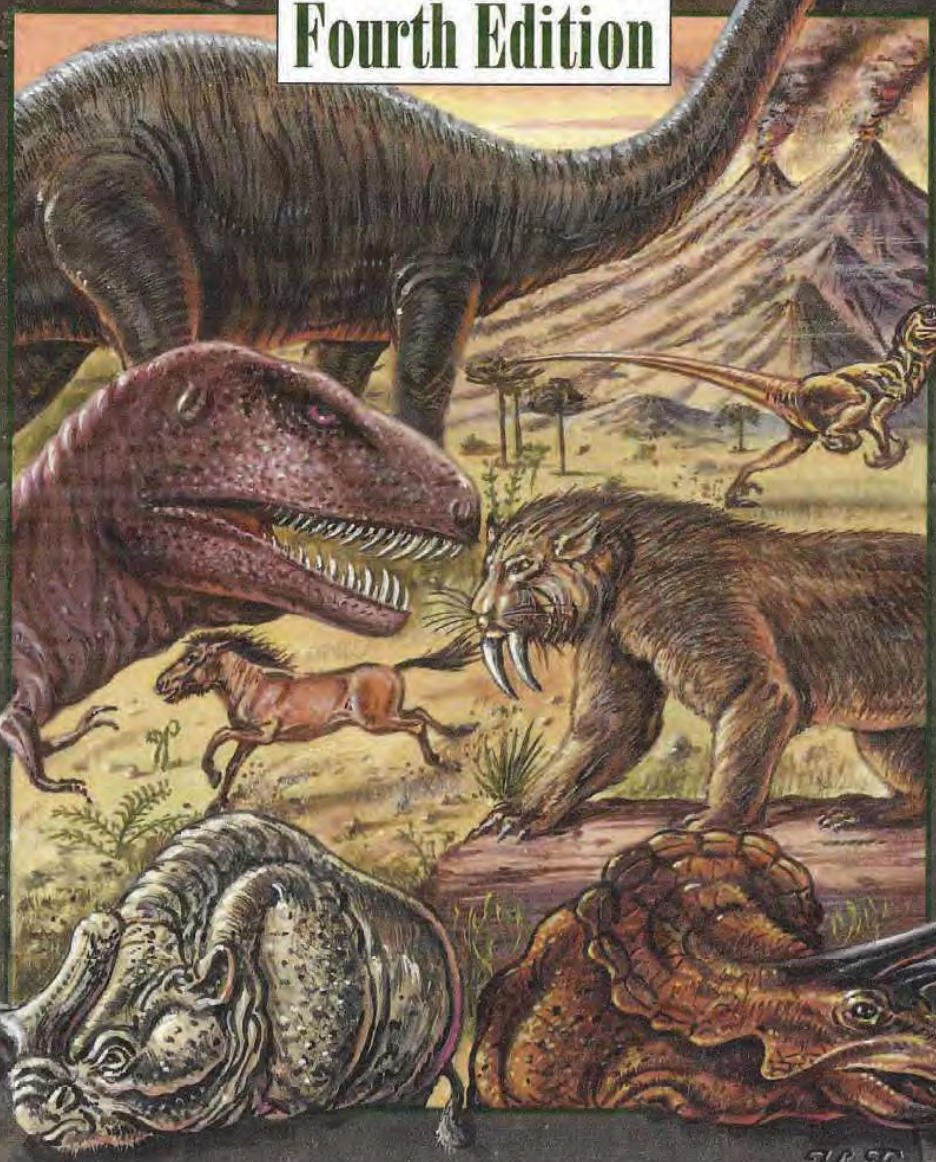
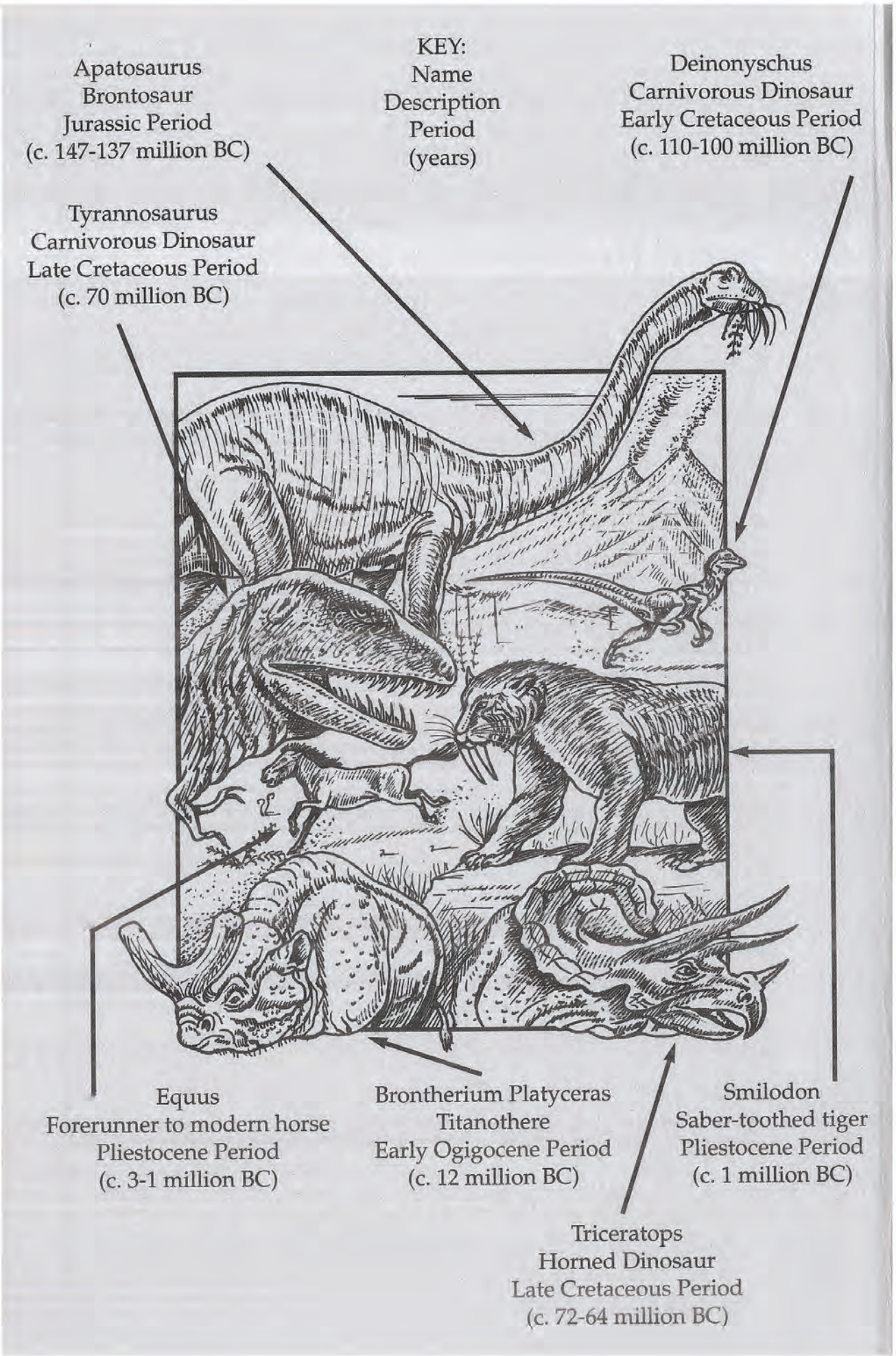
OPERATING SYSTEM CONCEPTS

