wait queue. After the system call completes, the application is moved back to the run queue, where it is eligible to resume execution, at which time it will receive the values returned by the system call. The physical actions performed by I/O devices are generally asynchronous—they take a varying or unpredictable amount of time. Nevertheless, most operating systems use blocking system calls for the application interface, because blocking application code is easier to understand than nonblocking application code.

Some user-level processes need **nonblocking** I/O. One example is a user interface that receives keyboard and mouse input while processing and displaying data on the screen. Another example is a video application that reads

frames from a file on disk while simultaneously decompressing and displaying the output on the display.

One way that an application writer can overlap execution with I/O is to write a multithreaded application. Some threads can perform blocking system calls, while others continue executing. The Solaris developers used this technique to implement a user-level library for asynchronous I/O, freeing the application writer from that task. Some operating systems provide nonblocking I/O system calls. A nonblocking call does not halt the execution of the application for an extended time. Instead, it returns quickly, with a return value that indicates how many bytes were transferred.

An alternative to a nonblocking system call is an asynchronous system call. An asynchronous call returns immediately, without waiting for the I/O to complete. The application continues to execute its code. The completion of the I/O at some future time is communicated to the application, either through the setting of some variable in the address space of the application, or through the triggering of a signal or software interrupt or a call-back routine that is executed outside the linear control flow of the application. The difference between nonblocking and asynchronous system calls is that a nonblocking `read()` returns immediately with whatever data are available—the full number of bytes requested, fewer, or none at all. An asynchronous `read()` call requests a transfer that will be performed in its entirety, but that will complete at some future time.

A good example of nonblocking behavior is the `select()` system call for network sockets. This system call takes an argument that specifies a maximum waiting time. By setting it to 0, an application can poll for network activity without blocking. But using `select()` introduces extra overhead, because the `select()` call only checks whether I/O is possible. For a data transfer, `select()` must be followed by some kind of `read()` or `write()` command. A variation of this approach, found in Mach, is a blocking multiple-read call. It specifies desired reads for several devices in one system call, and returns as soon as any one of them completes.

13.4 ■ Kernel I/O Subsystem

Kernels provide many services related to I/O. Several services—scheduling, buffering, caching, spooling, device reservation, and error handling—are provided by the kernel's I/O subsystem and build on the hardware and device-driver infrastructure.

13.4.1 I/O Scheduling

To schedule a set of I/O requests means to determine a good order in which to execute them. The order in which applications issue system calls rarely is the best choice. Scheduling can improve overall system performance, can share

device access fairly among processes, and can reduce the average waiting time for I/O to complete. Here is a simple example to illustrate the opportunity. Suppose that a disk arm is near the beginning of a disk, and that three applications issue blocking read calls to that disk. Application 1 requests a block near the end of the disk, application 2 requests one near the beginning, and application 3 requests one in the middle of the disk. The operating system can reduce the distance that the disk arm travels by serving the applications in order 2, 3, 1. Rearranging the order of service in this way is the essence of I/O scheduling.

Operating-system developers implement scheduling by maintaining a queue of requests for each device. When an application issues a blocking I/O system call, the request is placed on the queue for that device. The I/O scheduler rearranges the order of the queue to improve the overall system efficiency and the average response time experienced by applications. The operating system may also try to be fair, so that no one application receives especially poor service, or it may give priority service for delay-sensitive requests. For instance, requests from the virtual-memory subsystem may take priority over application requests. Several scheduling algorithms for disk I/O are detailed in Section 14.2.

One way that the I/O subsystem improves the efficiency of the computer is by scheduling I/O operations. Another way is by using storage space in main memory or on disk, via techniques called buffering, caching, and spooling.

13.4.2 Buffering

A **buffer** is a memory area that stores data while they are transferred between two devices or between a device and an application. Buffering is done for three reasons. One reason is to cope with a speed mismatch between the producer and consumer of a data stream. Suppose, for example, that a file is being received via modem for storage on the hard disk. The modem is about a thousand times slower than the hard disk. So a buffer is created in main memory to accumulate the bytes received from the modem. When an entire buffer of data has arrived, the buffer can be written to disk in a single operation. Since the disk write is not instantaneous and the modem still needs a place to store additional incoming data, two buffers are used. After the modem fills the first buffer, the disk write is requested. The modem then starts to fill the second buffer while the first buffer is written to disk. By the time the modem has filled the second buffer, the disk write from the first one should have completed, so the modem can switch back to the first buffer while the disk writes the second one. This **double buffering** decouples the producer of data from the consumer, thus relaxing timing requirements between them. The need for this decoupling is illustrated in Figure 13.8, which lists the enormous differences in device speeds for typical computer hardware.

A second use of buffering is to adapt between devices that have different data-transfer sizes. Such disparities are especially common in computer

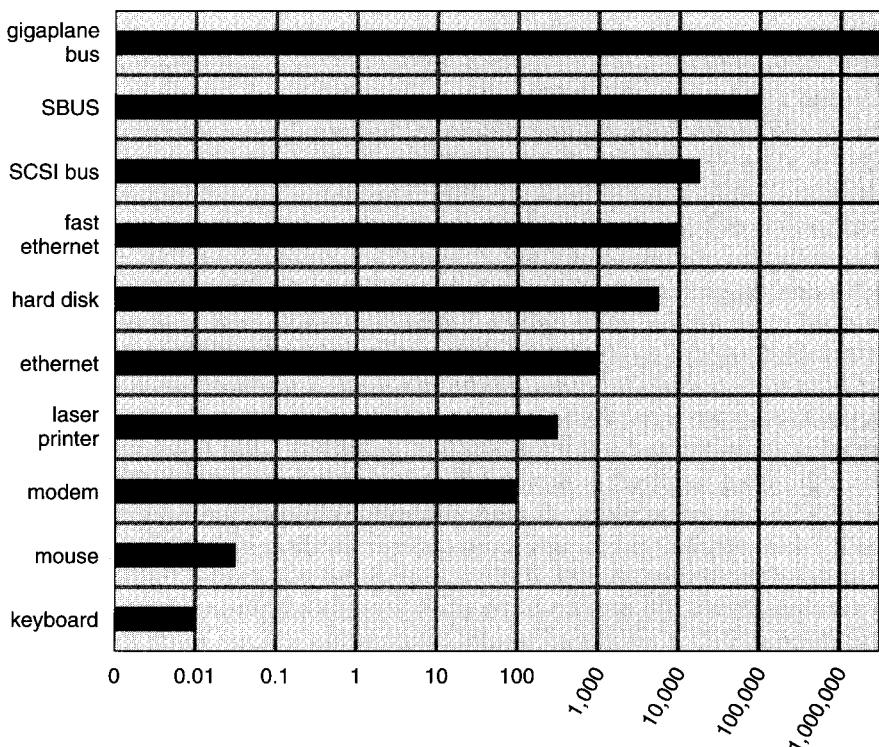


Figure 13.8 Sun Enterprise 6000 device-transfer rates (logarithmic).

networking, where buffers are used widely for fragmentation and reassembly of messages. At the sending side, a large message is fragmented into small network packets. The packets are sent over the network, and the receiving side places them in a reassembly buffer to form an image of the source data.

A third use of buffering is to support copy semantics for application I/O. An example will clarify the meaning of “copy semantics.” Suppose that an application has a buffer of data that it wishes to write to disk. It calls the `write()` system call, providing a pointer to the buffer and an integer specifying the number of bytes to write. After the system call returns, what happens if the application changes the contents of the buffer? With **copy semantics**, the version of the data written to disk is guaranteed to be the version at the time of the application system call, independent of any subsequent changes in the application’s buffer. A simple way that the operating system can guarantee copy semantics is for the `write()` system call to copy the application data into a kernel buffer before returning control to the application. The disk write is performed from the kernel buffer, so that subsequent changes to the application buffer have no effect. Copying of data between kernel buffers and application

data space is common in operating systems, despite the overhead that this operation introduces, because of the clean semantics. The same effect can be obtained more efficiently by clever use of virtual-memory mapping and copy-on-write page protection.

13.4.3 Caching

A **cache** is a region of fast memory that holds copies of data. Access to the cached copy is more efficient than access to the original. For instance, the instructions of the currently running process are stored on disk, cached in physical memory, and copied again in the CPU's secondary and primary caches. The difference between a buffer and a cache is that a buffer may hold the only existing copy of a data item, whereas a cache, by definition, just holds a copy on faster storage of an item that resides elsewhere.

Caching and buffering are distinct functions, but sometimes a region of memory can be used for both purposes. For instance, to preserve copy semantics and to enable efficient scheduling of disk I/O, the operating system uses buffers in main memory to hold disk data. These buffers are also used as a cache, to improve the I/O efficiency for files that are shared by applications or that are being written and reread rapidly. When the kernel receives a file I/O request, the kernel first accesses the buffer cache to see whether that region of the file is already available in main memory. If so, a physical disk I/O can be avoided or deferred. Also, disk writes are accumulated in the buffer cache for several seconds, so that large transfers are gathered to allow efficient write schedules. This strategy of delaying writes to improve I/O efficiency is discussed, in the context of remote file access, in Section 16.3.

13.4.4 Spooling and Device Reservation

A **spool** is a buffer that holds output for a device, such as a printer, that cannot accept interleaved data streams. Although a printer can serve only one job at a time, several applications may wish to print their output concurrently, without having their output mixed together. The operating system solves this problem by intercepting all output to the printer. Each application's output is spooled to a separate disk file. When an application finishes printing, the spooling system queues the corresponding spool file for output to the printer. The spooling system copies the queued spool files to the printer one at a time. In some operating systems, spooling is managed by a system daemon process. In other operating systems, it is handled by an in-kernel thread. In either case, the operating system provides a control interface that enables users and system administrators to display the queue, to remove unwanted jobs before those jobs print, to suspend printing while the printer is serviced, and so on.

Some devices, such as tape drives and printers, cannot usefully multiplex the I/O requests of multiple concurrent applications. Spooling is one way that

operating systems can coordinate concurrent output. Another way to deal with concurrent device access is to provide explicit facilities for coordination. Some operating systems (including VMS) provide support for exclusive device access, by enabling a process to allocate an idle device, and to deallocate that device when it is no longer needed. Other operating systems enforce a limit of one open file handle to such a device. Many operating systems provide functions that enable processes to coordinate exclusive access among themselves. For instance, Windows NT provides system calls to wait until a device object becomes available. It also has a parameter to the `open()` system call that declares the types of access to be permitted to other concurrent threads. On these systems, it is up to the applications to avoid deadlock.

13.4.5 Error Handling

An operating system that uses protected memory can guard against many kinds of hardware and application errors, so that a complete system failure is not the usual result of each minor mechanical glitch. Devices and I/O transfers can fail in many ways, either for transient reasons, such as a network becoming overloaded, or for “permanent” reasons, such as a disk controller becoming defective. Operating systems can often compensate effectively for transient failures. For instance, a disk `read()` failure results in a `read()` retry, and a network `send()` error results in a `resend()`, if the protocol so specifies. Unfortunately, if an important component experiences a permanent failure, the operating system is unlikely to recover.

As a general rule, an I/O system call will return 1 bit of information about the status of the call, signifying either success or failure. In the UNIX operating system, an additional integer variable named `errno` is used to return an error code—one of about 100 values—indicating the general nature of the failure (for example, argument out of range, bad pointer, or file not open). By contrast, some hardware can provide highly detailed error information, although many current operating systems are not designed to convey this information to the application. For instance, a failure of a SCSI device is reported by the SCSI protocol in terms of a **sense key** that identifies the general nature of the failure, such as a hardware error or an illegal request; an **additional sense code** that states the category of failure, such as a bad command parameter or a self-test failure; and an **additional sense-code qualifier** that gives even more detail, such as which command parameter was in error, or which hardware subsystem failed its self-test. Further, many SCSI devices maintain internal pages of error-log information that can be requested by the host, but that seldom are.

13.4.6 Kernel Data Structures

The kernel needs to keep state information about the use of I/O components. It does so through a variety of in-kernel data structures, such as the open-file

table structure from Section 12.1. The kernel uses many similar structures to track network connections, character-device communications, and other I/O activities.

UNIX provides file-system access to a variety of entities, such as user files, raw devices, and the address spaces of processes. Although each of these entities supports a `read()` operation, the semantics differ. For instance, to read a user file, the kernel needs to probe the buffer cache before deciding whether to perform a disk I/O. To read a raw disk, the kernel needs to ensure that the request size is a multiple of the disk sector size, and is aligned on a sector boundary. To read a process image, it is merely necessary to copy data from memory. UNIX encapsulates these differences within a uniform structure by using an object-oriented technique. The open-file record, shown in Figure 13.9, contains a dispatch table that holds pointers to the appropriate routines, depending on the type of file.

Some operating systems use object-oriented methods even more extensively. For instance, Windows NT uses a message-passing implementation for I/O. An I/O request is converted into a message that is sent through the kernel to the I/O manager and then to the device driver, each of which may change the message contents. For output, the message contains the data to be written. For input, the message contains a buffer to receive the data. The message-passing approach can add overhead, by comparison with procedural techniques that use shared data structures, but it simplifies the structure and design of the I/O system, and adds flexibility.

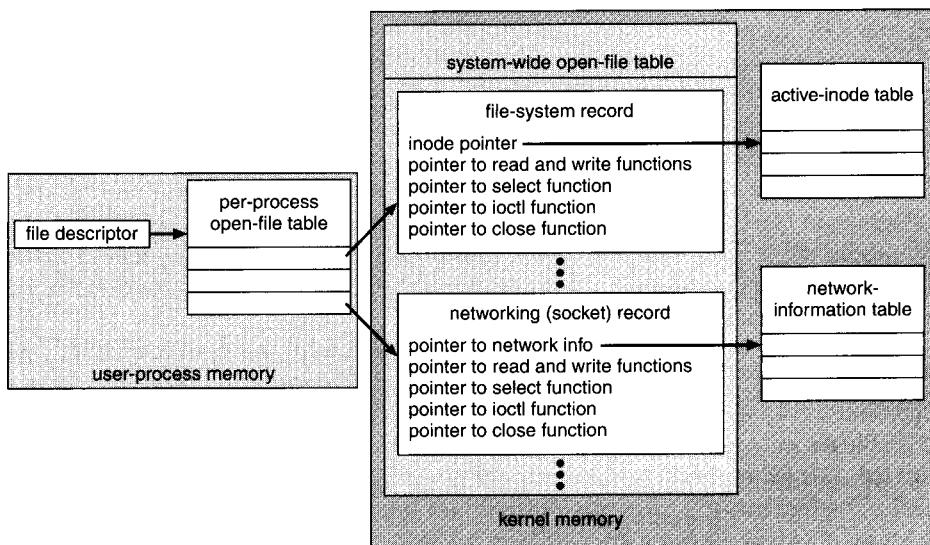


Figure 13.9 UNIX I/O kernel structure.

In summary, the I/O subsystem coordinates an extensive collection of services that are available to applications and to other parts of the kernel. The I/O subsystem supervises

- The management of the name space for files and devices
- Access control to files and devices
- Operation control (for example, a modem cannot `seek()`)
- File system space allocation
- Device allocation
- Buffering, caching, and spooling
- I/O scheduling
- Device status monitoring, error handling, and failure recovery
- Device driver configuration and initialization

The upper levels of the I/O subsystem access devices via the uniform interface provided by the device drivers.

13.5 ■ Transforming I/O to Hardware Operations

Earlier, we described the handshaking between a device driver and a device controller, but we did not explain how the operating system connects an application request to a set of network wires or to a specific disk sector. Let us consider the example of reading a file from disk. The application refers to the data by a file name. Within a disk, the file system maps from the file name through the file-system directories to obtain the space allocation of the file. For instance, in MS-DOS, the name maps to a number that indicates an entry in the file-access table, and that table entry tells which disk blocks are allocated to the file. In UNIX, the name maps to an inode number, and the corresponding inode contains the space-allocation information.

How is the connection made from the file name to the disk controller (the hardware port address or the memory-mapped controller registers)? First, we consider MS-DOS, a relatively simple operating system. The first part of an MS-DOS file name, preceding the colon, is a string that identifies a specific hardware device. For example, `c:` is the first part of every file name on the primary hard disk. The fact that `c:` represents the primary hard disk is built into the operating system; `c:` is mapped to a specific port address through a device table. Because of the colon separator, the device name space is separate from the file-system name space within each device. This separation makes it easy for the operating

system to associate extra functionality with each device. For instance, it is easy to invoke spooling on any files written to the printer.

If, instead, the device name space is incorporated in the regular file-system name space, as it is in UNIX, the normal file-system name services are provided automatically. If the file system provides ownership and access control to all file names, then devices have owners and access control. Since files are stored on devices, such an interface provides access to the I/O system at two levels. Names can be used to access the devices themselves, or to access the files stored on the devices.

UNIX represents device names in the regular file-system name space. Unlike an MS-DOS file name, which has the colon separator, a UNIX path name has no clear separation of the device portion. In fact, no part of the path name is the name of a device. UNIX has a **mount table** that associates prefixes of path names with specific device names. To resolve a path name, UNIX looks up the name in the mount table to find the longest matching prefix; the corresponding entry in the mount table gives the device name. This device name also has the form of a name in the file-system name space. When UNIX looks up this name in the file-system directory structures, instead of finding an inode number, UNIX finds a *<major, minor>* device number. The major device number identifies a device driver that should be called to handle I/O to this device. The minor device number is passed to the device driver to index into a device table. The corresponding device-table entry gives the port address or the memory-mapped address of the device controller.

Modern operating systems obtain significant flexibility from the multiple stages of lookup tables in the path between a request and a physical device controller. The mechanisms that pass requests between applications and drivers are general. Thus, we can introduce new devices and drivers into a computer without recompiling the kernel. In fact, some operating systems have the ability to load device drivers on demand. At boot time, the system first probes the hardware buses to determine what devices are present, and then the system loads in the necessary drivers, either immediately, or when first required by an I/O request.