Fourth Edition



SILBERSCHATZ
GALVIN

EX-1026
U.S. Patent 8,793,336


KEY:
Name
Description
Period
(years)
Apatosaurus
Brontosaur
Jurassic Period
(c. 147-137 million BC)
Deinonyschus
Carnivorous Dinosaur
Early Cretaceous Period
(c. 110-100 million BC)
Tyrannosaurus
Carnivorous Dinosaur
Late Cretaceous Period
(c. 70 million BC)
Equus
Forerunner to modern horse
Pliestocene Period
(c. 3-1 million BC)
Brontherium Platyceras
Titanothere
Early Ogigocene Period
(c. 12 million BC)
Smilodon
Saber-toothed tiger
Pliestocene Period
(c. 1 million BC)
Triceratops
Horned Dinosaur
Late Cretaceous Period
(c. 72-64 million BC)

FOURTH EDITION



OPERATING SYSTEM CONCEPTS

Abraham Silberschatz
University of Texas

Peter B. Galvin
Brown University

Addison-Wesley Publishing Company
Reading, Massachusetts • Menlo Park, California • New York
Don Mills, Ontario • Wokingham, England • Amsterdam • Bonn
Sydney • Singapore • Tokyo • Madrid • San Juan • Milan • Paris

Sponsoring Editor: *Deborah Lafferty*
Senior Editor: *Tom Stone*
Senior Production Supervisor: *Helen Wythe*
Marketing Manager: *Phyllis Cerys*
Technical Art Coordinator: *Susan London-Payne*
Cover and Endpaper Designer: *Howard S. Friedman*
Manufacturing Manager: *Roy Logan*

Many of the designations used by manufacturers and sellers to distinguish their products are claimed as trademarks. Where those designations appear in this book, and Addison-Wesley was aware of a trademark claim, the designations have been printed in initial caps or all caps.

The procedures and applications presented in this book have been included for their instructional value. They have been tested with care but are not guaranteed for any particular purpose. The publisher does not offer any warranties or representations, nor does it accept any liabilities with respect to the programs or applications.

Library of Congress Cataloging-in-Publication Data

Silberschatz, Abraham.
Operating system concepts / Abraham Silberschatz, Peter B. Galvin.
p. cm.
Includes bibliographical references and index.
ISBN 0-201-50480-4
1. Operating systems (Computers) I. Galvin, Peter B. II. Title.
QA76.76.063S5583 1994
005.4'3--dc20 93-24415
CIP

Reproduced by Addison-Wesley from camera-ready copy supplied by the authors.

Reprinted with corrections June, 1994

Copyright © 1994 by Addison-Wesley Publishing Company, 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, or otherwise, without the prior written permission of the publisher. Printed in the United States of America.

ISBN 0-201-50480-4
3 4 5 6 7 8 9 10-MA-9594

To my parents, Wira and Mietek,
my wife, Haya,
and my children, Lemor, Sivan and Aaron.

Avi Silberschatz

To Carla and Gwendolyn.

Peter Galvin

PREFACE

Operating systems are an essential part of a computer system. Similarly, a course on operating systems is an essential part of a computer-science education. This book is intended as a text for an introductory course in operating systems at the junior or senior undergraduate level, or first-year graduate level. It provides a clear description of the *concepts* that underlie operating systems.

This book does not concentrate on any particular operating system or hardware. Instead, it discusses fundamental concepts that are applicable to a variety of systems. We do, however, present a large number of examples that pertain to UNIX and other popular operating systems. In particular, we use Sun Microsystem's Solaris 2 operating system, a version of UNIX, which recently has been transformed into a modern operating system with support for threads at the kernel and user levels, symmetric multiprocessing, and real-time scheduling. Other examples used include Microsoft MS-DOS, Windows, and Windows/NT, IBM OS/2, the Apple Macintosh Operating System, and DEC VMS and TOPS-20, among others.

Prerequisites

As prerequisites, we assume that the reader is familiar with general computer organization and a high-level language, such as PASCAL. The hardware topics required for an understanding of operating systems are included in Chapter 2. We use pseudo-PASCAL notation for code examples, but the algorithms can be understood without a thorough knowledge of PASCAL.

Content of this Book

The text is organized in six major parts:

- **Overview** (Chapters 1 to 3). These chapters 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 lower-level classes who want to learn what an operating system is, without getting into the details of the internal algorithms. Additionally, Chapter 2 covers the hardware topics which are important to an understanding of operating systems. Readers well-versed in hardware topics, including I/O, DMA, and hard disk operation, may chose to skim or skip this chapter.

- **Process management** (Chapters 4 to 7). The process concept and concurrency are at the very 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 various methods for process scheduling, interprocess communication, process synchronization, and deadlock handling. Also included under this topic is a discussion of threads.

- **Storage management** (Chapters 8 to 12). A process must be in main memory (at least partially) 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 particular situation. Since main memory is usually too small to accommodate all data and programs and 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 deal with the classic internal algorithms and structures of storage management. They provide a firm practical understanding of the algorithms used — the properties, advantages, and disadvantages.

- **Protection and security** (Chapters 13 and 14). The various processes in an operating system must be protected from one another's activities. For that purpose, mechanisms exist that can be used to 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. 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, together with some 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.

- **Distributed systems** (Chapters 15 to 18). A *distributed system* is a collection of processors that do not share memory or a clock. Such a system provides the user with access to the various resources 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 various 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.

- **Case studies** (Chapters 19 to 21). The various concepts described in this book can be drawn together by describing real operating systems. Two UNIX-based operating systems are covered in detail — Berkeley 4.3BSD and Mach. These operating systems were chosen in part because UNIX at one time was almost small enough to understand and yet was not a toy operating system. Most of its internal algorithms were selected for *simplicity*, not for speed or sophistication. UNIX is readily available to computer-science departments, so many students have access to it. Mach provides an opportunity for us to study a modern operating system that provides compatibility with 4.3BSD but has a drastically different design and implementation. Chapter 21 briefly describes some of the most influential operating systems.

- **The Nachos System** (Appendix). A good way to gain a deeper understanding of modern operating systems concepts is for the 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 impact of those changes. The Nachos instructional operating system, which is briefly described in the Appendix, provides the

opportunity to see how the basic concepts introduced in this text can be used to solve real-world problems. The Nachos system was developed by Professor Thomas Anderson from the University of California at Berkeley, to complement the third edition of this text, and it is freely available in the public domain via the Internet. Reviewers, who have used the Nachos project at other universities, call it a practical and positive supplement.

The Fourth Edition

Many comments and suggestions were forwarded to us concerning our previous editions. These, together with our own observations, have prodded us to produce this fourth edition. Our basic procedure was to reorganize and rewrite the material in each chapter, adding new information, examples, and diagrams where appropriate. We also brought older material up to date and removed material that was no longer of interest. Finally, we improved the exercises and updated the references.

Substantive revisions were made in the following chapters:

- **Chapter 1.** We have condensed some of the material related to older systems and have expanded our discussion of parallel, distributed, and real-time systems.
- **Chapter 2.** We collected coverage of strictly-hardware topics from the other chapters and reorganized them here, making this material easier to skip if it is already understood, and easier to use as a reference. We also expanded discussion of I/O topics, caching, and protection.
- **Chapter 4.** This chapter introduces the process concept. The material in this chapter appeared in parts of old Chapters 4 and 5. We moved the IPC material from old Chapter 5 to Chapter 4, since we believe that the material should be covered as part of the discussion on the process concept rather that as part of the process coordination chapter. We also expanded our discussion on threads considerably, and included Solaris 2 threads as an example.
- **Chapter 5.** This chapter is a reorganized old Chapter 4. It now deals primarily with CPU scheduling issues.
- **Chapter 6.** This chapter is a reorganized old Chapter 5. We removed Eisenberg and McGuire's solution to the critical-section problem for n processes from the main text (it is now an exercise). We also condensed the discussions concerning the critical region concept. We added new material on atomic transactions, including write-ahead logging and concurrency control schemes. Synchronization in Solaris 2 is included as an example.

- **Chapters 8 and 9.** We have added new material on up-to-date computer architectures that support paging and segmentation for large address spaces. Segmentation and paging are illuminated by an OS/2 example.
- **Chapters 10, 11, and 12.** We have expanded the material and completely reorganized the presentation of the file-system concept and implementation. We now present the logical aspect of the file system in Chapter 10, the implementation issues in Chapter 11, and the underlying secondary storage system in Chapter 12. We also have added new material on swap space, stable storage, recovery, reliability and performance.
- **Chapters 13 and 14.** We have separated old Chapter 11 into two chapters — one dealing with protection issues (Chapter 13), the other dealing with security issues (Chapter 14). In each of these chapters, we have reorganized the material, and have added new information. Major expansions include coverage of the Internet Worm and viruses.
- **Chapters 15 and 16.** We have separated old Chapter 12 into two chapters — one dealing with network structures (Chapter 15), the other dealing with distributed system structure (Chapter 16). In each of these chapters, we have reorganized the material, and have added new information. Major expansions include coverage of network protocols and functionality, remote services, thread-management, and the Open Software Foundation's Distributed Computing Environment (DCE) thread package.
- **Chapter 17.** This is old Chapter 14 on distributed file systems. We have brought the material up-to-date in this rapidly changing area.
- **Chapter 18.** This is old Chapter 13 on distributed coordination. We have brought the material up-to-date and added new sections on the two-phase commit protocol and concurrency control schemes.
- **Chapter 19.** This chapter on UNIX has been updated to reflect the current state of BSD UNIX and its current implementation.
- **Chapter 20.** This is old Chapter 16 on the Mach operating system. It has been updated to describe components of Mach version 3.
- **Appendix.** This is a new Appendix, which was was authored by Professor Thomas Anderson from UC Berkeley. This Appendix provides a brief tutorial introduction to the Nachos system. The Appendix presents the philosophy governing the Nachos environment as well as providing a general introduction to the Nachos operating system and the five project activities which accompany the software. The Appendix concludes with instructions for retrieving Nachos from the Internet via ftp.

Mailing List and Supplements

We now provide an environment where users can communicate among themselves and with us. We have created a mailing list consisting of users of our book with the e-mail address — os-book@cs.utexas.edu. If you wish to be on the list, please send a message to avi@cs.utexas.edu indicating your name, affiliation, and e-mail address.

For information about the teaching supplements, which complement this book, mail may be sent to os4e@aw.com.

Errata

We have attempted to clean up every error in this new edition, but — as happens with operating systems — there will undoubtedly still be some obscure bugs. We would appreciate it if you, the reader, would notify us of any errors or omissions in the book. Also, if you would like to suggest improvements or to contribute exercises, we would be glad to hear from you. Any correspondence should be sent to A. Silberschatz, Department of Computer Sciences, The University of Texas.

Acknowledgments

This book is derived from the previous editions, all of which were coauthored by James Peterson. Other people that have helped with the previous editions include Randy Bentson, Jeff Brumfield, Gael Buckley, Thomas Casavant, Ajoy Kumar Datta, Joe Deck, Robert Fowler, G. Scott Graham, Rebecca Hartman, Wayne Hathaway, Christopher Haynes, Richard Kieburtz, Carol Kroll, Thomas LeBlanc, John Leggett, Michael Molloy, Ed Posnak, John Quarterman, Charles Oualline, John Stankovic, Steven Stepanek, Louis Stevens, and John Werth.