Now we describe the typical lifecycle of a blocking read request, as depicted in Figure 13.10. The figure suggests that an I/O operation requires a great many steps that together consume a tremendous number of CPU cycles.

1. A process issues a blocking `read()` system call to a file descriptor of a file that has been *opened* previously.
2. The system-call code in the kernel checks the parameters for correctness. In the case of input, if the data are already available in the buffer cache, the data are returned to the process and the I/O request is completed.
3. Otherwise, a physical I/O needs to be performed, so the process is removed from the run queue and is placed on the wait queue for the device, and the

I/O request is scheduled. Eventually, the I/O subsystem sends the request to the device driver. Depending on the operating system, the request is sent via a subroutine call or via an in-kernel message.

- The device driver allocates kernel buffer space to receive the data, and schedules the I/O. Eventually, the driver sends commands to the device controller by writing into the device control registers.

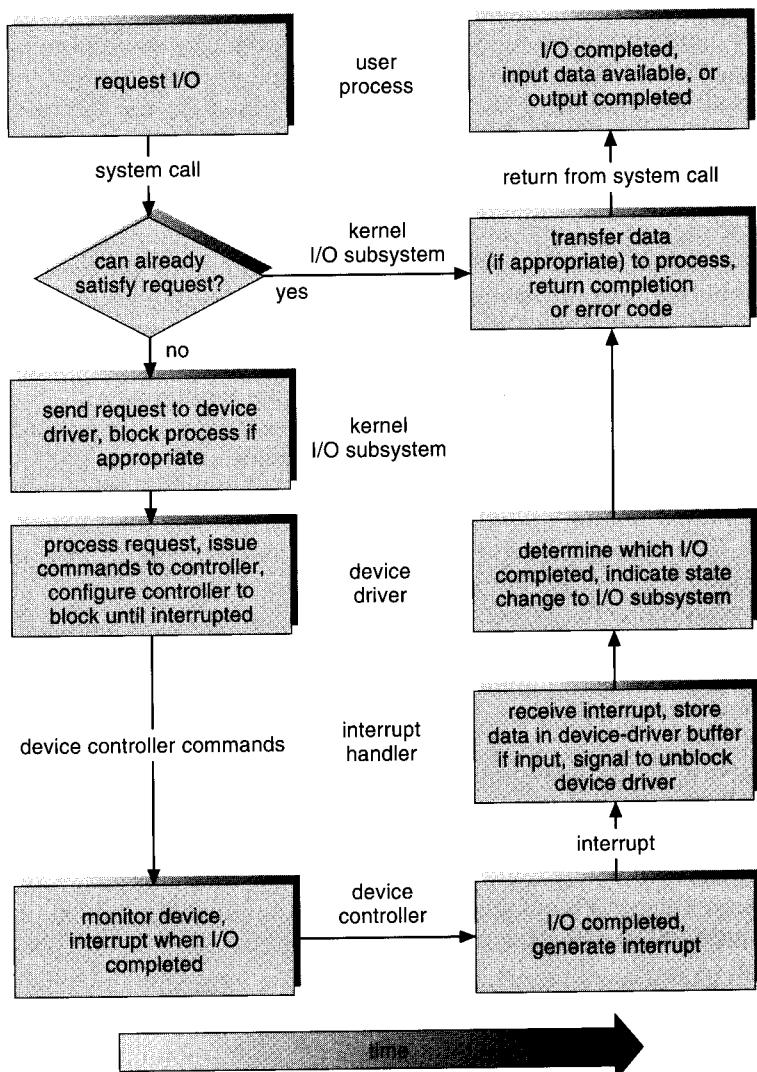


Figure 13.10 The life cycle of an I/O request.

5. The device controller operates the device hardware to perform the data transfer.
6. The driver may poll for status and data, or it may have set up a DMA transfer into kernel memory. We assume that the transfer is managed by a DMA controller, which generates an interrupt when the transfer completes.
7. The correct interrupt handler receives the interrupt via the interrupt-vector table, stores any necessary data, signals the device driver, and returns from the interrupt.
8. The device driver receives the signal, determines which I/O request completed, determines the request's status, and signals the kernel I/O subsystem that the request has been completed.
9. The kernel transfers data or return codes to the address space of the requesting process, and moves the process from the wait queue back to the ready queue.
10. Moving the process to the ready queue unblocks the process. When the scheduler assigns the process to the CPU, the process resumes execution at the completion of the system call.

13.6 ■ STREAMS

UNIX System V has an interesting mechanism, called **STREAMS**, that enables an application to assemble pipelines of driver code dynamically. A stream is a full-duplex connection between a device driver and a user-level process. It consists of a **stream head** that interfaces with the user process, a **driver end** that controls the device, and zero or more **stream modules** between them. The stream head, the driver end, and each module contain a pair of queues—a read queue and a write queue. Message passing is used to transfer data between queues. The STREAMS structure is shown in Figure 13.11.

Modules provide the functionality of STREAMS processing and they are *pushed* onto a stream using the `ioctl()` system call. For example, a process can open a serial-port device via a stream, and can push on a module to handle input editing. Because messages are exchanged between queues in adjacent modules, a queue in one module may overflow an adjacent queue. To prevent this from occurring, a queue may support **flow control**. Without flow control, a queue accepts all messages and immediately sends them on to the queue in the adjacent module without buffering them. A queue supporting flow control buffers messages and does not accept messages without sufficient buffer space. Flow control is supported by exchanging control messages between queues in adjacent modules.

A user process writes data to a device using either the `write()` or `putmsg()` system calls. The `write()` system call writes raw data to the stream

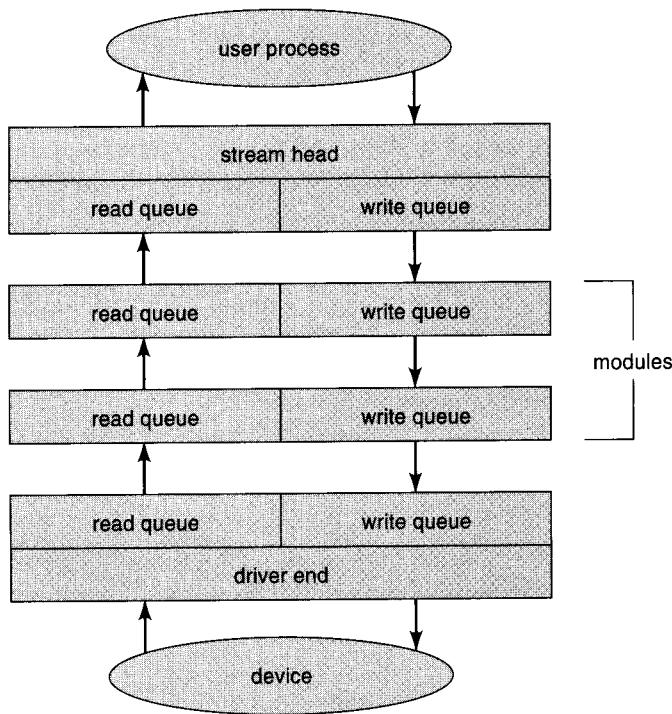


Figure 13.11 The STREAMS structure.

whereas `putmsg()` allows the user process to specify a message. Regardless of the system call used by the user process, the stream head copies the data into a message and delivers it to the queue for the next module in line. This copying of messages continues until the message is copied to the driver end and hence the device. Similarly, the user process reads data from the stream head using either the `read()` or `getmsg()` system calls. If `read()` is used, the stream head gets a message from its adjacent queue and returns ordinary data (an unstructured byte stream) to the process. If `getmsg()` is used, a message is returned to the process.

STREAMS I/O is asynchronous (or non-blocking) with the exception of when the user process communicates with the stream head. When writing to the stream, the user process will block, assuming the next queue uses flow control, until there is room to copy the message. Likewise, the user process will block when reading from the stream until data is available.

The driver end is similar to a stream head or a module in that it has a read and write queue. However, the driver end must respond to interrupts such as one triggered when a frame is ready to be read from a network. Unlike the stream head that may block if it is unable to copy a message to the next queue in line, the driver end must handle all incoming data. Drivers must support flow

control as well. However, if a device's buffer is full, a device typically resorts to dropping incoming messages. Consider a network card whose input buffer is full. The network card must simply drop further messages until there is ample buffer space to store incoming messages.

The benefit of using STREAMS is that it provides a framework to a modular and incremental approach to writing device drivers and network protocols.

Modules may be used by different STREAMS and hence by different devices. For example, a networking module may be used by both an Ethernet network card and a token ring network card. Furthermore, rather than treating character device I/O as an unstructured byte stream, STREAMS allow support for message boundaries and control information between modules. Support for STREAMS is widespread among most UNIX variants and it is the preferred method for writing protocols and device drivers. For example, in System V UNIX and Solaris, the socket mechanism is implemented using STREAMS.

13.7 ■ Performance

I/O is a major factor in system performance. It places heavy demands on the CPU to execute device-driver code and to schedule processes fairly and efficiently as they block and unblock. The resulting context switches stress the CPU and its hardware caches. I/O also exposes any inefficiencies in the interrupt-handling mechanisms in the kernel, and I/O loads down the memory bus during data copy between controllers and physical memory, and again during copies between kernel buffers and application data space. Coping gracefully with all these demands is one of the major concerns of a computer architect.

Although modern computers can handle many thousands of interrupts per second, interrupt handling is a relatively expensive task: Each interrupt causes the system to perform a state change, to execute the interrupt handler, and then to restore state. Programmed I/O can be more efficient than interrupt-driven I/O, if the number of cycles spent busy-waiting is not excessive. An I/O completion typically unblocks a process, leading to the full overhead of a context switch.

Network traffic can also cause a high context-switch rate. Consider, for instance, a remote login from one machine to another. Each character typed on the local machine must be transported to the remote machine. On the local machine, the character is typed; a keyboard interrupt is generated; and the character is passed through the interrupt handler to the device driver, to the kernel, and then to the user process. The user process issues a network I/O system call to send the character to the remote machine. The character then flows into the local kernel, through the network layers that construct a network packet, and into the network device driver. The network device driver transfers the packet to the network controller, which sends the character and generates

an interrupt. The interrupt is passed back up through the kernel to cause the network I/O system call to complete.

Now, the remote system's network hardware receives the packet, and an interrupt is generated. The character is unpacked from the network protocols and is given to the appropriate network daemon. The network daemon identifies which remote login session is involved, and passes the packet to the appropriate subdaemon for that session. Throughout this flow there are context switches and state switches (Figure 13.12). Usually, the receiver echoes the character back to the sender; that approach doubles the work.

The Solaris developers reimplemented the **telnet** daemon using in-kernel threads to eliminate the context switches involved in moving each character between daemons and the kernel. Sun estimates that this improvement

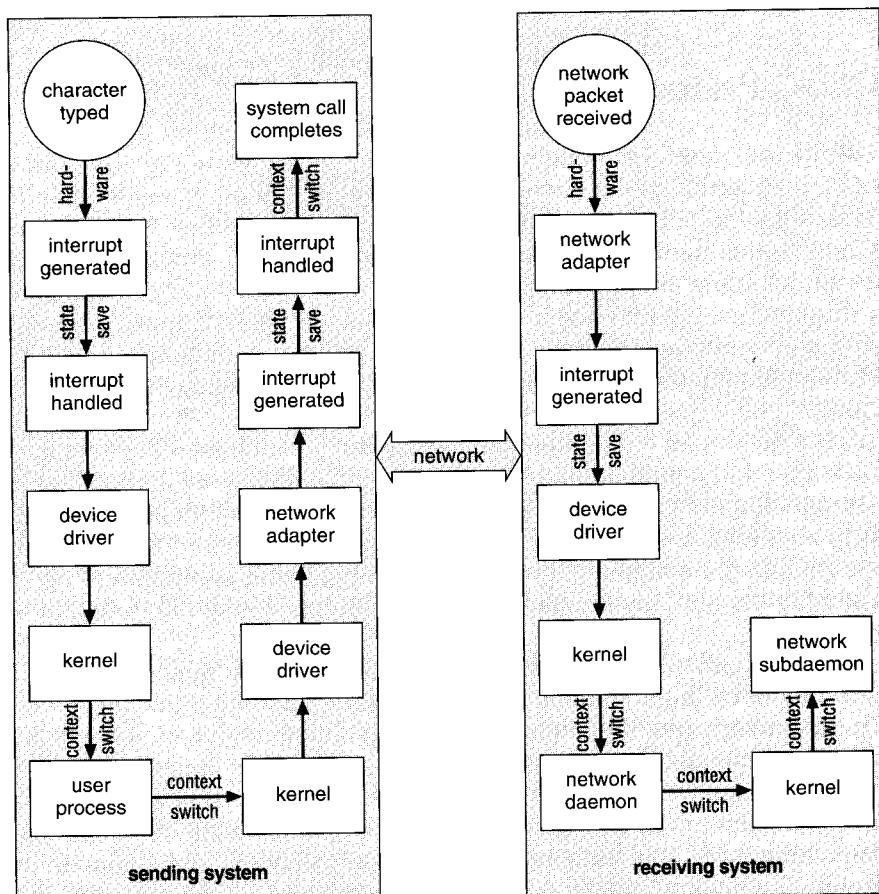


Figure 13.12 Intercomputer communications.

increased the maximum number of network logins from a few hundred to a few thousand on a large server.

Other systems use separate **front-end processors** for terminal I/O, to reduce the interrupt burden on the main CPU. For instance, a **terminal concentrator** can multiplex the traffic from hundreds of remote terminals into one port on a large computer. An **I/O channel** is a dedicated, special-purpose CPU found in mainframes and in other high-end systems. The job of a channel is to offload I/O work from the main CPU. The idea is that the channels keep the data flowing smoothly, while the main CPU remains free to process the data. Like the device controllers and DMA controllers found in smaller computers, a channel can process more general and sophisticated programs, so channels can be tuned for particular workloads.

We can employ several principles to improve the efficiency of I/O:

- Reduce the number of context switches.
- Reduce the number of times that data must be copied in memory while passing between device and application.
- Reduce the frequency of interrupts by using large transfers, smart controllers, and polling (if busy-waiting can be minimized).
- Increase concurrency by using DMA-knowledgeable controllers or channels to offload simple data copying from the CPU.
- Move processing primitives into hardware, to allow their operation in device controllers concurrent with the CPU and bus operation.
- Balance CPU, memory subsystem, bus, and I/O performance, because an overload in any one area will cause idleness in others.

Devices vary greatly in complexity. For instance, a mouse is simple. The mouse movements and button clicks are converted into numeric values that are passed from hardware, through the mouse device driver, to the application. By contrast, the functionality provided by the NT disk device driver is complex. It not only manages individual disks but also implements RAID arrays (Section 14.5). To do so, it converts an application's read or write request into a coordinated set of disk I/O operations. Moreover, it implements sophisticated error-handling and data-recovery algorithms, and takes many steps to optimize disk performance, because of the importance of secondary-storage performance to overall system performance.

Where should the I/O functionality be implemented—in the device hardware, in the device driver, or in application software? Sometimes we observe the progression depicted in Figure 13.13.

- Initially, we implement experimental I/O algorithms at the application level, because application code is flexible, and application bugs are unlikely

to cause system crashes. Furthermore, by developing code at the application level, we avoid the need to reboot or reload device drivers after every change to the code. An application-level implementation can be inefficient, however, because of the overhead of context switches, and because the application cannot take advantage of internal kernel data structures and kernel functionality (such as efficient in-kernel messaging, threading, and locking).

- When an application-level algorithm has demonstrated its worth, we may reimplement it in the kernel. This can improve the performance, but the development effort is more challenging, because an operating-system kernel is a large, complex software system. Moreover, an in-kernel implementation must be thoroughly debugged to avoid data corruption and system crashes.
- The highest performance may be obtained by a specialized implementation in hardware, either in the device or in the controller. The disadvantages of a hardware implementation include the difficulty and expense of making further improvements or of fixing bugs, the increased development time (months rather than days), and the decreased flexibility. For instance, a hardware RAID controller may not provide any means for the kernel to influence the order or location of individual block reads and writes, even if the kernel has special information about the workload that would enable the kernel to improve the I/O performance.

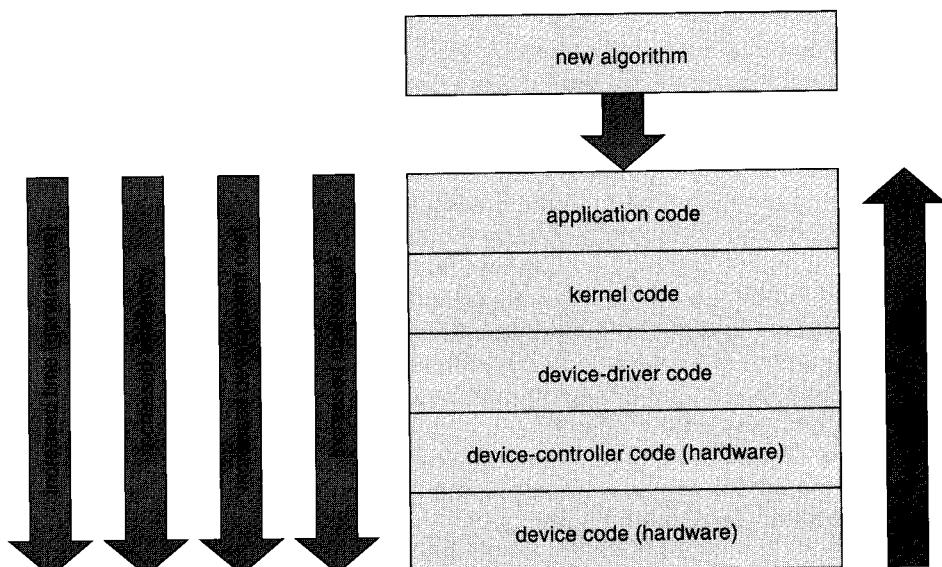


Figure 13.13 Device-functionality progression.