Lyn Dupré copyedited the book; Cliff Wilkes provided technical copyediting; Sara Strandtman edited our text into troff format. Debbie Lafferty, Tom Stone, and Helen Wythe were helpful with book production.

Chapter 17 was derived from a paper by Levy and Silberschatz [1990]. Chapter 19 was derived from a paper by Quarterman et al. [1985]. John Quarterman helped us to convert the material on UNIX 4.2BSD to UNIX 4.3BSD. David Black worked extensively with us to update Chapter 20.

We thank the following people, who reviewed this edition of the book: Joseph Boykin, P. C. Capon, John Carpenter, Thomas Doeppner, Caleb Drake, Hans Flack, Mark Holliday, Jerrold Leichter, Ted Leung, Gary Lippman, Carolyn Miller, Yoichi Muraoka, Jim M. Ng, Boris Putanec, Adam Stauffer, Hal Stern, David Umbaugh, Steve Vinoski, and J. S. Weston.

A.S.
P.B.G.

CONTENTS

PART ONE ■ OVERVIEW

Chapter 1 Introduction

1.1 What Is an Operating System? 3
1.2 Early Systems 6
1.3 Simple Batch Systems 7
1.4 Multiprogrammed Batched Systems 13
1.5 Time-Sharing Systems 15
1.6 Personal-Computer Systems 17
1.7 Parallel Systems 20
1.8 Distributed Systems 22
1.9 Real-Time Systems 23
1.10 Summary 25
Exercises 26
Bibliographic Notes 27

Chapter 2 Computer-System Structures

2.1 Computer-System Operation 29
2.2 I/O Structure 32
2.3 Storage Structure 37
2.4 Storage Hierarchy 42
2.5 Hardware Protection 45
2.6 General-System Architecture 51
2.7 Summary 52
Exercises 53
Bibliographic Notes 55

Chapter 3 Operating-System Structures

3.1 System Components 57
3.2 Operating-System Services 63
3.3 System Calls 65
3.4 System Programs 74
3.5 System Structure 76
3.6 Virtual Machines 82

3.7 System Design and Implementation 86
3.8 System Generation 89
3.9 Summary 90
Exercises 91
Bibliographic Notes 92

PART TWO ■ PROCESS MANAGEMENT

Chapter 4 Processes

4.1 Process Concept 97
4.2 Process Scheduling 100
4.3 Operation on Processes 105
4.4 Cooperating Processes 108
4.5 Threads 111
4.6 Interprocess Communication 116
4.7 Summary 126
Exercises 127
Bibliographic Notes 129

Chapter 5 CPU Scheduling

5.1 Basic Concepts 131
5.2 Scheduling Criteria 135
5.3 Scheduling Algorithms 137
5.4 Multiple-Processor Scheduling 149
5.5 Real-Time Scheduling 150
5.6 Algorithm Evaluation 152
5.7 Summary 158
Exercises 159
Bibliographic Notes 161

Chapter 6 Process Synchronization

6.1 Background 163
6.2 The Critical-Section Problem 165
6.3 Synchronization Hardware 172
6.4 Semaphores 175
6.5 Classical Problems of Synchronization 181
6.6 Critical Regions 186
6.7 Monitors 190
6.8 Synchronization in Solaris 2 198
6.9 Atomic Transactions 199
6.10 Summary 208
Exercises 210
Bibliographic Notes 214

Chapter 7 Deadlocks

7.1 System Model 217
7.2 Deadlock Characterization 219
7.3 Methods for Handling Deadlocks 223
7.4 Deadlock Prevention 224
7.5 Deadlock Avoidance 227
7.6 Deadlock Detection 234
7.7 Recovery from Deadlock 238

7.8 Combined Approach to Deadlock Handling 240
7.9 Summary 241
Bibliographic Notes 245
Exercises 242

PART THREE ■ STORAGE MANAGEMENT

Chapter 8 Memory Management

8.1 Background 249
8.2 Logical versus Physical Address Space 255
8.3 Swapping 256
8.4 Contiguous Allocation 259
8.5 Paging 267
8.6 Segmentation 283
8.7 Segmentation with Paging 290
8.8 Summary 294
Exercises 296
Bibliographic Notes 299

Chapter 9 Virtual Memory

9.1 Background 301
9.2 Demand Paging 303
9.3 Performance of Demand Paging 309
9.4 Page Replacement 312
9.5 Page-Replacement Algorithms 315
9.6 Allocation of Frames 326
9.7 Thrashing 329
9.8 Other Considerations 334
9.9 Demand Segmentation 341
9.10 Summary 342
Exercises 343
Bibliographic Notes 348

Chapter 10 File-System Interface

10.1 File Concept 349
10.2 Access Methods 358
10.3 Directory Structure 361
10.4 Protection 373
10.5 Consistency Semantics 378
10.6 Summary 379
Exercises 380
Bibliographic Notes 381

Chapter 11 File-System Implementation

11.1 File-System Structure 383
11.2 Allocation Methods 387
11.3 Free-Space Management 397
11.4 Directory Implementation 399
11.5 Efficiency and Performance 401
11.6 Recovery 403
11.7 Summary 405
Exercises 406
Bibliographic Notes 408

Chapter 12 Secondary-Storage Structure

12.1 Disk Structure 409
12.2 Disk Scheduling 410
12.3 Disk Management 417
12.4 Swap-Space Management 419
12.5 Disk Reliability 422
12.6 Stable-Storage Implementation 424
12.7 Summary 425
Exercises 426
Bibliographic Notes 427

PART FOUR ■ PROTECTION AND SECURITY

Chapter 13 Protection

13.1 Goals of Protection 431
13.2 Domain of Protection 432
13.3 Access Matrix 438
13.4 Implementation of Access Matrix 443
13.5 Revocation of Access Rights 446
13.6 Capability-Based Systems 448
13.7 Language-Based Protection 451
13.8 Summary 455
Exercises 455
Bibliographic Notes 457

Chapter 14 Security

14.1 The Security Problem 459
14.2 Authentication 461
14.3 Program Threats 464
14.4 System Threats 465
14.5 Threat Monitoring 469
14.6 Encryption 471
14.7 Summary 473
Exercises 473
Bibliographic Notes 474

PART FIVE ■ DISTRIBUTED SYSTEMS

Chapter 15 Network Structures

15.1 Background 479
15.2 Motivation 481
15.3 Topology 482
15.4 Network Types 488
15.5 Communication 491
15.6 Design Strategies 498
15.7 Networking Example 501
15.8 Summary 504
Exercises 504
Bibliographic Notes 505

Chapter 16 Distributed-System Structures

16.1 Network-Operating Systems 507
16.2 Distributed-Operating Systems 509
16.3 Remote Services 512
16.4 Robustness 517
16.5 Design Issues 519
16.6 Summary 521
Exercises 522
Bibliographic Notes 523

Chapter 17 Distributed-File Systems

17.1 Background 525
17.2 Naming and Transparency 527
17.3 Remote File Access 531
17.4 Stateful versus Stateless Service 536
17.5 File Replication 538
17.6 Example Systems 539
17.7 Summary 567
Exercises 568
Bibliographic Notes 569

Chapter 18 Distributed Coordination

18.1 Event Ordering 571
18.2 Mutual Exclusion 574
18.3 Atomicity 577
18.4 Concurrency Control 581
18.5 Deadlock Handling 586
18.6 Election Algorithms 595
18.7 Reaching Agreement 598
18.8 Summary 600
Exercises 601
Bibliographic Notes 602

PART SIX ■ CASE STUDIES

Chapter 19 The UNIX System

19.1 History 607
19.2 Design Principles 613
19.3 Programmer Interface 615
19.4 User Interface 623
19.5 Process Management 627
19.6 Memory Management 632
19.7 File System 636
19.8 I/O System 645
19.9 Interprocess Communication 649
19.10 Summary 655
Exercises 655
Bibliographic Notes 657

Chapter 20 The Mach System

20.1 History 659
20.2 Design Principles 661
20.3 System Components 662
20.4 Process Management 666
20.5 Interprocess Communication 673
20.6 Memory Management 679
20.7 Programmer Interface 685
20.8 Summary 686
Exercises 687
Bibliographic Notes 688

Chapter 21 Historical Perspective

21.1 Atlas 691
21.2 XDS-940 692
21.3 THE 693
21.4 RC 4000 694
21.5 CTSS 695
21.6 MULTICS 696
21.7 OS/360 696
21.8 Other Systems 698

Appendix The Nachos System

A.1 Overview 700
A.2 Nachos Software Structure 702
A.3 Sample Assignments 705
A.4 Information on Obtaining a Copy of Nachos 711
A.5 Conclusions 713
Bibliographic Notes 713

Bibliography 715

Credits 745

Index 747

PART ONE

OVERVIEW

An *operating system* is a program that acts as an intermediary between a 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 to current multiprogrammed and time-shared systems. Understanding the reasons behind the development 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 to ensure such proper behavior. 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 the programming task easier. The specific services provided will, of course, differ from one operating system to another, but there are some common classes of services that we identify and explore.

CHAPTER 1



INTRODUCTION

An *operating system* is a program that acts as an intermediary between a 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. The primary goal of an operating system is thus to make the computer system *convenient* to use. A secondary goal is to use the computer hardware in an *efficient* manner.

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 to current multiprogrammed and time-shared 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. Understanding the reasons behind the development of operating systems gives us an appreciation for what tasks an operating system does and how it does them.

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 *applications programs*, and the *users* (Figure 1.1).

The hardware — the central processing unit (CPU), memory, and input/output (I/O) devices — provides the basic computing resources. The applications programs — such as compilers, database systems, games, and