13.8 ■ Summary

The basic hardware elements involved in I/O are buses, device controllers, and the devices themselves. The work of moving data between devices and main memory is performed by the CPU as programmed I/O, or is offloaded to a DMA controller. The kernel module that controls a device is a device driver. The system-call interface provided to applications is designed to handle several basic categories of hardware, including block devices, character devices, memory-mapped files, network sockets, and programmed interval timers. The system calls usually block the process that issues them, but nonblocking and asynchronous calls are used by the kernel itself, and by applications that must not sleep while waiting for an I/O operation to complete.

The kernel's I/O subsystem provides numerous services. Among these are I/O scheduling, buffering, spooling, error handling, and device reservation. Another service is name translation, to make the connection between hardware devices and the symbolic file names used by applications. It involves several levels of mapping that translate from a character string name to a specific device driver and device address, and then to physical addresses of I/O ports or bus controllers. This mapping may occur within the file-system name space, as it does in UNIX, or in a separate device-name space, as it does in MS-DOS.

STREAMS is an implementation and methodology for making drivers reusable and easy to use. Through them, drivers can be stacked, with data passed through them sequentially and bidirectionally for processing.

I/O system calls are costly in terms of CPU consumption, because of the many layers of software between a physical device and the application. These layers imply the overheads of context switching to cross the kernel's protection boundary, of signal and interrupt handling to service the I/O devices, and of the load on the CPU and memory system to copy data between kernel buffers and application space.

■ Exercises

13.1 State three advantages of placing functionality in a device controller, rather than in the kernel. State three disadvantages.

13.2 Consider the following I/O scenarios on a single-user PC.

- a. A mouse used with a graphical user interface
- b. A tape drive on a multitasking operating system (assume no device preallocation is available)
- c. A disk drive containing user files
- d. A graphics card with direct bus connection, accessible through memory-mapped I/O

For each of these I/O scenarios, would you design the operating system to use buffering, spooling, caching, or a combination? Would you use polled I/O or interrupt-driven I/O? Give reasons for your choices.

- 13.3 The example of handshaking in Section 13.2 used 2 bits: a busy bit and a command-ready bit. Is it possible to implement this handshaking with only 1 bit? If it is, describe the protocol. If not, explain why 1 bit is insufficient.
- 13.4 Describe three circumstances under which blocking I/O should be used. Describe three circumstances under which nonblocking I/O should be used. Why not just implement nonblocking I/O and have processes busy-wait until their device is ready?
- 13.5 Why might a system use interrupt-driven I/O to manage a single serial port, but polling I/O to manage a front-end processor, such as a terminal concentrator?
- 13.6 Polling for an I/O completion can waste a large number of CPU cycles if the processor iterates a busy-waiting loop many times before the I/O completes. But if the I/O device is ready for service, polling can be much more efficient than is catching and dispatching an interrupt. Describe a hybrid strategy that combines polling, sleeping, and interrupts for I/O device service. For each of these three strategies (pure polling, pure interrupts, hybrid), describe a computing environment in which that strategy is more efficient than is either of the others.
- 13.7 UNIX coordinates the activities of the kernel I/O components by manipulating shared in-kernel data structures, whereas Windows NT uses object-oriented message passing between kernel I/O components. Discuss three pros and three cons of each approach.
- 13.8 How does DMA increase system concurrency? How does it complicate the hardware design?
- 13.9 Write (in pseudocode) an implementation of virtual clocks, including the queueing and management of timer requests for the kernel and applications. Assume that the hardware provides three timer channels.
- 13.10 Why is it important to scale up system bus and device speeds as the CPU speed increases?
- 13.11 Distinguish between a STREAMS driver and a STREAMS module.

Bibliographical Notes

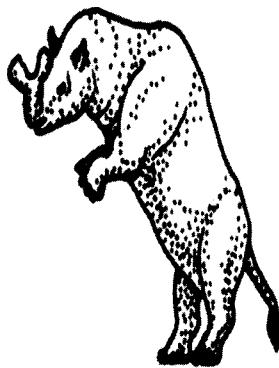
Vahalia [1996] provides a good overview of I/O and networking in UNIX. Lefler et al. [1989] detail the I/O structures and methods employed in BSD UNIX.

Milenkovic [1987] discusses the complexity of I/O methods and implementation. The use and programming of the various interprocess-communication and network protocols in UNIX is explored in Stevens [1992]. Brain [1996] documents the Windows NT application interface. The I/O implementation in the sample MINIX OS is described in Tanenbaum and Woodhulll [1997]. Custer [1994] includes detailed information on the NT message-passing implementation of I/O.

For details of hardware-level I/O handling and memory-mapping functionality, processor reference manuals (Motorola [1993] and Intel [1993]), are among the best sources. Hennessy and Patterson [1996] describe multiprocessor systems and cache-consistency issues. Tanenbaum [1990] describes hardware I/O design at a low level, and Sargent and Shoemaker [1995] provide a programmer's guide to low-level PC hardware and software. The IBM PC device I/O address map is given in IBM [1983]. An issue of the March 1994 *IEEE Computer* is devoted to advanced I/O hardware and software. Rago [1993] provides a good discussion on STREAMS.

Chapter 14

MASS- STORAGE STRUCTURE



The file system can be viewed logically as consisting of three parts. In Chapter 11, we saw the user and programmer interface to the file system. In Chapter 12, we described the internal data structures and algorithms used by the operating system to implement this interface. In this chapter, we discuss the lowest level of the file system: the secondary and tertiary storage structures. We first describe disk-scheduling algorithms that schedule the order of disk I/Os to improve performance. Next, we discuss disk formatting and management of boot blocks, damaged blocks, and swap space. We examine secondary storage structure, covering disk reliability and stable-storage implementation. We conclude with a brief description of tertiary storage devices, and the problems that arise when an operating system uses tertiary storage.

14.1 ■ Disk Structure

Disks provide the bulk of secondary storage for modern computer systems. Magnetic tape was used as an early secondary-storage medium, but the access time is much slower than for disks. Thus, tapes are currently used mainly for backup, for storage of infrequently used information, as a medium for transferring information from one system to another, and for storing quantities of data so large that they are impractical as disk systems. Tape storage is described in Section 14.8.

Modern disk drives are addressed as large one-dimensional arrays of **logical blocks**, where the logical block is the smallest unit of transfer. The size

of a logical block is usually 512 bytes, although some disks can be **low-level formatted** to choose a different logical block size, such as 1,024 bytes. This option is described in Section 14.3.1.

The one-dimensional array of logical blocks is mapped onto the sectors of the disk sequentially. Sector 0 is the first sector of the first track on the outermost cylinder. The mapping proceeds in order through that track, then through the rest of the tracks in that cylinder, and then through the rest of the cylinders from outermost to innermost.

By using this mapping, we can—at least in theory—convert a logical block number into an old-style disk address that consists of a cylinder number, a track number within that cylinder, and a sector number within that track. In practice, it is difficult to perform this translation, for two reasons. First, most disks have some defective sectors, but the mapping hides this by substituting spare sectors from elsewhere on the disk. Second, the number of sectors per track is not a constant on some drives. On media that use **constant linear velocity (CLV)**, the density of bits per track is uniform. The farther a track is from the center of the disk, the greater its length, so the more sectors it can hold. As we move from outer zones to inner zones, the number of sectors per track decreases. Tracks in the outermost zone typically hold 40 percent more sectors than do tracks in the innermost zone. The drive increases its rotation speed as the head moves from the outer to the inner tracks to keep the same rate of data moving under the head. This method is used in CD-ROM and DVD-ROM drives. Alternatively, the disk rotation speed can stay constant, and the density of bits decreases from inner tracks to outer tracks to keep the data rate constant. This method is used in hard disks and is known as **constant angular velocity (CAV)**.

The number of sectors per track has been increasing as disk technology improves, and the outer zone of a disk usually has several hundred sectors per track. Similarly, the number of cylinders per disk has been increasing; large disks have tens of thousands of cylinders.

14.2 ■ Disk Scheduling

One of the responsibilities of the operating system is to use the hardware efficiently. For the disk drives, meeting this responsibility entails having a fast access time and disk bandwidth. The access time has two major components (also see Section 2.3.2). The **seek time** is the time for the disk arm to move the heads to the cylinder containing the desired sector. The **rotational latency** is the additional time waiting for the disk to rotate the desired sector to the disk head. The **disk bandwidth** is the total number of bytes transferred, divided by the total time between the first request for service and the completion of the last transfer. We can improve both the access time and the bandwidth by scheduling the servicing of disk I/O requests in a good order.

As we discussed in Chapter 2, whenever a process needs I/O to or from the disk, it issues a system call to the operating system. The request specifies several pieces of information:

- Whether this operation is input or output
- What the disk address for the transfer is
- What the memory address for the transfer is
- What the number of bytes to be transferred is

If the desired disk drive and controller are available, the request can be serviced immediately. If the drive or controller is busy, any new requests for service will be placed on the queue of pending requests for that drive. For a multiprogramming system with many processes, the disk queue may often have several pending requests. Thus, when one request is completed, the operating system chooses which pending request to service next.

14.2.1 FCFS Scheduling

The simplest form of disk scheduling is, of course, the first-come, first-served (FCFS) algorithm. This algorithm is intrinsically fair, but it generally does not provide the fastest service. Consider, for example, a disk queue with requests for I/O to blocks on cylinders

98, 183, 37, 122, 14, 124, 65, 67,

in that order. If the disk head is initially at cylinder 53, it will first move from 53 to 98, then to 183, 37, 122, 14, 124, 65, and finally to 67, for a total head movement of 640 cylinders. This schedule is diagrammed in Figure 14.1.

The wild swing from 122 to 14 and then back to 124 illustrates the problem with this schedule. If the requests for cylinders 37 and 14 could be serviced together, before or after the requests at 122 and 124, the total head movement could be decreased substantially, and performance could be thereby improved.

14.2.2 SSTF Scheduling

It seems reasonable to service all the requests close to the current head position, before moving the head far away to service other requests. This assumption is the basis for the **shortest-seek-time-first (SSTF)** algorithm. The SSTF algorithm selects the request with the minimum seek time from the current head position. Since seek time increases with the number of cylinders traversed by the head, SSTF chooses the pending request closest to the current head position.

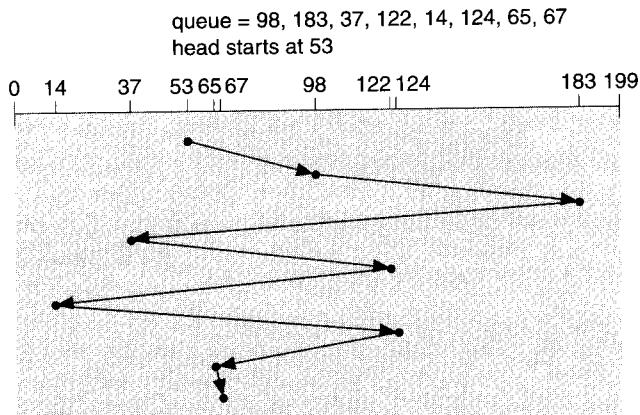


Figure 14.1 FCFS disk scheduling.

For our example request queue, the closest request to the initial head position (53) is at cylinder 65. Once we are at cylinder 65, the next closest request is at cylinder 67. From there, the request at cylinder 37 is closer than 98, so 37 is served next. Continuing, we service the request at cylinder 14, then 98, 122, 124, and finally 183 (Figure 14.2). This scheduling method results in a total head movement of only 236 cylinders—little more than one-third of the distance needed for FCFS scheduling of this request queue. This algorithm gives a substantial improvement in performance.

SSTF scheduling is essentially a form of shortest-job-first (SJF) scheduling, and, like SJF scheduling, it may cause starvation of some requests. Remember that requests may arrive at any time. Suppose that we have two requests in

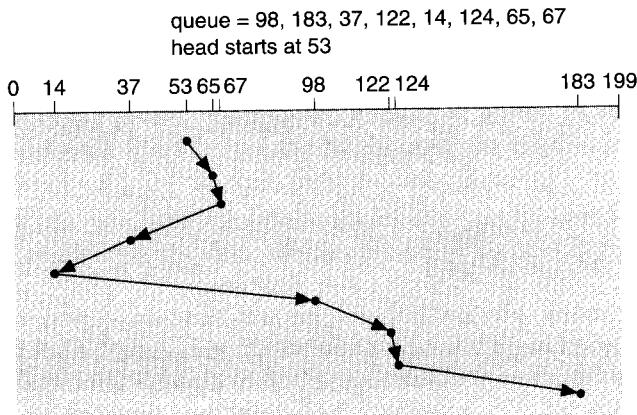


Figure 14.2 SSTF disk scheduling.

the queue, for cylinders 14 and 186, and while servicing the request from 14, a new request near 14 arrives. This new request will be serviced next, making the request at 186 wait. While this request is being serviced, another request close to 14 could arrive. In theory, a continual stream of requests near one another could arrive, causing the request for cylinder 186 to wait indefinitely. This scenario becomes increasingly likely if the pending-request queue grows long.

Although the SSTF algorithm is a substantial improvement over the FCFS algorithm, it is not optimal. In the example, we can do better by moving the head from 53 to 37, even though the latter is not closest, and then to 14, before turning around to service 65, 67, 98, 122, 124, and 183. This strategy reduces the total head movement to 208 cylinders.

14.2.3 SCAN Scheduling

In the **SCAN algorithm**, the disk arm starts at one end of the disk, and moves toward the other end, servicing requests as it reaches each cylinder, until it gets to the other end of the disk. At the other end, the direction of head movement is reversed, and servicing continues. The head continuously scans back and forth across the disk. We again use our example.

Before applying SCAN to schedule the requests on cylinders 98, 183, 37, 122, 14, 124, 65, and 67, we need to know the direction of head movement, in addition to the head's current position (53). If the disk arm is moving toward 0, the head will service 37 and then 14. At cylinder 0, the arm will reverse and will move toward the other end of the disk, servicing the requests at 65, 67, 98, 122, 124, and 183 (Figure 14.3). If a request arrives in the queue just in front of the head, it will be serviced almost immediately; a request arriving just behind

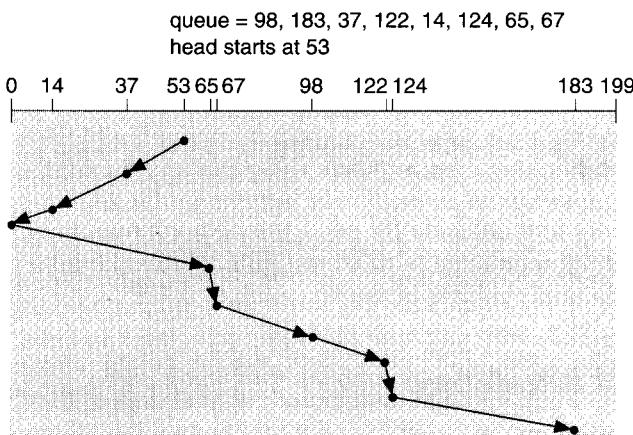


Figure 14.3 SCAN disk scheduling.

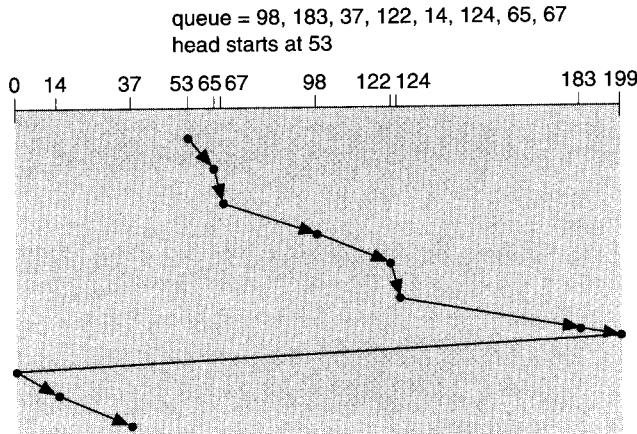


Figure 14.4 C-SCAN disk scheduling.

the head will have to wait until the arm moves to the end of the disk, reverses direction, and comes back.

The SCAN algorithm is sometimes called the **elevator algorithm**, since the disk arm behaves just like an elevator in a building, first servicing all the requests going up, and then reversing to service requests the other way.

Assuming a uniform distribution of requests for cylinders, consider the density of requests when the head reaches one end and reverses direction. At this point, relatively few requests are immediately in front of the head, since these cylinders have recently been serviced. The heaviest density of requests is at the other end of the disk. These requests have also waited the longest, so why not go there first? That is the idea of the next algorithm.

14.2.4 C-SCAN Scheduling

Circular SCAN (C-SCAN) scheduling is a variant of SCAN designed to provide a more uniform wait time. Like SCAN, C-SCAN moves the head from one end of the disk to the other, servicing requests along the way. When the head reaches the other end, however, it immediately returns to the beginning of the disk, without servicing any requests on the return trip (Figure 14.4). The C-SCAN scheduling algorithm essentially treats the cylinders as a circular list that wraps around from the final cylinder to the first one.

14.2.5 LOOK Scheduling

As we described them, both SCAN and C-SCAN move the disk arm across the full width of the disk. In practice, neither algorithm is implemented this way. More commonly, the arm goes only as far as the final request in each direction.

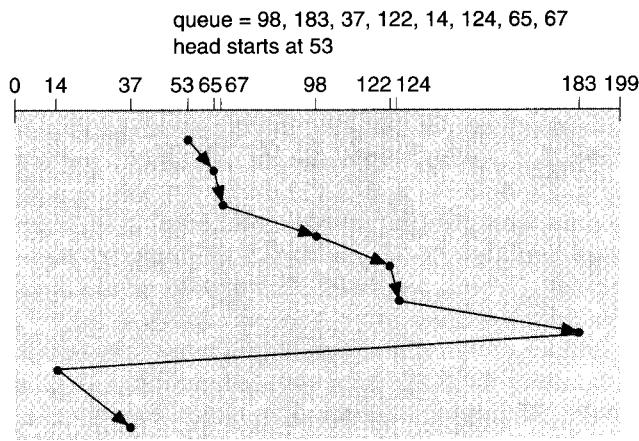


Figure 14.5 C-LOOK disk scheduling.