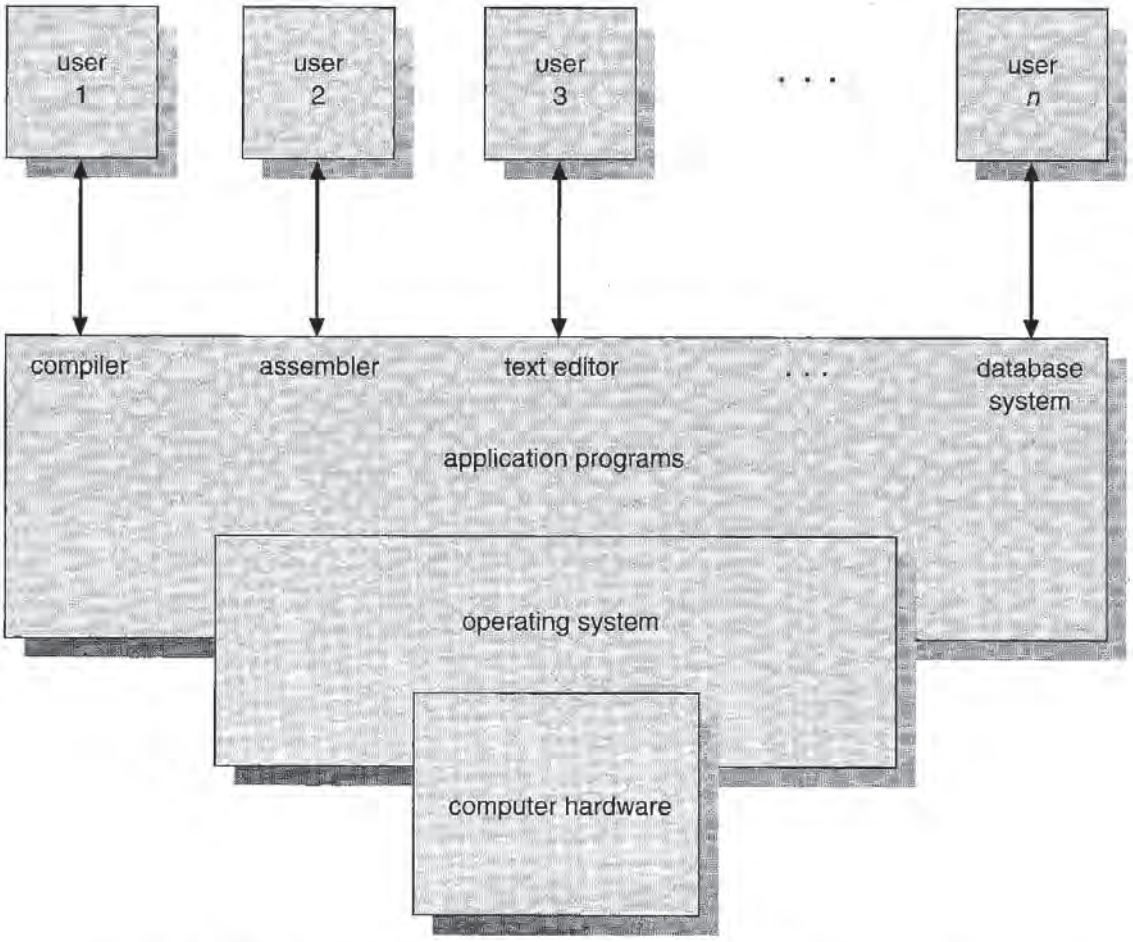


Figure 1.1 Abstract view of the components of a computer system.

business programs — define the ways in which these resources are used to solve the computing problems of the users. There may be many different users (people, machines, other computers) trying to solve different problems. Accordingly, there may be many different applications programs. The operating system controls and coordinates the use of the hardware among the various applications programs for the various users.

An operating system is similar to a *government*. 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. Like a government, the operating system performs no useful function by itself. It simply provides an *environment* within which other programs can do useful work.

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 and allocates them to specific programs and users as necessary for their

tasks. Since there may be many, possibly conflicting, requests for resources, the operating system must decide which requests are allocated resources to operate the computer system efficiently and fairly.

A slightly different view of an operating system focuses on the need to control the various I/O devices and user programs. An operating system is a *control program*. A control program controls 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, there is 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, applications programs are developed. These various 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.

There is also no universally accepted definition of what is part of the operating system and what is not. A simple viewpoint is that everything a vendor ships when you order "the operating system" should be considered. The memory requirements and features included, however, vary greatly across systems. Some take up less than 1 megabyte of space (a megabyte is one million bytes) and lack even a full-screen editor, while others require hundreds of megabytes of space and include spelling checkers and entire "window 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 applications programs. The latter is more common and is the one we generally follow.

It is easier to define operating systems by what they *do*, rather than by what they *are*. The primary goal of an operating system is *convenience for the user*. Operating systems exist because they are supposed to make it easier to compute with one than without one. This view is particularly clear when you look at operating systems for small personal computers.

A secondary goal is *efficient* operation of the computer system. This goal is particularly important for large, shared multiuser systems. These systems are typically 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 considerations were often more important than convenience. Thus, much of operating-system theory concentrates on optimal use of computing resources.

To see what operating systems are and what operating systems do, let us consider how they have developed over the last 30 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 had a great deal of influence on each other. To facilitate the use of the hardware, operating systems were developed. As operating systems were designed and used, it became obvious that changes in the design of the hardware could simplify them. In this short historical review, notice how operating-system problems lead to the introduction of new hardware features.

1.2 ■ Early Systems

Early computers were (physically) enormously large machines run from a console. The programmer, who was also the operator of the computer system, would write a program, and then would operate the program directly from the operator's console. First, the program would be loaded manually into memory, from the front panel switches (one instruction at a time), from paper tape, or from punched cards. Then, the appropriate buttons would be pushed to set the starting address and to start the execution of the program. As the program ran, the programmer/operator could monitor its execution by the display lights on the console. If errors were discovered, the programmer could halt the program, examine the contents of memory and registers, and debug the program directly from the console. Output was printed, or was punched onto paper tape or cards for later printing.

As time went on, additional software and hardware were developed. Card readers, line printers, and magnetic tape became commonplace. Assemblers, loaders, and linkers were designed to ease the programming task. Libraries of common functions were created. Common functions could then be copied into a new program without having to be written again, providing software reusability.

The routines that performed I/O were especially important. Each new I/O device had its own characteristics, requiring careful programming. A special subroutine was written for each I/O device. Such a subroutine is called a *device driver*. A device driver knows how the buffers, flags, registers, control bits, and status bits for a particular device should be used. Each different type of device has its own driver. A simple task, such as reading a character from a paper-tape reader, might involve complex sequences of device-specific operations. Rather than writing the necessary code every time, the device driver was simply used from the library.

Later, compilers for FORTRAN, COBOL, and other languages appeared, making the programming task much easier, but the operation of the computer more complex. To prepare a FORTRAN program for execution, for example, the programmer would first need to load the FORTRAN compiler into the computer. The compiler was normally kept on magnetic tape, so the proper tape would need to be mounted on a tape drive. The program would be read through the card reader and written onto another tape. The

FORTRAN compiler produced assembly-language output, which then needed to be assembled. This procedure required mounting another tape with the assembler. The output of the assembler would need to be linked to supporting library routines. Finally, the binary object form of the program would be ready to execute. It could be loaded into memory and debugged from the console, as before.

Notice that there could be a significant amount of *set-up time* involved in the running of a job. Each job consisted of many separate steps: loading the FORTRAN compiler tape, running the compiler, unloading the compiler tape, loading the assembler tape, running the assembler, unloading the assembler tape, loading the object program, and running the object program. If an error occurred during any step, you might have to start over at the beginning. Each job step might involve the loading and unloading of magnetic tapes, paper tapes, and punch cards.

1.3 ■ Simple Batch Systems

The job set-up time was a real problem. While tapes were being mounted or the programmer was operating the console, the CPU sat idle. Remember that, in the early days, few computers were available, and they were expensive (they cost millions of dollars). In addition, there were the operational costs of power, cooling, programmers, and so on. Thus, computer time was extremely valuable, and owners wanted their computers to be used as much as possible. They needed high *utilization* to get as much as they could from their investments.

1.3.1 Resident Monitor

The solution was two-fold. First, a professional computer operator was hired. The programmer no longer operated the machine. As soon as one job was finished, the operator could start the next. Since the operator had more experience with mounting tapes than a programmer, set-up time was reduced. The user provided whatever cards or tapes were needed, as well as a short description of how the job was to be run. Of course, the operator could not debug an incorrect program at the console, since the operator would not understand the program. Therefore, in the case of program error, a dump of memory and registers was taken, and the programmer had to debug from the dump. Dumping the memory and registers allowed the operator to continue immediately with the next job, but left the programmer with a much more difficult debugging problem.