Then, it reverses direction immediately, without going all the way to the end of the disk. These versions of SCAN and C-SCAN are called **LOOK** and **C-LOOK scheduling**, because they *look* for a request before continuing to move in a given direction (Figure 14.5).

14.2.6 Selection of a Disk-Scheduling Algorithm

Given so many disk-scheduling algorithms, how do we choose the best one? SSTF is common and has a natural appeal because it increases performance over FCFS. SCAN and C-SCAN perform better for systems that place a heavy load on the disk, because they are less likely to have a starvation problem. For any particular list of requests, we can define an optimal order of retrieval, but the computation needed to find an optimal schedule may not justify the savings over SSTF or SCAN.

With any scheduling algorithm, however, performance depends heavily on the number and types of requests. For instance, suppose that the queue usually has just one outstanding request. Then, all scheduling algorithms are forced to behave the same, because they have only one choice for where to move the disk head: They all behave like FCFS scheduling.

The requests for disk service can be greatly influenced by the file-allocation method. A program reading a contiguously allocated file will generate several requests that are close together on the disk, resulting in limited head movement. A linked or indexed file, on the other hand, may include blocks that are widely scattered on the disk, resulting in greater head movement.

The location of directories and index blocks is also important. Since every file must be opened to be used, and opening a file requires searching the directory structure, the directories will be accessed frequently. Suppose that a

directory entry is on the first cylinder and a file's data are on the final cylinder. In this case, the disk head has to move the entire width of the disk. If the directory entry were on the middle cylinder, the head would have to move, at most, one-half the width. Caching the directories and index blocks in main memory can also help to reduce the disk-arm movement, particularly for read requests.

Because of these complexities, the disk-scheduling algorithm should be written as a separate module of the operating system, so that it can be replaced with a different algorithm if necessary. Either SSTF or LOOK is a reasonable choice for the default algorithm.

The scheduling algorithms described here consider only the seek distances. For modern disks, the rotational latency can be nearly as large as the average seek time. But it is difficult for the operating system to schedule for improved rotational latency because modern disks do not disclose the physical location of logical blocks. Disk manufacturers have been alleviating this problem by implementing disk-scheduling algorithms in the controller hardware built into the disk drive. If the operating system sends a batch of requests to the controller, the controller can queue them and then schedule them to improve both the seek time and the rotational latency. If I/O performance were the only consideration, the operating system would gladly turn over the responsibility of disk scheduling to the disk hardware. In practice, however, the operating system may have other constraints on the service order for requests. For instance, demand paging may take priority over application I/O, and writes are more urgent than reads if the cache is running out of free pages. Also, it may be desirable to guarantee the order of a set of disk writes to make the file system robust in the face of system crashes. Consider what could happen if the operating system allocated a disk page to a file, and the application wrote data into that page before the operating system had a chance to flush the modified inode and free-space list back to disk. To accommodate such requirements, an operating system may choose to do its own disk scheduling and to spoon-feed the requests to the disk controller, one by one, for some types of I/O.

14.3 ■ Disk Management

The operating system is responsible for several other aspects of disk management, too. Here we discuss disk initialization, booting from disk, and bad-block recovery.

14.3.1 Disk Formatting

A new magnetic disk is a blank slate: It is just platters of a magnetic recording material. Before a disk can store data, it must be divided into sectors that the disk controller can read and write. This process is called low-level formatting

(or **physical formatting**). **Low-level formatting** fills the disk with a special data structure for each sector. The data structure for a sector typically consists of a header, a data area (usually 512 bytes in size), and a trailer. The header and trailer contain information used by the disk controller, such as a sector number and an **error-correcting code (ECC)**. When the controller writes a sector of data during normal I/O, the ECC is updated with a value calculated from all the bytes in the data area. When the sector is read, the ECC is recalculated and is compared with the stored value. If the stored and calculated numbers are different, this mismatch indicates that the data area of the sector has become corrupted and that the disk sector may be bad (Section 14.3.3). The ECC is an *error-correcting* code because it contains enough information that, if only a few bits of data have been corrupted, the controller can identify which bits have changed and can calculate what their correct values should be. The controller automatically does the ECC processing whenever a sector is read or written.

Most hard disks are low-level formatted at the factory as a part of the manufacturing process. This formatting enables the manufacturer to test the disk and to initialize the mapping from logical block numbers to defect-free sectors on the disk. For many hard disks, when the disk controller is instructed to low-level format the disk, it can also be told how many bytes of data space to leave between the header and trailer of all sectors. It is usually possible to choose among a few sizes, such as 256, 512, and 1,024 bytes. Formatting a disk with a larger sector size means that fewer sectors can fit on each track, but that also means fewer headers and trailers are written on each track, and thus increases the space available for user data. Some operating systems can handle only a sector size of 512 bytes.

To use a disk to hold files, the operating system still needs to record its own data structures on the disk. It does so in two steps. The first step is to **partition** the disk into one or more groups of cylinders. The operating system can treat each partition as though it were a separate disk. For instance, one partition can hold a copy of the operating system's executable code, while another holds user files. After partitioning, the second step is **logical formatting** (or creation of a file system). In this step, the operating system stores the initial file-system data structures onto the disk. These data structures may include maps of free and allocated space (a FAT or inodes) and an initial empty directory.

Some operating systems give special programs the ability to use a disk partition as a large sequential array of logical blocks, without any file-system data structures. This array is sometimes called the raw disk, and I/O to this array is termed raw I/O. For example, some database systems prefer raw I/O because it enables them to control the exact disk location where each database record is stored. Raw I/O bypasses all the file-system services, such as the buffer cache, file locking, prefetching, space allocation, file names, and directories. We can make certain applications more efficient by implementing their own special-purpose storage services on a raw partition, but most applications perform better when they use the regular file-system services.

14.3.2 Boot Block

For a computer to start running—for instance, when it is powered up or rebooted—it needs to have an initial program to run. This initial bootstrap program tends to be simple. It initializes all aspects of the system, from CPU registers to device controllers and the contents of main memory, and then starts the operating system. To do its job, the bootstrap program finds the operating-system kernel on disk, loads that kernel into memory, and jumps to an initial address to begin the operating-system execution.

For most computers, the bootstrap is stored in **read-only memory (ROM)**. This location is convenient, because ROM needs no initialization and is at a fixed location that the processor can start executing when powered up or reset. And, since ROM is read only, it cannot be infected by a computer virus. The problem is that changing this bootstrap code requires changing the ROM hardware chips. For this reason, most systems store a tiny bootstrap loader program in the boot ROM, whose only job is to bring in a full bootstrap program from disk. The full bootstrap program can be changed easily: A new version is simply written onto the disk. The full bootstrap program is stored in a partition called the boot blocks, at a fixed location on the disk. A disk that has a boot partition is called a **boot disk** or **system disk**.

The code in the boot ROM instructs the disk controller to read the boot blocks into memory (no device drivers are loaded at this point), and then starts executing that code. The full bootstrap program is more sophisticated than the bootstrap loader in the boot ROM; it is able to load the entire operating system from a nonfixed location on disk, and to start the operating system running. Even so, the full bootstrap code may be small. For example, MS-DOS uses one 512-byte block for its boot program (Figure 14.6).

14.3.3 Bad Blocks

Because disks have moving parts and small tolerances (recall that the disk head flies just above the disk surface), they are prone to failure. Sometimes the failure is complete, and the disk needs to be replaced, and its contents restored from backup media to the new disk. More frequently, one or more sectors become defective. Most disks even come from the factory with **bad blocks**. Depending on the disk and controller in use, these blocks are handled in a variety of ways.

On simple disks, such as some disks with IDE controllers, bad blocks are handled manually. For instance, the MS-DOS `format` command does a logical format and, as a part of the process, scans the disk to find bad blocks. If `format` finds a bad block, it writes a special value into the corresponding FAT entry to tell the allocation routines not to use that block. If blocks go bad during normal operation, a special program (such as `chkdsk`) must be run manually to search for the bad blocks and to lock them away as before. Data that resided on the bad blocks usually are lost.

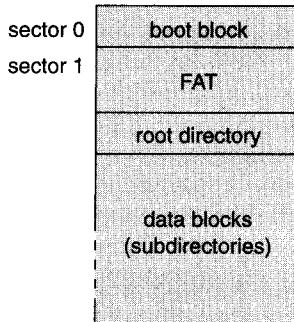


Figure 14.6 MS-DOS disk layout.

More sophisticated disks, such as the SCSI disks used in high-end PCs and most workstations and servers, are smarter about bad-block recovery. The controller maintains a list of bad blocks on the disk. The list is initialized during the low-level format at the factory, and is updated over the life of the disk. Low-level formatting also sets aside spare sectors not visible to the operating system. The controller can be told to replace each bad sector logically with one of the spare sectors. This scheme is known as **sector sparing** or **forwarding**.

A typical bad-sector transaction might be as follows:

- The operating system tries to read logical block 87.
- The controller calculates the ECC and finds that the sector is bad. It reports this finding to the operating system.
- The next time that the system is rebooted, a special command is run to tell the SCSI controller to replace the bad sector with a spare.
- After that, whenever the system requests logical block 87, the request is translated into the replacement sector's address by the controller.

Such a redirection by the controller could invalidate any optimization by the operating system's disk-scheduling algorithm! For this reason, most disks are formatted to provide a few spare sectors in each cylinder, and a spare cylinder as well. When a bad block is remapped, the controller uses a spare sector from the same cylinder, if possible.

As an alternative to sector sparing, some controllers can be instructed to replace a bad block by **sector slipping**. Here is an example: Suppose that logical block 17 becomes defective, and the first available spare follows sector 202. Then, sector slipping would remap all the sectors from 17 to 202, moving them all down one spot. That is, sector 202 would be copied into the spare, then sector 201 into 202, and then 200 into 201, and so on, until sector 18 is copied into sector 19. Slipping the sectors in this way frees up the space of sector 18, so sector 17 can be mapped to it.

The replacement of a bad block generally is not a totally automatic process because the data in the bad block are usually lost. Thus, whatever file was using that block must be repaired (for instance, by restoration from a backup tape), and that requires manual intervention.

14.4 ■ Swap-Space Management

Swap-space management is another low-level task of the operating system. Virtual memory uses disk space as an extension of main memory. Since disk access is much slower than memory access, using swap space significantly decreases system performance. The main goal for the design and implementation of swap space is to provide the best throughput for the virtual-memory system. In this section, we discuss how swap space is used, where swap space is located on disk, and how swap space is managed.

14.4.1 Swap-Space Use

Swap space is used in various ways by different operating systems, depending on the implemented memory-management algorithms. For instance, systems that implement swapping may use swap space to hold the entire process image, including the code and data segments. Paging systems may simply store pages that have been pushed out of main memory. The amount of swap space needed on a system can therefore vary depending on the amount of physical memory, the amount of virtual memory it is backing, and the way in which the virtual memory is used. It can range from a few megabytes of disk space to gigabytes.

Some operating systems, such as UNIX, allow the use of multiple swap spaces. These swap spaces are usually put on separate disks, so the load placed on the I/O system by paging and swapping can be spread over the system's I/O devices.

Note that it is safer to overestimate than to underestimate swap space, because if a system runs out of swap space it may be forced to abort processes or may crash entirely. Overestimation wastes disk space that could otherwise be used for files, but does no other harm.

14.4.2 Swap-Space Location

A swap space can reside in two places: Swap space can be carved out of the normal file system, or it can be in a separate disk partition. If the swap space is simply a large file within the file system, normal file-system routines can be used to create it, name it, and allocate its space. This approach, though easy to implement, is also inefficient. Navigating the directory structure and the disk-allocation data structures takes time and (potentially) extra disk accesses. External fragmentation can greatly increase swapping times by forcing multiple

seeks during reading or writing of a process image. We can improve performance by caching the block location information in physical memory, and by using special tools to allocate physically contiguous blocks for the swap file, but the cost of traversing the file-system data structures still remains.

Alternatively, swap space can be created in a separate disk partition. No file system or directory structure is placed on this space. Rather, a separate swap-space storage manager is used to allocate and deallocate the blocks. This manager uses algorithms optimized for speed, rather than for storage efficiency. Internal fragmentation may increase, but this tradeoff is acceptable because data in the swap space generally live for much shorter amounts of time than do files in the file system, and the swap area may be accessed much more frequently. This approach creates a fixed amount of swap space during disk partitioning. Adding more swap space can be done only via repartitioning of the disk (which involves moving or destroying and restoring the other file-system partitions from backup), or via adding another swap space elsewhere.

Some operating systems are flexible and can swap both in raw partitions and in file-system space. Solaris 2 is an example. The policy and implementation are separate, allowing the machine's administrator to decide which type to use. The tradeoff is between the convenience of allocation and management in the file system, and the performance of swapping in raw partitions.

14.4.3 Swap-Space Management: An Example

To illustrate the methods used to manage swap space, we now follow the evolution of swapping and paging in UNIX. As discussed fully in Appendix A, UNIX started with an implementation of swapping that copied entire processes between contiguous disk regions and memory. UNIX evolved to a combination of swapping and paging, as paging hardware became available.

In 4.3 BSD, swap space is allocated to a process when the process is started. Enough space is set aside to hold the program, known as the **text pages** or the **text segment**, and the **data segment** of the process. Preallocating all the needed space in this way generally prevents a process from running out of swap space while it executes. When a process starts, its text is paged in from the file system. These pages are written out to swap when necessary, and are read back in from there, so the file system is consulted only once for each text page. Pages from the data segment are read in from the file system, or are created (if they are uninitialized), and are written to swap space and paged back in as needed. One optimization (for instance, when two users run the same editor) is that processes with identical text pages share these pages, both in physical memory and in swap space.

Two per-process **swap maps** are used by the kernel to track swap-space use. The text segment is a fixed size, so its swap space is allocated in 512 KB chunks, except for the final chunk, which holds the remainder of the pages, in 1 KB increments (Figure 14.7).

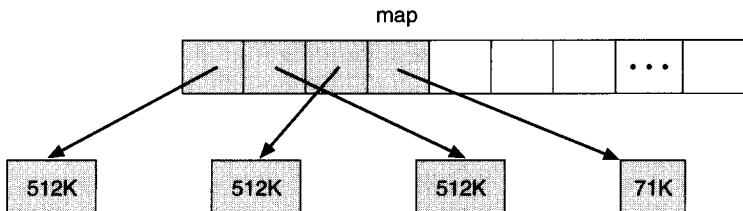


Figure 14.7 4.3 BSD text-segment swap map.

The data-segment swap map is more complicated, because the data segment can grow over time. The map is of fixed size, but contains swap addresses for blocks of varying size. Given index i , a block pointed to by swap-map entry i is of size $2^i \times 16$ KB, to a maximum of 2 MB. This data structure is shown in Figure 14.8. (The block size minimum and maximum are variable, and can be changed at system reboot.) When a process tries to grow its data segment beyond the final allocated block in its swap area, the operating system allocates another block, twice as large as the previous one. This scheme results in small processes using only small blocks. It also minimizes fragmentation. The blocks of large processes can be found quickly, and the swap map remains small.

In Solaris 1 (SunOS 4), the designers made changes to standard UNIX methods to improve efficiency and reflect technological changes. When a process executes, text-segment pages are brought in from the file system, accessed in main memory, and thrown away if selected for pageout. It is more efficient to reread a page from the file system than to write it to swap space and then to reread it from there.

More changes were made in Solaris 2. The biggest change is that Solaris 2 allocates swap space only when a page is forced out of physical memory, rather than when the virtual-memory page is first created. This change gives better performance on modern computers, which have more physical memory than older systems and tend to page less.

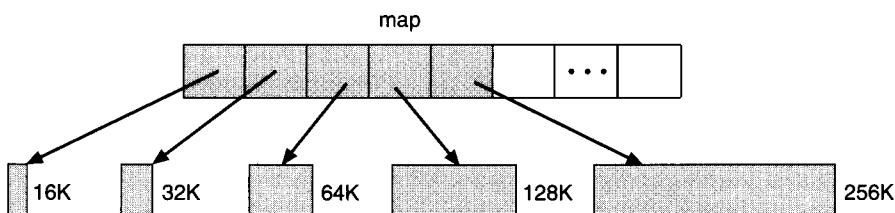


Figure 14.8 4.3 BSD data-segment swap map.

14.5 ■ RAID Structure

Disk drives have continued to get smaller and cheaper, so it is now economically feasible to attach a large number of disks to a computer system.

Having a large number of disks in a system presents opportunities for improving the rate at which data can be read or written, if the disks are operated in parallel. Furthermore, this setup offers the potential for improving the reliability of data storage, because redundant information can be stored on multiple disks. Thus, failure of one disk does not lead to loss of data. A variety of disk-organization techniques, collectively called redundant arrays of inexpensive disks (RAID), are commonly used to address the performance and reliability issues.

In the past, RAIDs composed of small cheap disks were viewed as a cost-effective alternative to large, expensive disks; today, RAIDs are used for their higher reliability and higher data-transfer rate, rather than for economic reasons. Hence, the *I* in *RAID* stands for “independent”, instead of “inexpensive.”

14.5.1 Improvement of Reliability via Redundancy

Let us first consider reliability. The chance that some disk out of a set of N disks will fail is much higher than the chance that a specific single disk will fail. Suppose that the **mean time to failure** of a single disk is 100,000 hours. Then, the mean time to failure of some disk in an array of 100 disks will be $100,000/100 = 1,000$ hours, or 41.66 days, which is not long at all! If we store only one copy of the data, then each disk failure will result in loss of a significant amount of data—such a high rate of data loss is unacceptable.

The solution to the problem of reliability is to introduce **redundancy**; we store extra information that is not needed normally, but that can be used in the event of failure of a disk to rebuild the lost information. Thus, even if a disk fails, data are not lost.