The second major time savings involved reducing set-up time. Jobs with similar needs were *batched* together and run through the computer as a group. For instance, suppose the operator received one FORTRAN job, one COBOL job, and another FORTRAN job. If she ran them in that order, she

would have to set up for FORTRAN (load the compiler tapes, and so on), then set up for COBOL, and then set up for FORTRAN again. If she ran the two FORTRAN programs as a batch, however, she could set up only once for FORTRAN, saving operator time.

These changes, making the operator distinct from the user and batching similar jobs, improved utilization quite a bit. Programmers would leave their programs with the operator. The operator would sort them 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.

But there were still problems. For example, when a job stopped, the operator would have to notice that fact by observing the console, determine why the program stopped (normal or abnormal termination), take a dump if necessary, and then load the appropriate device with the next job and restart the computer. During this transition from one job to the next, the CPU sat idle.

To overcome this idle time, people developed *automatic job sequencing;* with this technique, the first rudimentary operating systems were created. What was desired was a procedure for automatically transferring control from one job to the next. A small program, called a *resident monitor,* was created for this purpose (Figure 1.2). The resident monitor is always (resident) in memory.

When the computer was turned on, the resident monitor was invoked, and it would transfer control to a program. When the program terminated,

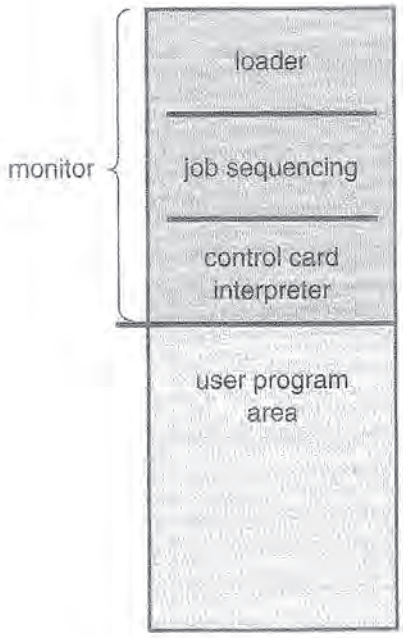


Figure 1.2 Memory layout for a resident monitor.

it would return control to the resident monitor, which would then go on to the next program. Thus, the resident monitor would automatically sequence from one program to another and from one job to another.

But how would the resident monitor know which program to execute? Previously, the operator had been given a short description of what programs were to be run on what data. So that this information could be provided directly to the monitor, *control cards* were introduced. The idea is quite simple. In addition to the program or data for a job, the programmer included special cards (control cards) containing directives to the resident monitor indicating the program to run. For example, a normal user program might require one of three programs to run: the FORTRAN compiler (FTN), the assembler (ASM), or the user's program (RUN). We could use a separate control card for each of these:

$FTN — Execute the FORTRAN compiler.
$ASM — Execute the assembler.
$RUN — Execute the user program.

These cards tell the resident monitor which programs to run.

We can use two additional control cards to define the boundaries of each job:

$JOB — First card of a job.
$END — Last card of a job.

These two cards might be useful for accounting for the machine resources used by the programmer. Parameters can be used to define the job name, account number to be charged, and so on. Other control cards can be defined for other functions, such as asking the operator to load or unload a tape.

One problem with control cards is how to distinguish them from data or program cards. The usual solution is to identify them by a special character or pattern on the card. Several systems used the dollar-sign character ($) in the first column to identify a control card. Others used a different code. IBM's Job Control Language (JCL) used slash marks (//) in the first two columns. Figure 1.3 shows a sample card-deck setup for a simple batch system.

A resident monitor thus has several identifiable parts. One is the *control-card interpreter* that is responsible for reading and carrying out the instructions on the cards at the point of execution. The control-card interpreter at intervals invokes a loader to load systems programs and applications programs into memory. Thus, a loader is a part of the resident monitor. Both the control-card interpreter and the loader need to perform I/O, so the resident monitor has a set of device drivers for the system's I/O devices. Often, the system and applications programs are linked to these

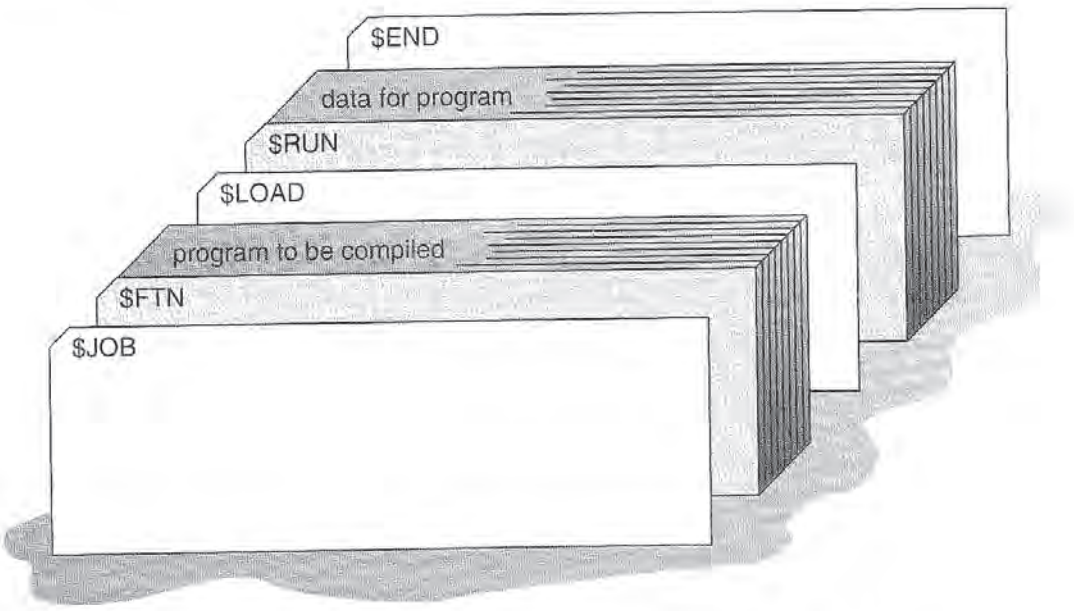


Figure 1.3 Card deck for a simple batch system.

same device drivers, providing continuity in their operation, as well as saving memory space and programming time.