The simplest (but most expensive) approach to introducing redundancy is to duplicate every disk. This technique is called **mirroring** (or **shadowing**). A logical disk then consists of two physical disks, and every write is carried out on both disks. If one of the disks fails, the data can be read from the other. Data will be lost only if the second disk fails before the first failed disk is replaced.

The mean time to failure—where *failure* is the loss of data—of a mirrored disk depends on two factors: the mean time to failure of the individual disks, as well as on the **mean time to repair**, which is the time it takes (on average) to replace a failed disk and to restore the data on it. Suppose that the failures of the two disks are **independent**; that is, the failure of one disk is not connected to the failure of the other. Then, if the mean time to failure of a single disk is 100,000 hours and the mean time to repair is 10 hours, then the **mean time to data loss** of a mirrored disk system is $100,000^2/(2 * 10) = 500 * 10^6$ hours, or 57,000 years!

You should be aware that the assumption of independence of disk failures is not valid. Power failures and natural disasters, such as earthquakes, fires, and floods, may result in damage to both disks at the same time. Also, manufacturing defects in a batch of disks can cause correlated failures. As disks age, the probability of failure increases, increasing the chance that a second disk will fail while the first is being repaired. In spite of all these considerations, however, mirrored-disk systems offer much higher reliability than do single-disk systems.

Power failures are a particular source of concern, since they occur far more frequently than do natural disasters. However, even with mirroring of disks, if writes are in progress to the same block in both disks, and power fails before both blocks are fully written, the two blocks can be in an inconsistent state. The solution to this problem is to write one copy first, then the next, so that one of the two copies is always consistent. Some extra actions are required when we restart after a power failure, to recover from incomplete writes.

14.5.2 Improvement in Performance via Parallelism

Now let us consider the benefit of parallel access to multiple disks. With disk mirroring, the rate at which read requests can be handled is doubled, since read requests can be sent to either disk (as long as both disks in a pair are functional, as is almost always the case). The transfer rate of each read is the same as in a single-disk system, but the number of reads per unit time has doubled.

With multiple disks, we can improve the transfer rate as well (or instead) by striping data across multiple disks. In its simplest form, **data striping** consists of splitting the bits of each byte across multiple disks; such striping is called **bit-level striping**. For example, if we have an array of eight disks, we write bit i of each byte to disk i . The array of eight disks can be treated as a single disk with sectors that are eight times the normal size, and, more important, that have eight times the access rate. In such an organization, every disk participates in every access (read or write), so the number of accesses that can be processed per second is about the same as on a single disk, but each access can read eight times as many data in the same time as on a single disk.

Bit-level striping can be generalized to a number of disks that either is a multiple of 8 or divides 8. For example, if we use an array of four disks, bits i and $4+i$ of each byte go to disk i . Further, striping does not need to be at the level of bits of a byte: For example, in **block-level striping**, blocks of a file are striped across multiple disks; with n disks, block i of a file goes to disk $(i \bmod n) + 1$. Other levels of striping, such as bytes of a sector or sectors of a block, also are possible.

In summary, there are two main goals of parallelism in a disk system:

1. Increase the throughput of multiple small accesses (that is, page accesses) by load balancing.
2. Reduce the response time of large accesses.

14.5.3 RAID Levels

Mirroring provides high reliability, but it is expensive. Striping provides high data-transfer rates, but it does not improve reliability. Numerous schemes to provide redundancy at lower cost by using the idea of disk striping combined with “parity” bits (which we describe next) have been proposed. These schemes have different cost–performance tradeoffs and are classified into levels called **RAID levels**. We describe the various levels here; Figure 14.9 shows them pictorially (in the figure, P indicates error-correcting bits and C indicates a second copy of the data). In all cases depicted in the figure, four disks’ worth of

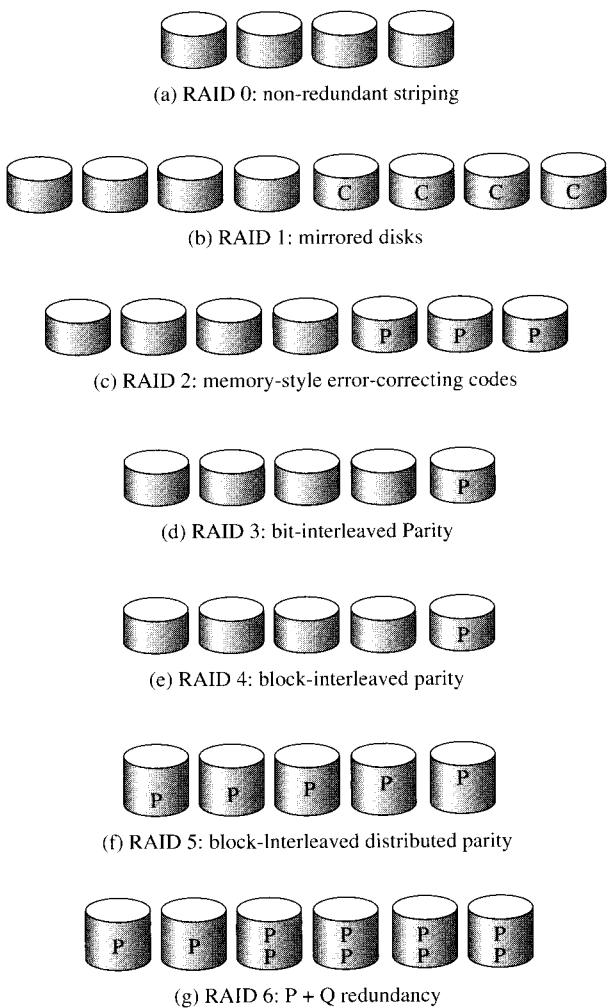


Figure 14.9 RAID levels.

data is stored, and the extra disks are used to store redundant information for failure recovery.

- **RAID Level 0:** RAID level 0 refers to disk arrays with striping at the level of blocks, but without any redundancy (such as mirroring or parity bits). Figure 14.9a shows an array of size 4.
- **RAID Level 1:** RAID level 1 refers to disk mirroring. Figure 14.9b shows a mirrored organization that holds four disks' worth of data.
- **RAID Level 2:** RAID level 2 is also known as **memory-style error-correcting-code (ECC) organization**. Memory systems have long implemented error detection using parity bits. Each byte in a memory system may have a parity bit associated with it that records whether the numbers of bits in the byte set to 1 is even (parity=0) or odd (parity=1). If one of the bits in the byte gets damaged (either a 1 becomes a 0, or a 0 becomes a 1), the parity of the byte changes and thus will not match the stored parity. Similarly, if the stored parity bit gets damaged, it will not match the computed parity. Thus, all single-bit errors are detected by the memory system. Error-correcting schemes store two or more extra bits, and can reconstruct the data if a single bit gets damaged. The idea of ECC can be used directly in disk arrays via striping of bytes across disks. For example, the first bit of each byte could be stored in disk 1, the second bit in disk 2, and so on until the eighth bit is stored in disk 8, and the error-correction bits are stored in further disks. This scheme is shown pictorially in Figure 14.9, where the disks labeled P store the error-correction bits. If one of the disks fails, the remaining bits of the byte and the associated error-correction bits can be read from other disks and be used to reconstruct the damaged data. Figure 14.9c shows an array of size 4; note RAID level 2 requires only three disks' overhead for four disks of data, unlike RAID level 1, which required four disks' overhead.
- **RAID level 3:** RAID level 3, or **bit-interleaved parity organization**, improves on level 2 by noting that, unlike memory systems, disk controllers can detect whether a sector has been read correctly, so a single parity bit can be used for error correction, as well as for detection. The idea is as follows. If one of the sectors gets damaged, we know exactly which sector it is, and, for each bit in the sector, we can figure out whether it is a 1 or a 0 by computing the parity of the corresponding bits from sectors in the other disks. If the parity of the remaining bits is equal to the stored parity, the missing bit is 0; otherwise, it is 1. RAID level 3 is as good as level 2 but is less expensive in the number of extra disks (it has only a one-disk overhead), so level 2 is not used in practice. This scheme is shown pictorially in Figure 14.9d.

RAID level 3 has two benefits over level 1. Only one parity disk is needed for several regular disks, unlike one mirror disk for every disk in

level 1, thus reducing the storage overhead. Since reads and writes of a byte are spread out over multiple disks, with N -way striping of data, the transfer rate for reading or writing a single block is N times as fast as with a RAID-level-1 organization using N -way striping. On the other hand, RAID level 3 supports a lower number of I/Os per second, since every disk has to participate in every I/O request. A further performance problem with RAID 3 (as with all parity-based RAID levels) is the expense of computing and writing the parity. This overhead results in significantly slower writes, as compared to non-parity RAID arrays. To moderate this performance penalty, many RAID storage arrays include a hardware controller with dedicated parity hardware. This offloads the parity computation from the CPU to the array. The array has a **non-volatile RAM (NVRAM)** cache as well, to store the blocks while the parity is computed and to buffer the writes from the controller to the spindles. This combination can make parity RAID almost as fast as non-parity. In fact, a caching array doing parity RAID can outperform a non-caching non-parity RAID.

- **RAID Level 4:** RAID level 4, or **block-interleaved parity organization**, uses block-level striping, as in RAID 0, and in addition keeps a parity block on a separate disk for corresponding blocks from N other disks. This scheme is shown pictorially in Figure 14.9e. If one of the disks fails, the parity block can be used with the corresponding blocks from the other disks to restore the blocks of the failed disk.

A block read accesses only one disk, allowing other requests to be processed by the other disks. Thus, the data-transfer rate for each access is slower, but multiple read accesses can proceed in parallel, leading to a higher overall I/O rate. The transfer rates for large reads is high, since all the disks can be read in parallel; large writes also have high transfer rates, since the data and parity can be written in parallel.

Small independent writes, on the other hand, cannot be performed in parallel. A write of a block has to access the disk on which the block is stored, as well as the parity disk, since the parity block has to be updated. Moreover, both the old value of the parity block and the old value of the block being written have to be read for the new parity to be computed. This is known as the **read-modify-write**. Thus, a single write requires four disk accesses: two to read the two old blocks, and two to write the two new blocks.

- **RAID level 5:** RAID level 5, or **block-interleaved distributed parity**, differs from level 4 by spreading data and parity among all $N + 1$ disks, rather than storing data in N disks and parity in one disk. For each block, one of the disks stores the parity, and the others store data. For example, with an array of five disks, the parity for the n th block is stored in disk $(n \bmod 5) + 1$; the n th blocks of the other four disks store actual data for that block. This setup is denoted pictorially in Figure 14.9f, where the P s are distributed across all

the disks. A parity block cannot store parity for blocks in the same disk, because a disk failure would result in loss of data as well as of parity, and hence would not be recoverable. By spreading the parity across all the disks in the set, RAID 5 avoids the potential overuse of a single parity disk that can occur with RAID 4.

- **RAID Level 6:** RAID level 6, also called the **P+Q redundancy scheme**, is much like RAID level 5, but stores extra redundant information to guard against multiple disk failures. Instead of using parity, error-correcting codes such as the **Reed–Solomon codes** are used. In the scheme shown in Figure 14.9g, 2 bits of redundant data are stored for every 4 bits of data—unlike 1 parity bit in level 5—and the system can tolerate two disk failures.
- **RAID level 0 + 1:** RAID level 0 + 1 refers to a combination of RAID levels 0 and 1. RAID 0 provides the performance, while RAID 1 provides the reliability. Generally, it provides better performance than RAID 5. It is common in environments where both performance and reliability are important. Unfortunately, it doubles the number of disks needed for storage, as does RAID 1, so it is also more expensive. In RAID 0 + 1, a set of disks are striped, and then the stripe is mirrored to another, equivalent stripe. Another RAID option that is becoming available commercially is RAID 1 + 0, in which disks are mirrored in pairs, and then the resulting mirror pairs are striped. This RAID has some theoretical advantages over RAID 0 + 1. For example, if a single disk fails in RAID 0 + 1, the entire stripe is inaccessible, leaving only the other stripe available. With a failure in RAID 1 + 0, the single disk is unavailable, but its mirrored pair is still available as are all the rest of the disks (Figure 14.10).

Finally, we note that numerous variations have been proposed to the basic RAID schemes described here. As a result, some confusion may exist about the exact definitions of the different RAID levels.

14.5.4 Selecting a RAID Level

If a disk fails, the time to rebuild its data can be significant and will vary with the RAID level used. Rebuilding is easiest for RAID level 1, since data can be copied from another disk; for the other levels, we need to access all the other disks in the array to rebuild data in a failed disk. The rebuild performance of a RAID system may be an important factor if continuous supply of data is required, as it is in high-performance or interactive database systems. Furthermore, rebuild performance influences the mean time to failure.

RAID level 0 is used in high-performance applications where data loss is not critical. RAID level 1 is popular for applications that require high reliability with fast recovery. RAID 0 + 1 and 1 + 0 are used where performance and reliability are important, for example for small databases. Due to RAID 1's high

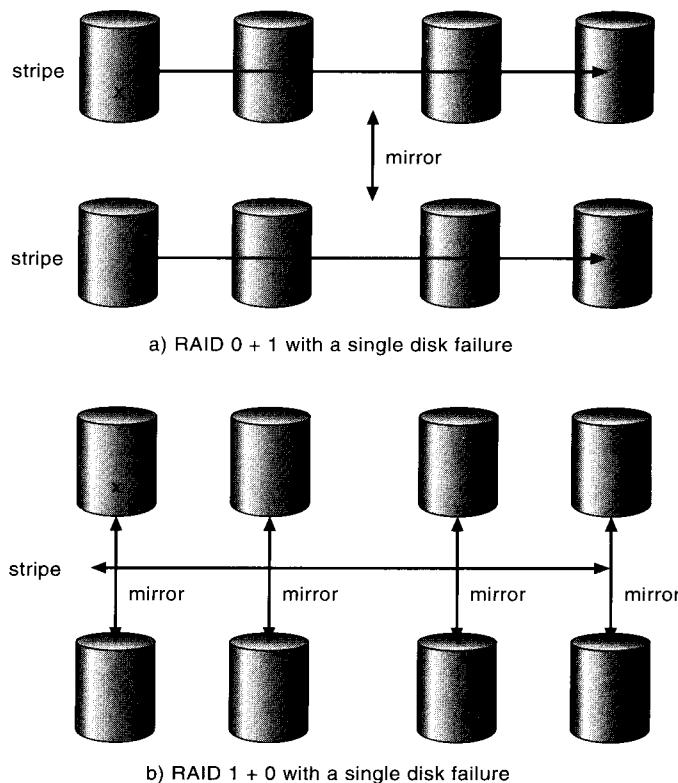


Figure 14.10 RAID 0 + 1 and 1 + 0.

space overhead, RAID level 5 is often preferred for storing large volumes of data. Level 6 is not supported currently by many RAID implementations, but it should offer better reliability than level 5.

RAID system designers have to make several other decisions as well. For example, how many disks should be in an array? How many bits should be protected by each parity bit? If more disks are in an array, data-transfer rates are higher, but the system is more expensive. If more bits are protected by a parity bit, the space overhead due to parity bits is lower, but the chance that a second disk will fail before the first failed disk is repaired is greater, and that will result in data loss.

One other aspect of most RAID implementations is a hot spare disk or disks. A **hot spare** is not used for data, but is configured to be used as a replacement should any other disk fail. For instance, a hot spare can be used to rebuild a mirror pair should one of the disks in the pair fail. In this way, the RAID level can be reestablished automatically, without waiting for the failed disk to be replaced. Allocating more than one hot spare allows more than one failure to be repaired without human intervention.

14.5.5 Extensions

The concepts of RAID have been generalized to other storage devices, including arrays of tapes, and even to the broadcast of data over wireless systems. When applied to arrays of tapes, the RAID structures are able to recover data even if one of the tapes in an array of tapes is damaged. When applied to broadcast of data, a block of data is split into short units and is broadcast along with a parity unit; if one of the units is not received for any reason, it can be reconstructed from the other units. Commonly, tape-drive robots containing multiple tape drives will stripe data across all the drives to increase throughput and decrease backup time.

14.6 ■ Disk Attachment

Computers access disk storage in two ways. One way is via I/O ports (or **host-attached storage**); this is common on small systems. The other way is via a remote host via a distributed file system; this is referred to as **network-attached storage**.

14.6.1 Host-Attached Storage

Host-attached storage is storage accessed via local I/O ports. These ports are available in several technologies. The typical desktop PC uses an I/O bus architecture called IDE or ATA. This architecture supports a maximum of two drives per I/O bus. High-end workstations and servers generally use more sophisticated I/O architectures such as SCSI and fibre channel (FC).

SCSI is a bus architecture. Its physical medium is usually a ribbon cable having a large number of conductors (typically 50 or 68). The SCSI protocol supports a maximum of 16 devices on the bus. Typically this consists of one controller card in the host (the **SCSI initiator**), and up to 15 storage devices (the **SCSI targets**). A SCSI disk is a typical SCSI target, but the protocol provides the ability to address up to 8 **logical units** in each SCSI target. A typical use of logical unit addressing is to direct commands to components of a RAID array, or components of a removable media library (such as a CD jukebox sending commands to the media changer mechanism or to one of the drives).

FC is a high-speed serial architecture. This architecture can operate over optical fiber or over a 4-conductor copper cable. It has two variants. One is a large switched fabric having a 24-bit address space. This method is expected to dominate in the future, and is the basis of **storage-area networks (SANs)**. Because of the large address space and the switched nature of the communication, multiple hosts and storage devices can attach to the fabric, allowing great flexibility in I/O communication. The other is an **arbitrated loop (FC-AL)** that can address 126 devices (drives and controllers).

A wide variety of storage devices are suitable for use as host-attached storage. Among these are hard disk drives, RAID arrays, and CD, DVD, and tape drives.

The I/O commands that initiate data transfers to a host-attached storage device are reads and writes of logical data blocks, directed to specifically identified storage units (such as bus ID, SCSI ID, and target logical unit, for example).

14.6.2 Network-Attached Storage

A network-attached storage device is a special-purpose storage system that is accessed remotely over a data network (Figure 14.11). Clients access network-attached storage (NAS) via a remote-procedure-call interface such as NFS for UNIX systems, or CIFS for Windows machines. The remote procedure calls (RPCs) are carried via TCP or UDP over an IP network—usually the same local-area network (LAN) that carries all data traffic to the clients. The network-attached storage unit is usually implemented as a RAID array with software that implements the remote procedure call interface. It is easiest to think of NAS as simply another storage-access protocol. For example, rather than using a SCSI device driver and SCSI protocols to access storage, a system using NAS would use RPC over TCP/IP.

Network-attached storage provides a convenient way for all the computers on a LAN to share a pool of storage, with the same ease of naming and access enjoyed with local host-attached storage. However, it tends to be less efficient and have lower performance than some direct-attached storage options.

14.6.3 Storage-Area Network

One drawback of network-attached storage systems is that the storage I/O operations consume bandwidth on the data network, thereby increasing the latency of network communication. This problem can be particularly acute

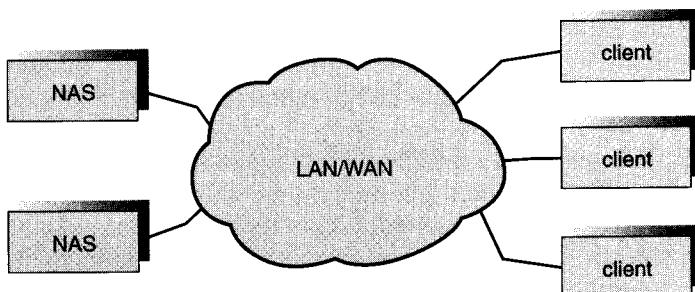


Figure 14.11 Network-attached storage.

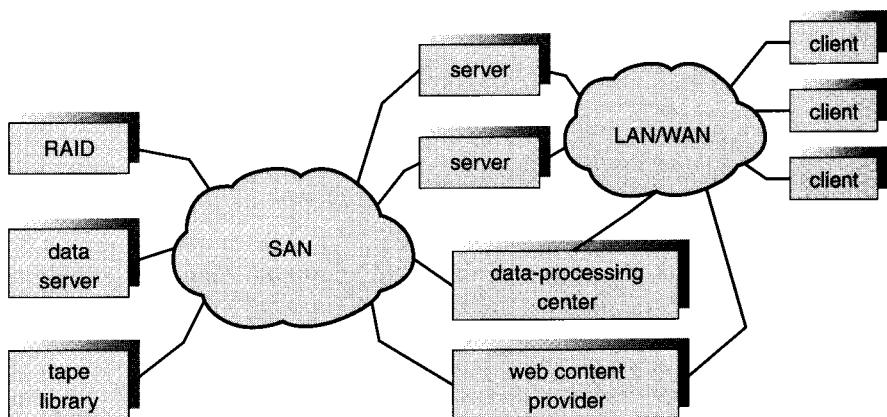


Figure 14.12 Storage-area network.

in large client–server installations—the communication between servers and clients competes for bandwidth with the communication among servers and storage devices.

A storage-area network (SAN) is a private network (using storage protocols rather than networking protocols) among the servers and storage units, separate from the LAN or WAN that connects the servers to the clients (Figure 14.12). The power of a SAN lies in its flexibility. Multiple hosts and multiple storage arrays can attach to the same SAN, and storage can be dynamically allocated to hosts. As one example, if a host is running low on disk space, the SAN can be configured to allocate more storage to that host. In 2001, many proprietary single-vendor SAN systems are available, but SAN components are not well standardized or interoperable. Most SAN systems in 2001 are based on fibre-channel loops or fibre-channel switched networks. One emerging alternative to a fibre-channel interconnect for the SAN is storage over IP network infrastructure such as Gigabit Ethernet. Another potential alternative is a special-purpose SAN architecture named Infiniband, which provides hardware and software support for high-speed interconnection networks for servers and storage units.

14.7 ■ Stable-Storage Implementation

In Chapter 7, we introduced the write-ahead log, which required the availability of stable storage. By definition, information residing in stable storage is *never* lost. To implement such storage, we need to replicate the needed information on multiple storage devices (usually disks) with independent failure modes. We need to coordinate the writing of updates in a way that guarantees that a failure during an update will not leave all the copies in a damaged state, and that, when we are recovering from a failure, we can force all copies to a

consistent and correct value, even if another failure occurs during the recovery. In the remainder of this section, we discuss how to meet our needs.

A disk write results in one of three outcomes:

1. **Successful completion:** The data were written correctly on disk.
2. **Partial failure:** A failure occurred in the midst of transfer, so only some of the sectors were written with the new data, and the sector being written during the failure may have been corrupted.
3. **Total failure:** The failure occurred before the disk write started, so the previous data values on the disk remain intact.

We require that, whenever a failure occurs during writing of a block, the system detects it and invokes a recovery procedure to restore the block to a consistent state. To do that, the system must maintain two physical blocks for each logical block. An output operation is executed as follows:

1. Write the information onto the first physical block.
2. When the first write completes successfully, write the same information onto the second physical block.
3. Declare the operation complete only after the second write completes successfully.

During recovery from a failure, each pair of physical blocks is examined. If both are the same and no detectable error exists, then no further action is necessary. If one block contains a detectable error, then we replace its contents with the value of the other block. If both blocks contain no detectable error, but they differ in content, then we replace the content of the first block with the value of the second. This recovery procedure ensures that a write to stable storage either succeeds completely or results in no change.

We can extend this procedure easily to allow the use of an arbitrarily large number of copies of each block of stable storage. Although a large number of copies further reduces the probability of a failure, it is usually reasonable to simulate stable storage with only two copies. The data in stable storage are guaranteed to be safe unless a failure destroys all the copies.

Because waiting for disk writes to complete (synchronous I/O) is time consuming, many storage arrays add NVRAM as a cache. Because the memory is non-volatile (usually it has battery power as a backup to the unit's power), it can be trusted to store the data on its way to the disks. It is thus considered part of the stable storage. Writes to it are much faster than to disk, so performance is greatly improved.

14.8 ■ Tertiary-Storage Structure

Would you buy a VCR that had inside it only one tape that you could not take out or replace? Or an audio cassette player or CD player that had one album sealed inside? Of course not. You expect to use a VCR or CD player with many relatively inexpensive tapes or disks. On a computer as well, using many inexpensive cartridges with one drive lowers the overall cost.

14.8.1 Tertiary-Storage Devices

Low cost is the defining characteristic of tertiary storage. So, in practice, tertiary storage is built with **removable media**. The most common examples of removable media are floppy disks, CD-ROMs, and tapes; many other kinds of tertiary-storage devices are available as well.

14.8.1.1 Removable Disks

Removable disks are one kind of tertiary storage. Floppy disks are an example of removable magnetic disks. They are made from a thin flexible disk coated with magnetic material, enclosed in a protective plastic case. Although common floppy disks can hold only about 1 MB, similar technology is used for removable magnetic disks that hold more than 1 GB. Removable magnetic disks can be nearly as fast as hard disks, although the recording surface is at greater risk of damage from scratches.

A **magneto-optic disk** is another kind of removable disk. It records data on a rigid platter coated with magnetic material, but the recording technology is quite different from that for a magnetic disk. The magneto-optic head flies much farther from the disk surface than a magnetic disk head does, and the magnetic material is covered with a thick protective layer of plastic or glass. This arrangement makes the disk much more resistant to head crashes.

The drive has a coil that produces a magnetic field; at room temperature, the field is too large and too weak to magnetize a bit on the disk. To write a bit, the disk head flashes a laser beam at the disk surface. The laser is aimed at a tiny spot where a bit is to be written. The laser heats this spot, which makes the spot susceptible to the magnetic field. So the large, weak magnetic field can record a tiny bit.

The magneto-optic head is too far from the disk surface to read the data by detecting the tiny magnetic fields in the way that the head of a hard disk does. Instead, the drive reads a bit using a property of laser light called the **Kerr effect**. When a laser beam is bounced off of a magnetic spot, the polarization of the laser beam is rotated clockwise or counter-clockwise, depending on the orientation of the magnetic field. This rotation is what the head detects to read a bit.

Another category of removable disk is the **optical disk**. These disks do not use magnetism at all. They use special materials that can be altered by laser light to have relatively dark or bright spots. One example of optical-disk technology is the phase-change disk.

The **phase-change disk** is coated with a material that can freeze into either a crystalline or an amorphous state. The crystalline state is more transparent, and hence a laser beam is brighter when it passes through the phase-change material and bounces off the reflective layer. The phase-change drive uses laser light at three different powers: low power to read data, medium power to erase the disk by melting and refreezing the recording medium into the crystalline state, and a high power to melt the medium into the amorphous state to write to the disk. The most common examples of this technology are the re-recordable CD-RW and DVD-RW.

The kinds of disks described here can be used over and over. They are called **read-write disks**. In contrast, **write-once, read-many-times (WORM) disks** form another category. An old way to make a WORM disk is to manufacture a thin aluminum film sandwiched between two glass or plastic platters. To write a bit, the drive uses a laser light to burn a small hole through the aluminum. Because this burning cannot be reversed, any sector on the disk can be written only once. Although it is possible to destroy the information on a WORM disk by burning holes everywhere, it is virtually impossible to alter data on the disk, because holes can only be added, and the ECC code associated with each sector is likely to detect such additions. WORM disks are considered to be durable and reliable because the metal layer is safely encapsulated between the protective glass or plastic platters, and magnetic fields cannot damage the recording. A newer write-once technology records on an organic polymer dye instead of an aluminum layer: the dye absorbs laser light to form marks. This technology is used in the recordable CD-R and DVD-R.

Read-only disks, such as CD-ROM and DVD, come from the factory with the data pre-recorded. They use technology similar to that of WORM disks (although the pits are pressed, not burnt), and they are very durable.

Most removable disks are slower than their non-removable counterparts. The writing process is slower, as are rotation and sometimes seek time.

14.8.1.2 Tapes

Magnetic tape is another type of removable medium. As a general rule, a tape holds more data than an optical or magnetic disk cartridge. Tape drives and disk drives have similar transfer rates. But random access to tape is much slower than a disk seek, because it requires a fast-forward or rewind operation that takes tens of seconds, or even minutes.

Although a typical tape drive is more expensive than a typical disk drive, the price of a tape cartridge is lower than the price of the equivalent capacity of magnetic disks. So tape is an economical medium for purposes that do not

require fast random access. Tapes are commonly used to hold backup copies of disk data. They are also used in large supercomputer centers to hold the enormous volumes of data used in scientific research and by large commercial enterprises.

Some tapes can hold much more data than can a disk drive; the surface area of a tape is much larger than the surface area of a disk. The storage capacity of tapes could improve even further, because at present the **areal density** (or bits per square inch) of tape technology is much less than that for magnetic disks.

Large tape installations typically use robotic tape changers that move tapes between tape drives and storage slots in a tape library. These mechanisms give the computer automated access to a large number of tape cartridges.

A robotic tape library can lower the overall cost of data storage. A disk-resident file that will not be needed for a while can be **archived** to tape, where the cost per gigabyte can be lower; if the file is needed in the future, the computer can **stage** it back into disk storage for active use. A robotic tape library is sometimes called **near-line** storage, since it is between the high performance of **on-line** magnetic disks and the low cost of **off-line** tapes sitting on shelves in a storage room.

14.8.1.3 Future Technology

In the future, other storage technologies may become important. One promising storage technology, **holographic storage**, uses laser light to record holographic photographs on special media. We can think of a black-and-white photograph as a two-dimensional array of pixels. Each pixel represents one bit: 0 for black, or 1 for white. A sharp photograph can hold millions of bits of data. And all the pixels in a hologram are transferred in one flash of laser light, so the data rate is extremely high. With continued development, holographic storage may become commercially viable.

Another storage technology under active research is based on **micro-electronic mechanical systems (MEMS)**. The idea is to apply the fabrication technologies that produce electronic chips in order to manufacture small data storage machines. One proposal calls for the fabrication of an array of 10,000 tiny disk heads, with a square centimeter of magnetic storage material suspended above the array. When the storage material is moved lengthwise over the heads, each head accesses its own linear track of data on the material. The storage material can be shifted sideways slightly to enable all the heads to access their next track. Although it remains to be seen whether this technology can be successful, it may provide a nonvolatile data storage technology that is faster than magnetic disk and cheaper than semiconductor DRAM.

Whether the storage medium is a removable magnetic disk, a DVD, or a magnetic tape, the operating system needs to provide several capabilities to use removable media for data storage. These capabilities are discussed in Section 14.8.2.

14.8.2 Operating-System Jobs

Two major jobs of an operating system are to manage physical devices and to present a virtual-machine abstraction to applications. In this chapter, we saw that, for hard disks, the operating system provides two abstractions. One is the raw device, which is just an array of data blocks. The other is a file system. For a file system on a magnetic disk, the operating system queues and schedules the interleaved requests from several applications. Now, we shall see how the operating system does its job when the storage media are removable.

14.8.2.1 Application Interface

Most operating systems can handle removable disks almost exactly as they do fixed disks. When a blank cartridge is inserted into the drive (or mounted), the cartridge must be formatted, and then an empty file system is generated on the disk. This file system is used just like a file system on a hard disk.

Tapes are often handled differently. The operating system usually presents a tape as a raw storage medium. An application does not open a file on the tape; it opens the whole tape drive as a raw device. Usually, the tape drive then is reserved for the exclusive use of that application until the application exits or closes the tape device. This exclusivity makes sense, because random access on a tape can take tens of seconds, or even a few minutes, so interleaving random accesses to tapes from more than one application would be likely to cause thrashing.

When the tape drive is presented as a raw device, the operating system does not provide file-system services. The application must decide how to use the array of blocks. For instance, a program that backs up a hard disk to tape might store a list of file names and sizes at the beginning of the tape, and then copy the data of the files to the tape in that order.

It is easy to see the problems that can arise from this way of using tape. Since every application makes up its own rules for how to organize a tape, a tape full of data can generally be used by only the program that created it. For instance, even if we know that a backup tape contains a list of file names and file sizes followed by the file data in that order, we still would find it difficult to use the tape. How exactly are the file names stored? Are the file sizes in binary or in ASCII? Are the files written one per block, or are they all concatenated together in one tremendously long string of bytes? We do not even know the block size on the tape, because this variable is generally one that can be chosen separately for each block written.

For a disk drive, the basic operations are `read`, `write`, and `seek`. Tape drives, on the other hand, have a different set of basic operations. Instead of `seek`, a tape drive uses the `locate` operation. The tape `locate` operation is more precise than the disk `seek` operation, because it positions the tape to a specific logical block, rather than an entire track. Locating to block 0 is the same as rewinding the tape.

For most kinds of tape drives, it is possible to locate to any block that has been written on a tape. In a partly filled tape, however, it is not possible to locate into the empty space beyond the written area, because most tape drives manage their physical space differently from disk drives. For a disk drive, the sectors have a fixed size, and the formatting process must be used to place empty sectors in their final positions before any data can be written. Most tape drives have a variable block size, and the size of each block is determined on the fly, when that block is written. If an area of defective tape is encountered during writing, the bad area is skipped and the block is written again. This operation explains why it is not possible to locate into the empty space beyond the written area—the positions and numbers of the logical blocks have not yet been determined.

Most tape drives have a `read` position operation that returns the logical block number where the tape head is. Many tape drives also support a `space` operation for relative motion. So, for example, the operation `space -2` would locate backward over two logical blocks.

For most kinds of tape drives, writing a block has the side effect of logically erasing everything beyond the position of the write. In practice, this side effect means that most tape drives are append-only devices, because updating a block in the middle of the tape also effectively erases everything beyond that block. The tape drive implements this appending by placing an end-of-tape (EOT) mark after a block that is written. The drive refuses to locate past the EOT mark, but it is possible to locate to the EOT and then to start writing. Doing so overwrites the old EOT mark, and places a new one at the end of the new blocks just written.

In principle, a file system can be implemented on a tape. But many of the file-system data structures and algorithms would be different from those used for disks, because of the append-only property of tape.

14.8.2.2 File Naming

Another question that the operating system needs to handle is how to name files on removable media. For a fixed disk, naming is not difficult. On a PC, the file name consists of a drive letter followed by a path name. In UNIX, the file name does not contain a drive letter, but the mount table enables the operating system to discover on what drive the file is located. But if the disk is removable, knowing a drive that contained the cartridge at some time in the past does not mean knowing how to find the file. If every removable cartridge in the world had a different serial number, the name of a file on a removable device could be prefixed with the serial number, but to ensure that no two serial numbers are the same would require each one to be about 12 digits in length. Who could remember the names of her files if she had to memorize a 12-digit serial number for each one?

The problem becomes even more difficult when we want to write data on a removable cartridge on one computer, and then use the cartridge in

another computer. If both machines are of the same type and have the same kind of removable drive, the only difficulty is knowing the contents and data layout on the cartridge. But if the machines or drives are different, many additional problems can arise. Even if the drives are compatible, different computers may store bytes in different orders, and may use different encodings for binary numbers and even for letters (such as ASCII on PCs versus EBCDIC on mainframes).

Today's operating systems generally leave the name-space problem unsolved for removable media, and depend on applications and users to figure out how to access and interpret the data. Fortunately, a few kinds of removable media are so well standardized that all computers use them the same way. One example is the CD. Music CDs use a universal format that is understood by any CD drive. Data CDs are available in only a few different formats, so it is usual for a CD drive and the operating-system device driver to be programmed to handle all the common formats. DVD formats are also well standardized.

14.8.2.3 Hierarchical Storage Management

A **robotic jukebox** enables the computer to change the removable cartridge in a tape or disk drive without human assistance. Two major uses of this technology are for backups and hierarchical storage systems. The use of a jukebox for backups is simple: when one cartridge becomes full, the computer instructs the jukebox to switch to the next cartridge. Some jukeboxes hold tens of drives and thousands of cartridges, with robotic arms managing the movement of tapes to the drives.

A hierarchical storage system extends the storage hierarchy beyond primary memory and secondary storage (that is, magnetic disk) to incorporate tertiary storage. Tertiary storage is usually implemented as a jukebox of tapes or removable disks. This level of the storage hierarchy is larger, cheaper, and probably slower.

Although the virtual-memory system can be extended in a straightforward manner to tertiary storage, this extension is rarely carried out in practice. The reason is that a retrieval from a jukebox can take tens of seconds or even minutes, and such a long delay is intolerable for demand paging and for other forms of virtual-memory use.

The usual way to incorporate tertiary storage is to extend the file system. Small and frequently used files remain on magnetic disk, while large and old files that are not actively used are archived to the jukebox. In some file-archiving systems, the directory entry for the file continues to exist, but the contents of the file no longer occupy space in secondary storage. If an application tries to open the file, the open system call is suspended until the file contents can be staged in from tertiary storage. When the contents are again available from magnetic disk, the open operation returns control to the application, which proceeds to use the disk-resident copy of the data. **Hierarchical storage management (HSM)** has been implemented in ordinary time-sharing

systems such as TOPS-20, which ran on minicomputers from Digital Equipment Corporation in the late 1970s. Today, HSM is usually found in supercomputing centers and other large installations that have enormous volumes of data.

14.8.3 Performance Issues

As with any component of the operating system, the three most important aspects of tertiary-storage performance are speed, reliability, and cost.

14.8.3.1 Speed

The speed of tertiary storage has two aspects: bandwidth and latency. We measure the bandwidth in bytes per second. The **sustained bandwidth** is the average data rate during a large transfer, that is, the number of bytes divided by the transfer time. The **effective bandwidth** calculates the average over the entire I/O time, including the time for seek or locate and any cartridge-switching time in a jukebox. In essence, the sustained bandwidth is the data rate when the data stream is actually flowing, and the effective bandwidth is the overall data rate provided by the drive. The *bandwidth of a drive* is generally understood to mean the sustained bandwidth.

For removable disks, the bandwidth ranges from less than 0.25 MB per second for the slowest, to several megabytes per second for the fastest. Tapes have an even wider range of bandwidths, from less than 0.25 MB per second to over 30 MB per second. The fastest tape drives have significantly higher bandwidth than do removable disk drives.

The second aspect of speed is the **access latency**. By this performance measure, disks are much faster than tapes: Disk storage is essentially two-dimensional—all the bits are out in the open. A disk access simply moves the arm to the selected cylinder and waits for the rotational latency, which may take less than 5 milliseconds. By contrast, tape storage is three-dimensional. At any time, a small portion of the tape is accessible to the head, whereas most of the bits are buried below hundreds or thousands of layers of tape wound on the reel. A random access on tape requires winding the tape reels until the selected block reaches the tape head, which can take tens or hundreds of seconds. So we can generally say that random access within a tape cartridge is more than a thousand times slower than random access on disk.

If a jukebox is involved, the access latency can be significantly higher. For a removable disk to be changed, the drive must stop spinning, then the robotic arm must switch the disk cartridges, and the drive must spin up the new cartridge. This operation takes several seconds—about a hundred times larger than the random-access time within one disk. So switching disks in a jukebox incurs a relatively high performance penalty.

For tapes, the robotic-arm time is about the same as for disk. But for tapes to be switched, the old tape generally must rewind before it can be ejected, and that operation can take as long as 4 minutes. And, after a new tape is loaded

into the drive, many seconds can be required for the drive to calibrate itself to the tape and to prepare for I/O. Although a slow tape jukebox can have a tape switch time of 1 or 2 minutes, this time is not enormously larger than the random-access time within one tape.

So, to generalize, we say that random access in a disk jukebox has a latency of tens of seconds, whereas random access in a tape jukebox has a latency of hundreds of seconds; switching disks is expensive, but switching tapes is not. Be careful not to overgeneralize: Some expensive tape jukeboxes can rewind, eject, load a new tape, and fast forward to a random item of data all in less than 30 seconds.

If we pay attention to only the performance of the drives in a jukebox, the bandwidth and latency seem reasonable. But if we focus our attention on the cartridges instead, there is a terrible bottleneck. Consider first the bandwidth. By comparison with a fixed disk, the bandwidth-to-storage-capacity ratio of a robotic library is much less favorable. To read all the data stored on a large hard disk could take about an hour. To read all the data stored in a large tape library could take years. The situation with respect to access latency is nearly as bad. To illustrate this, if 100 requests are queued for a disk drive, the average waiting time will be about 1 second. If 100 requests are queued for a tape library, the average waiting time could be over 1 hour. The low cost of tertiary storage results from having many cheap cartridges share a few expensive drives. But a removable library is best devoted to the storage of infrequently used data, because the library can satisfy only a relatively small number of I/O requests per hour.

14.8.3.2 Reliability

Although we often think *good performance* means *high speed*, another important aspect of performance is *reliability*. If we try to read some data and are unable to do so because of a drive or media failure, for all practical purposes the access time is infinitely long and the bandwidth is infinitely small. So it is important to understand the reliability of removable media.

Removable magnetic disks are somewhat less reliable than are fixed hard disks because the cartridge is more likely to be exposed to harmful environmental conditions such as dust, large changes in temperature and humidity, and mechanical forces such as shock and bending. Optical disks are considered very reliable, because the layer that stores the bits is protected by a transparent plastic or glass layer. The reliability of magnetic tape varies widely, depending on the kind of drive. Some inexpensive drives wear out tapes after a few dozen uses; other kinds are gentle enough to allow millions of reuses. By comparison with a magnetic disk, the head in a magnetic-tape drive is a weak spot. A disk head flies above the media, but a tape head is in close contact with the tape. The scrubbing action of the tape can wear out the head after a few thousands or tens of thousands of hours.

In summary, we say that a fixed disk drive is likely to be more reliable than a removable disk or tape drive, and an optical disk is likely to be more reliable than a magnetic disk or tape. But a fixed magnetic disk has one weakness. A head crash in a hard disk generally destroys the data, whereas the failure of a tape drive or optical disk drive often leaves the data cartridge unharmed.

14.8.3.3 Cost

Storage cost is another important factor. Here is a concrete example of how removable media may lower the overall storage cost. Suppose that a hard disk that holds X GB has a price of \$200; of this amount, \$190 is for the housing, motor, and controller, and \$10 is for the magnetic platters. Then, the storage cost for this disk is $\$200/X$ per gigabyte. Now, suppose that we can manufacture the platters in a removable cartridge. For one drive and 10 cartridges, the total price is \$190 + \$100 and the capacity is $10X$ GB, so the storage cost is $\$29/X$ per gigabyte. Even if it is a little more expensive to make a removable cartridge, the cost per gigabyte of removable storage may well be lower than the cost per gigabyte of a hard disk, because the expense of one drive is averaged with the low price of many removable cartridges.

Figures 14.13, 14.14, and 14.15 show the cost trends per megabyte for DRAM memory, magnetic hard disks, and tape drives. The prices in the graphs are the lowest price found in advertisements in *BYTE* magazine and *PC Magazine* at the end of each year. These prices reflect the small-computer marketplace of the

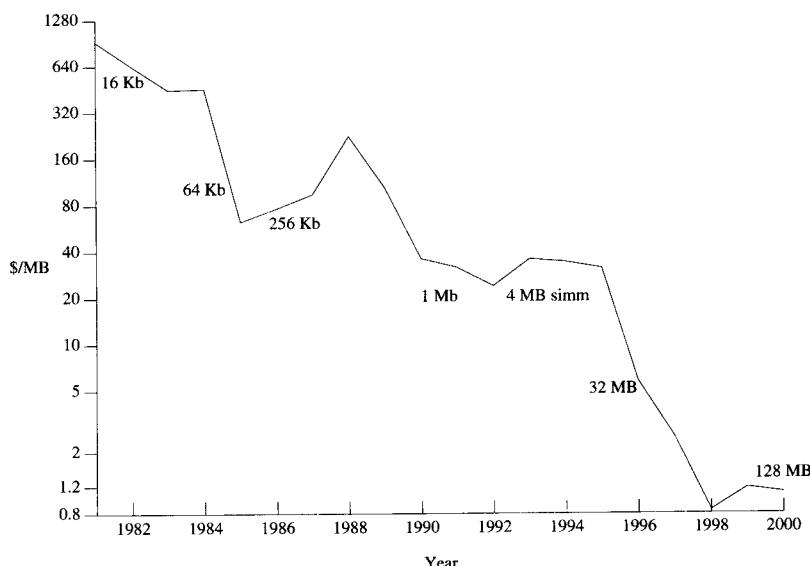


Figure 14.13 Price per megabyte of DRAM, from 1981 to 2000.

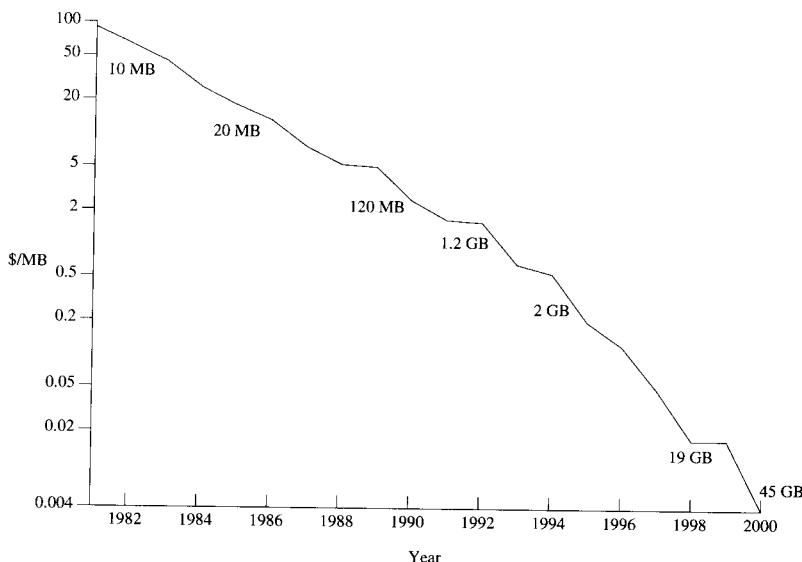


Figure 14.14 Price per megabyte of magnetic hard disk, from 1981 to 2000.

readership of these magazines, where prices are low by comparison with the mainframe and minicomputer markets. In the case of tape, the price is for a drive with one tape. The overall cost of tape storage becomes much lower as more tapes are purchased for use with the drive, because the price of a tape is a small fraction of the price of the drive. However, in a huge tape library containing thousands of cartridges, the storage cost is dominated by the cost of the tape cartridges. As of this writing in 2001, the cost per GB of tape cartridges can be approximated as \$2.

The cost of DRAM fluctuates widely. In the period from 1981 to 2000, we can see three price crashes (around 1981, 1989, and 1996), as excess production caused a glut in the marketplace. We can also see two periods (around 1987 and 1993), where shortages in the marketplace caused significant price increases. In the case of hard disks, the price declines have been much steadier, although the price decline appears to have accelerated since 1992. Tape-drive prices also fell steadily up to 1997. Since 1997 the price per gigabyte of inexpensive tape drives has ceased its dramatic fall, although mid-range tape technology (such as DAT/DDS) has continued to fall, and is now approaching that of the inexpensive drives. Tape-drive prices are not shown prior to 1984, because *BYTE* magazine is targeted to the small-computer marketplace, and tape drives were not widely used with small computers prior to 1984.

By comparing these graphs we see that the price of disk storage has plummeted relative to the price of DRAM and tape.

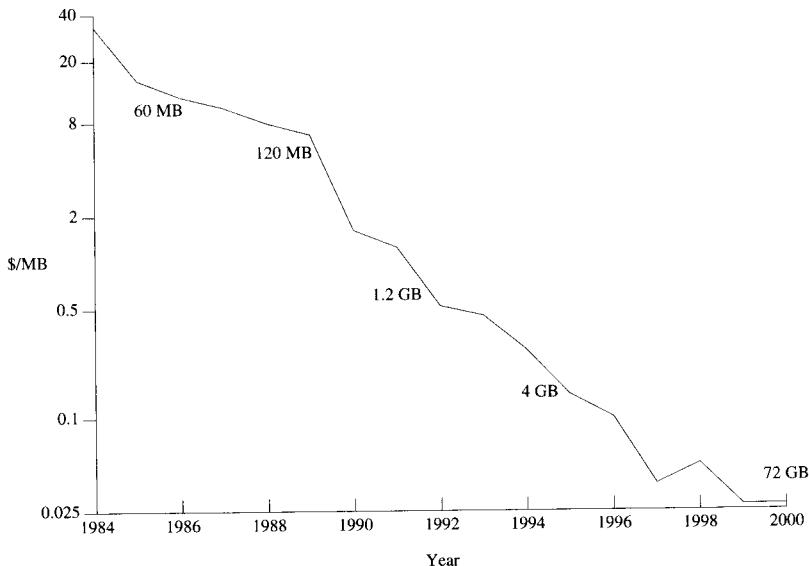


Figure 14.15 Price per megabyte of a tape drive, from 1984 to 2000.

The price per megabyte of magnetic disk has improved by more than four orders of magnitude during the past two decades, whereas the corresponding improvement for main memory has only been three orders of magnitude. Main memory today is more expensive than disk storage by a factor of 100.

The price per megabyte has dropped much more rapidly for disk drives than for tape drives. In fact, the price per megabyte of magnetic disk drives is approaching that of a tape cartridge without the tape drive. Consequently, small- and medium-size tape libraries have a higher storage cost than disk systems with equivalent capacity. The dramatic fall in disk prices has largely rendered tertiary storage obsolete: We no longer have any tertiary storage technology that is orders of magnitude less expensive than magnetic disk. It appears that the revival of tertiary storage must await a revolutionary technology breakthrough. Meanwhile, tape storage will find its use mostly limited to purposes such as backups of disk drives and archival storage in enormous tape libraries that greatly exceed the practical storage capacity of large disk farms.

14.9 ■ Summary

Disk drives are the major secondary-storage I/O device on most computers. Requests for disk I/O are generated by the file system and by the virtual-memory system. Each request specifies the address on the disk to be referenced, in the form of a logical block number.

Disk-scheduling algorithms can improve the effective bandwidth, the average response time, and the variance in response time. Algorithms such as SSTF, SCAN, C-SCAN, LOOK, and C-LOOK are designed to make such improvements by strategies for disk-queue ordering.

Performance can be harmed by external fragmentation. Some systems have utilities that scan the file system to identify fragmented files; they then move blocks around to decrease the fragmentation. Defragmenting a badly fragmented file system can significantly improve the performance, but the system may have reduced performance while the defragmentation is in progress. Sophisticated file systems, such as the UNIX Fast File System, incorporate many strategies to control fragmentation during space allocation, so that disk reorganization is not needed.

The operating system manages the disk blocks. First, a disk must be low-level formatted to create the sectors on the raw hardware—new disks usually come pre-formatted. Then, the disk is partitioned and file systems created, and boot blocks are allocated to store the system’s bootstrap program. Finally, when a block is corrupted, the system must have a way to lock out that block, or to replace it logically with a spare.

Because an efficient swap space is a key to good performance, systems usually bypass the file system and use raw disk access for paging I/O. Some systems dedicate a raw disk partition to swap space, and others use a file within the file system instead. Other systems allow the user or system administrator to make the decision by providing both options.

The write-ahead log scheme requires the availability of stable storage. To implement such storage, we need to replicate the needed information on multiple nonvolatile storage devices (usually disks) with independent failure modes. We also need to update the information in a controlled manner to ensure that we can recover the stable data after any failure during data transfer or recovery.

Because of the amount of storage required on large systems, disks are frequently made redundant via RAID algorithms. These algorithms allow more than one disk to be used for a given operation, and allow continued operation and even automatic recovery in the face of a disk failure. RAID algorithms are organized into different levels where each level provides some combination of reliability and high transfer rates.

Disk may be attached to a computer system one of two ways: (1) using the local I/O ports on the host computer or (2) using a network connection such as storage area networks.

Tertiary storage is built from disk and tape drives that use removable media. Many different technologies are available, including magnetic tape, removable magnetic and magneto-optic disks, and optical disks.

For removable disks, the operating system generally provides the full services of a file-system interface, including space management and request-queue scheduling. For many operating systems, the name of a file on a

removable cartridge is a combination of a drive name and a file name within that drive. This convention is simpler but potentially more confusing than is using a name that identifies a specific cartridge.

For tapes, the operating system generally just provides a raw interface. Many operating systems have no built-in support for jukeboxes. Jukebox support can be provided by a device driver or by a privileged application designed for backups or for HSM.

Three important aspects of performance are bandwidth, latency, and reliability. A wide variety of bandwidths is available for both disks and tapes, but the random-access latency for a tape is generally much slower than that for a disk. Switching cartridges in a jukebox is also relatively slow. Because a jukebox has a low ratio of drives to cartridges, reading a large fraction of the data in a jukebox can take a long time. Optical media, which protect the sensitive layer by a transparent coating, are generally more robust than magnetic media, which expose the magnetic material to a greater possibility of physical damage.

■ Exercises

- 14.1 None of the disk-scheduling disciplines, except FCFS, are truly *fair* (starvation may occur).
- Explain why this assertion is true.
 - Describe a way to modify algorithms such as SCAN to ensure fairness.
 - Explain why fairness is an important goal in a time-sharing system.
 - Give three or more examples of circumstances in which it is important that the operating system be *unfair* in serving I/O requests.
- 14.2 Suppose that a disk drive has 5,000 cylinders, numbered 0 to 4999. The drive is currently serving a request at cylinder 143, and the previous request was at cylinder 125. The queue of pending requests, in FIFO order, is

86, 1470, 913, 1774, 948, 1509, 1022, 1750, 130.

Starting from the current head position, what is the total distance (in cylinders) that the disk arm moves to satisfy all the pending requests for each of the following disk-scheduling algorithms?

- FCFS
- SSTF
- SCAN

- d. LOOK
- e. C-SCAN
- f. C-LOOK

14.3 Elementary physics states that when an object is subjected to a constant acceleration a , the relationship between distance d and time t is given by $d = \frac{1}{2}at^2$. Suppose that, during a seek, the disk in Exercise 14.2 accelerates the disk arm at a constant rate for the first half of the seek, then decelerates the disk arm at the same rate for the second half of the seek. Assume that the disk can perform a seek to an adjacent cylinder in 1 millisecond, and a full-stroke seek over all 5,000 cylinders in 18 milliseconds.

- a. The distance of a seek is the number of cylinders that the head moves. Explain why the seek time is proportional to the square root of the seek distance.
- b. Write an equation for the seek time as a function of the seek distance. This equation should be of the form $t = x + y\sqrt{L}$, where t is the time in milliseconds and L is the seek distance in cylinders.
- c. Calculate the total seek time for each of the schedules in Exercise 14.2. Determine which schedule is the fastest (has the smallest total seek time).
- d. The *percentage speedup* is the time saved divided by the original time. What is the percentage speedup of the fastest schedule over FCFS?

14.4 Suppose that the disk in Exercise 14.3 rotates at 7,200 RPM.

- a. What is the average rotational latency of this disk drive?
- b. What seek distance can be covered in the time that you found for part a?

14.5 The accelerating seek described in Exercise 14.3 is typical of hard-disk drives. By contrast, floppy disks (and many hard disks manufactured before the mid-1980s) typically seek at a fixed rate. Suppose that the disk in Exercise 14.3 has a constant-rate seek, rather than a constant-acceleration seek, so the seek time is of the form $t = x + yL$, where t is the time in milliseconds and L is the seek distance. Suppose that the time to seek to an adjacent cylinder is 1 millisecond, as before, and is 0.5 milliseconds for each additional cylinder.

- a. Write an equation for this seek time as a function of the seek distance.

- b. Using the seek-time function from part a, calculate the total seek time for each of the schedules in Exercise 14.2. Is your answer the same as it was for Exercise 14.3c? Explain why it is the same or why it is different.
 - c. What is the percentage speedup of the fastest schedule over FCFS in this case?
- 14.6** Write a Java program for disk scheduling using the SCAN and C-SCAN disk-scheduling algorithms.
- 14.7** Compare the performance of C-SCAN and SCAN scheduling, assuming a uniform distribution of requests. Consider the average response time (the time between the arrival of a request and the completion of that request's service), the variation in response time, and the effective bandwidth. How does performance depend on the relative sizes of seek time and rotational latency?
- 14.8** Is disk scheduling, other than FCFS scheduling, useful in a single-user environment? Explain your answer.
- 14.9** Explain why SSTF scheduling tends to favor middle cylinders over the innermost and outermost cylinders.
- 14.10** Requests are not usually uniformly distributed. For example, a cylinder containing the file system FAT or inodes can be expected to be accessed more frequently than a cylinder that contains only files. Suppose that you know that 50 percent of the requests are for a small, fixed number of cylinders.
- a. Would any of the scheduling algorithms discussed in this chapter be particularly good for this case? Explain your answer.
 - b. Propose a disk-scheduling algorithm that gives even better performance by taking advantage of this "hot spot" on the disk.
 - c. File systems typically find data blocks via an indirection table, such as a FAT in DOS or inodes in UNIX. Describe one or more ways to take advantage of this indirection to improve the disk performance.
- 14.11** Why is rotational latency usually not considered in disk scheduling? How would you modify SSTF, SCAN, and C-SCAN to include latency optimization?
- 14.12** How would the use of a RAM disk affect your selection of a disk-scheduling algorithm? What factors would you need to consider? Do the same considerations apply to hard-disk scheduling, given that the file system stores recently used blocks in a buffer cache in main memory?

- 14.13** Why is it important to balance file system I/O among the disks and controllers on a system in a multitasking environment?
- 14.14** What are the tradeoffs involved in rereading code pages from the file system, versus using swap space to store them?
- 14.15** Is there any way to implement truly stable storage? Explain your answer.
- 14.16** The reliability of a hard-disk drive is typically described in terms of a quantity called *mean time between failures (MTBF)*. Although this quantity is called a “time,” the MTBF actually is measured in drive-hours per failure.
- If a system contains 1,000 disk drives, each of which has a 750,000 hour MTBF, which of the following best describes how often a drive failure will occur in that disk farm: once per thousand years, once per century, once per decade, once per year, once per month, once per week, once per day, once per hour, once per minute, or once per second?
 - Mortality statistics indicate that, on the average, a U.S. resident has about 1:1,000 chance of dying between ages 20 and 21 years. Deduce the MTBF hours for 20 year olds. Convert this figure from hours to years. What does this MTBF tell you about the expected lifetime of a 20 year old?
 - The manufacturer guarantees a 1-million hour MTBF for a certain model of disk drive. What can you conclude about the number of years for which one of these drives is under warranty?
- 14.17** The term *fast wide SCSI-II* denotes a SCSI bus that operates at a data rate of 20 MB per second when it moves a packet of bytes between the host and a device. Suppose that a fast wide SCSI-II disk drive spins at 7,200 RPM, has a sector size of 512 bytes, and holds 160 sectors per track.
- Estimate the sustained transfer rate of this drive in megabytes per second.
 - Suppose that the drive has 7,000 cylinders, 20 tracks per cylinder, a head-switch time (from one platter to another) of 0.5 milliseconds, and an adjacent-cylinder seek time of 2 milliseconds. Use this additional information to give an accurate estimate of the sustained transfer rate for a huge transfer.
 - Suppose that the average seek time for the drive is 8 milliseconds. Estimate the I/Os per second and the effective transfer rate for a random-access workload that reads individual sectors scattered across the disk.

- d. Calculate the random-access I/Os per second and transfer rate for I/O sizes of 4 KB, 8 KB, and 64 KB.
 - e. If multiple requests are in the queue, a scheduling algorithm such as SCAN should be able to reduce the average seek distance. Suppose that a random-access workload is reading 8 KB pages, the average queue length is 10, and the scheduling algorithm reduces the average seek time to 3 milliseconds. Calculate the I/Os per second and the effective transfer rate of the drive.
- 14.18** More than one disk drive can be attached to a SCSI bus. In particular, a fast wide SCSI-II bus (Exercise 14.17) can be connected to at most 15 disk drives. Recall that this bus has a bandwidth of 20 MB per second. At any time, only one packet can be transferred on the bus between some disk's internal cache and the host. However, a disk can be moving its disk arm while some other disk is transferring a packet on the bus. Also, a disk can be transferring data between its magnetic platters and its internal cache while some other disk is transferring a packet on the bus. Considering the transfer rates that you calculated for the various workloads in Exercise 14.17, discuss how many disks can be used effectively by one fast wide SCSI-II bus.
- 14.19** Remapping of bad blocks by sector sparing or sector slipping could influence performance. Suppose that the drive in Exercise 14.17 has a total of 100 bad sectors at random locations, and that each bad sector is mapped to a spare that is located on a different track, but within the same cylinder. Estimate the number of I/Os per second and the effective transfer rate for a random-access workload consisting of 8 KB reads, with a queue length of 1 (that is, the choice of scheduling algorithm is not a factor). What is the effect of a bad sector on performance?
- 14.20** Discuss the relative advantages and disadvantages of sector sparing and sector slipping.
- 14.21** The operating system generally treats removable disks as shared file systems, but assigns a tape drive to only one application at a time. Give three reasons that could explain this difference in treatment of disks and tapes. Describe the additional features that an operating system would need to support shared file-system access to a tape jukebox. Would the applications sharing the tape jukebox need any special properties, or could they use the files as though the files were disk-resident? Explain your answer.
- 14.22** In a disk jukebox, what would be the effect if the number of open files was greater than the number of drives in the jukebox?

- 14.23 What would be the effects on cost and performance if tape storage had the same areal density as disk storage?
- 14.24 If magnetic hard disks eventually have the same cost per gigabyte as do tapes, will tapes become obsolete, or will they still be needed? Explain your answer.
- 14.25 You can use simple estimates to compare the cost and performance of a terabyte storage system made entirely from disks with one that incorporates tertiary storage. Suppose that magnetic disks each hold 10 GB, cost \$1,000, transfer 5 MB per second, and have an average access latency of 15 milliseconds. Suppose that a tape library costs \$10 per gigabyte, transfers 10 MB per second, and has an average access latency of 20 seconds. Compute the total cost, the maximum total data rate, and the average waiting time for a pure disk system. If you make any assumptions about the workload, describe and justify them. Now, suppose that 5 percent of the data are frequently used, so they must reside on disk, but the other 95 percent are archived in the tape library. Further suppose that the disk system handles 95 percent of the requests, and the library handles the other 5 percent. What are the total cost, the maximum total data rate, and the average waiting time for this hierarchical storage system?
- 14.26 It is sometimes said that tape is a sequential-access medium, whereas magnetic disk is a random-access medium. In fact, the suitability of a storage device for random access depends on the transfer size. The term *streaming transfer rate* denotes the data rate for a transfer underway, excluding the effect of access latency. By contrast, the *effective transfer rate* is the ratio of total bytes per total seconds, including overhead time such as the access latency.

Suppose that, in a computer, the level-2 cache has an access latency of 8 nanoseconds and a streaming transfer rate of 800 MB per second, the main memory has an access latency of 60 nanoseconds and a streaming transfer rate of 80 MB per second, the magnetic disk has an access latency of 15 millisecond and a streaming transfer rate of 5 MB per second, and a tape drive has an access latency of 60 seconds and a streaming transfer rate of 2 MB per second.

- a. Random access causes the effective transfer rate of a device to decrease, because no data are transferred during the access time. For the disk described, what is the effective transfer rate if a streaming transfer of 512 bytes, 8 KB, 1 MB, and 16 MB follows an average access?
- b. The utilization of a device is the ratio of effective transfer rate to streaming transfer rate. Calculate the utilization of the disk drive

for random access that performs transfers in each of the four sizes given in part a.

- c. Suppose that a utilization of 25 percent (or higher) is considered acceptable. Using the performance figures given, compute the smallest transfer size for disk that gives acceptable utilization.
- d. Complete the following sentence: A disk is a random-access device for transfers larger than _____ bytes, and is a sequential-access device for smaller transfers.
- e. Compute the minimum transfer sizes that give acceptable utilization for cache, memory, and tape.
- f. When is a tape a random-access device, and when is it a sequential-access device?

14.27 Imagine that a holographic storage drive has been invented. Suppose that a holographic drive costs \$10,000 and has an average access time of 40 milliseconds. Suppose that it uses a \$100 cartridge the size of a CD. This cartridge holds 40,000 images, and each image is a square black-and-white picture with resolution $6,000 \times 6,000$ pixels (each pixel stores 1 bit). Suppose that the drive can read or write one picture in 1 millisecond. Answer the following questions.

- a. What would be some good uses for this device?
- b. How would this device affect the I/O performance of a computing system?
- c. Which other kinds of storage devices, if any, would become obsolete as a result of this device being invented?

14.28 Suppose that a one-sided 5.25-inch optical-disk cartridge has an areal density of 1 gigabit per square inch. Suppose that a magnetic tape has an areal density of 20 megabits per square inch, and is 1/2 inch wide and 1,800 feet long. Calculate an estimate of the storage capacities of these two kinds of storage cartridges. Suppose that an optical tape exists that has the same physical size as the tape, but the same storage density as the optical disk. What volume of data could the optical tape hold? What would be a marketable price for the optical tape if the magnetic tape cost \$25?

14.29 Suppose that we agree that 1 KB is 1,024 bytes, 1 MB is $1,024^2$ bytes, and 1 GB is $1,024^3$ bytes. This progression continues through terabytes, petabytes, and exabytes ($1,024^6$). Several newly proposed scientific projects plan to be able to record and store a few exabytes of data during the next decade. To answer the following questions, you will need to

make a few reasonable assumptions; state the assumptions that you make.

- a. How many disk drives would be required to hold 4 exabytes of data?
- b. How many magnetic tapes would be required to hold 4 exabytes of data?
- c. How many optical tapes would be required to hold 4 exabytes of data (Exercise 14.28)?
- d. How many holographic storage cartridges would be required to hold 4 exabytes of data (Exercise 14.27)?
- e. How many cubic feet of storage space would each option require?

- 14.30 Discuss how an operating system could maintain a free-space list for a tape-resident file system. Assume that the tape technology is append-only, and that it uses the EOT mark and `locate`, `space`, and `read` position commands as described in Section 14.8.2.1.

Bibliographical Notes

Discussions of redundant arrays of independent disks (RAID) are presented by Patterson et al. [1988] and in the detailed survey of Chen et al. [1994]. Disk-system architectures for high-performance computing are discussed by Katz et al. [1989]. Teorey and Pinkerton [1972] present an early comparative analysis of disk-scheduling algorithms. They use simulations that model a disk for which seek time is linear in the number of cylinders crossed. For this disk, LOOK is a good choice for queue lengths below 140, and C-LOOK is good for queue lengths above 100. King [1990] describes ways to improve the seek time by moving the disk arm when the disk is otherwise idle. Seltzer et al. [1990] describe disk-scheduling algorithms that consider rotational latency in addition to seek time. Worthington et al. [1994] discuss disk performance, and show the negligible performance impact of defect management. The placement of hot data to improve seek times has been considered by Ruemmler and Wilkes [1991] and Akyurek and Salem [1993]. Ruemmler and Wilkes [1994] describe an accurate performance model for a modern disk drive. Worthington et al. [1995] tell how to determine low-level disk properties such as the zone structure, and this work is further advanced by Schindler and Gregory [1999].

The I/O size and randomness of the workload has a considerable influence on disk performance. Ousterhout et al. [1985] and Ruemmler and Wilkes [1993] report numerous interesting workload characteristics, including that most files are small, most newly created files are deleted soon thereafter, most files that are opened for reading are read sequentially in their entirety, and most seeks

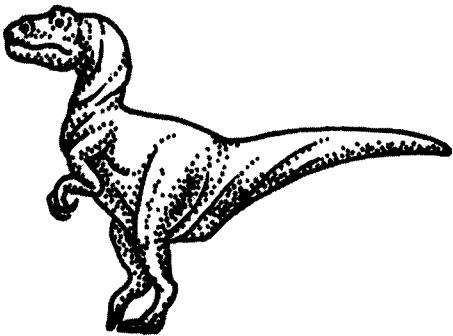
are short. McKusick et al. [1984] describe the Berkeley Fast File System, which uses many sophisticated techniques to obtain good performance for a wide variety of workloads. McVoy and Kleiman [1991] discuss further improvements to the basic FFS. Quinlan [1991] describes how to implement a file system on WORM storage with a magnetic disk cache; Richards [1990] discusses a file-system approach to tertiary storage. Maher et al. [1994] give an overview of the integration of distributed file systems and tertiary storage.

The concept of a storage hierarchy has been studied for more than a quarter of a century. For instance, a 1970 paper by Mattson et al. [1970] describes a mathematical approach to predict the performance of a storage hierarchy. Alt [1993] describes the accommodation of removable storage in a commercial operating system, and Miller and Katz [1993] describe the characteristics of tertiary-storage access in a supercomputing environment. Benjamin [1990] gives an overview of the massive storage requirements for the EOSDIS project at NASA.

Holographic-storage technology is the subject of an article by Psaltis and Mok [1995]; a collection of holographic-storage papers dating from 1963 has been assembled by Sincerbox [1994]. Asthana and Finkelstein [1995] describe several emerging storage technologies, including holographic storage, optical tape, and electron trapping. Toigo [2000] gives an in-depth description of modern disk technology and several potential future storage technologies.

Part Five

DISTRIBUTED SYSTEMS



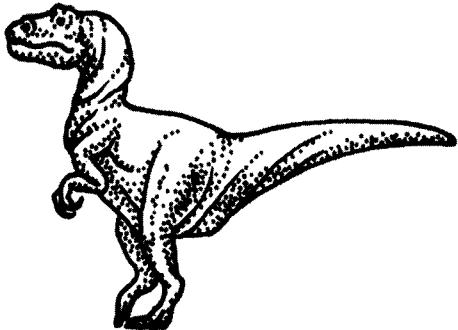
A distributed system is a collection of processors that do not share memory or a clock. Instead, each processor has its own local memory, and the processors communicate with each other through communication lines such as local- or wide-area networks. The processors in a distributed system vary in size and function. Such systems may include small handheld or real-time devices, personal computers, workstations, and large mainframe computer systems.

The benefits of a distributed system include user access to the resources maintained by the system and therefore computation speedup and improved data availability and reliability. A distributed file system is a file-service system whose users, servers, and storage devices are dispersed among the sites of a distributed system. Accordingly, service activity has to be carried out across the network; instead of a single centralized data repository, there are multiple and independent storage devices.

Because a system is distributed, however, it must provide mechanisms for process synchronization and communication, for dealing with the deadlock problem, and for dealing with failures that are not encountered in a centralized system.

Chapter 15

DISTRIBUTED SYSTEM STRUCTURES



A distributed system is a collection of processors that do not share memory or a clock. Instead, each processor has its own local memory. The processors communicate with one another through various communication networks, such as high-speed buses or telephone lines. In this chapter, we discuss the general structure of distributed systems and the networks that interconnect them. We contrast the main differences in operating-system design between these systems and the centralized systems with which we were concerned previously. Detailed discussions are given in Chapters 16 and 17.

15.1 ■ Background

A **distributed system** is a collection of loosely coupled processors interconnected by a **communication network**. From the point of view of a specific processor in a distributed system, the rest of the processors and their respective resources are **remote**, whereas its own resources are local.

The processors in a distributed system may vary in size and function. They may include small microprocessors, workstations, minicomputers, and large general-purpose computer systems. These processors are referred to by a number of names, such as *sites*, *nodes*, *computers*, *machines*, or *hosts*, depending on the context in which they are mentioned. We mainly use *site* to indicate the location of a machine, and *host* to refer to a specific system at a site. Generally, one host at one site, the *server*, has a resource that another host at another site, the *client* (or user), would like to use. The purpose of the distributed system is to