These batch systems work fairly well. The resident monitor provides automatic job sequencing as indicated by the control cards. When a control card indicates that a program is to be run, the monitor loads the program into memory and transfers control to it. When the program completes, it transfers control back to the monitor, which reads the next control card, loads the appropriate program, and so on. This cycle is repeated until all control cards are interpreted for the job. Then, the monitor automatically continues with the next job.

A batch operating system, thus, normally reads a stream of separate jobs (from a card reader, for example), each with its own control cards that predefine what the job does. When the job is complete, its output is usually printed (on a line printer, for example). The definitive feature of a batch system is the *lack* of interaction between the user and the job while that job is executing. The job is prepared and submitted. At some later time (perhaps minutes, hours, or days), the output appears. The delay between job submission and job completion (called *turnaround* time) may result from the amount of computing needed, or from delays before the operating system starts to process the job.

1.3.2 Overlapped CPU and I/O Operations

The switch to batch systems with automatic job sequencing was made to improve performance. The problem, quite simply, is that humans are extremely slow (relative to the computer, of course). Consequently, it is

desirable to replace human operation by operating-system software. Automatic job sequencing eliminates the need for human set-up time and job sequencing.

Even with automatic job sequencing, however, the CPU is often idle. The problem is the speed of the mechanical I/O devices, which are intrinsically slower than electronic devices. Even a slow CPU works in the microsecond range, with millions of instructions executed per second. A fast card reader, on the other hand, might read 1200 cards per minute (17 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 resulted in faster I/O devices. Unfortunately, CPU speeds increased even faster, so that the problem was not only unresolved, but also exacerbated.

1.3.2.1 Off-line processing

One common solution was to replace the very slow card readers (input devices) and line printers (output devices) with magnetic-tape units. The majority of computer systems in the late 1950s and early 1960s were batch systems reading from card readers and writing to line printers or card punches. Rather than have the CPU read directly from cards, however, the cards were first copied onto a magnetic tape via a separate device. When the tape was sufficiently full, it was taken down and carried over to the computer. When a card was needed for input to a program, the equivalent record was read from the tape. Similarly, output was written to the tape and the contents of the tape would be printed later. The card readers and line printers were operated *off-line*, rather than by the main computer (Figure 1.4).

The main advantage of off-line operation was that the main computer was no longer constrained by the speed of the card readers and line printers, but was limited by only the speed of the much faster magnetic

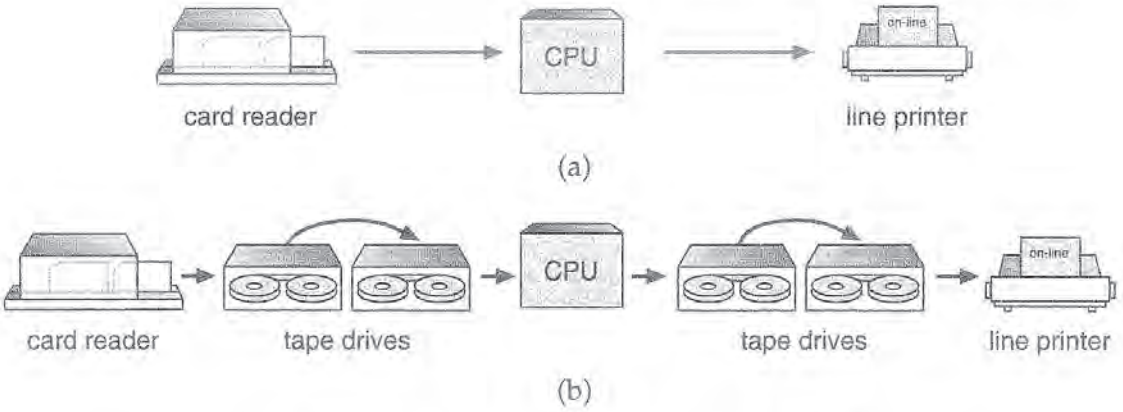


Figure 1.4 Operation of I/O devices. (a) On-line. (b) Off-line.

tape units. This technique of using magnetic tape for all I/O could be applied with any similar equipment (card readers, card punches, plotters, paper tape, printers).

The real gain in off-line operation comes from the possibility of using multiple reader-to-tape and tape-to-printer systems for one CPU. If the CPU can process input twice as fast as the reader can read cards, then two readers working simultaneously can produce enough tape to keep the CPU busy. On the other hand, there is now a longer delay in getting a particular job run. It must first be read onto tape. Then, there is a delay until enough other jobs are read onto the tape to "fill" it. The tape must then be rewound, unloaded, hand-carried to the CPU, and mounted on a free tape drive. This process is not unreasonable for batch systems, of course. Many similar jobs can be batched onto a tape before it is taken to the computer.

1.3.2.2 Spooling

Although off-line preparation of jobs continued for some time, it was quickly replaced in most systems. Disk systems became widely available and greatly improved on off-line operation. The problem with tape systems was that the card reader could not write onto one end of the tape while the CPU read from the other. The entire tape had to be written before it was rewound and read, because tapes are by nature *sequential-access devices*. Disk systems eliminated this problem by being *random-access devices*. Because the head is moved from one area of the disk to another, a disk can switch rapidly from the area on the disk being used by the card reader to store new cards, to the position needed by the CPU to read the "next" card.

In a disk system, cards are read directly from the card reader onto the disk. The location of card images is recorded in a table kept by the operating system. When a job is executed, the operating system satisfies its requests for card-reader input by reading from the disk. Similarly, when the job requests the printer to output a line, that line is copied into a system buffer and is written to the disk. When the job is completed, the output is actually printed.

This form of processing is called *spooling* (Figure 1.5). The name is an acronym for **s**imultaneous **p**eripheral **o**peration **on**-line. Spooling, in essence, uses the disk as a very large buffer, for reading as far ahead as possible on input devices and for storing output files until the output devices are able to accept them.

Spooling is also used for processing data at remote sites. The CPU sends the data via communications paths to a remote printer (or accepts an entire input job from a remote card reader). The remote processing is done at its own speed with no CPU intervention. The CPU just needs to be notified when the processing is completed, so that it can spool the next batch of data.

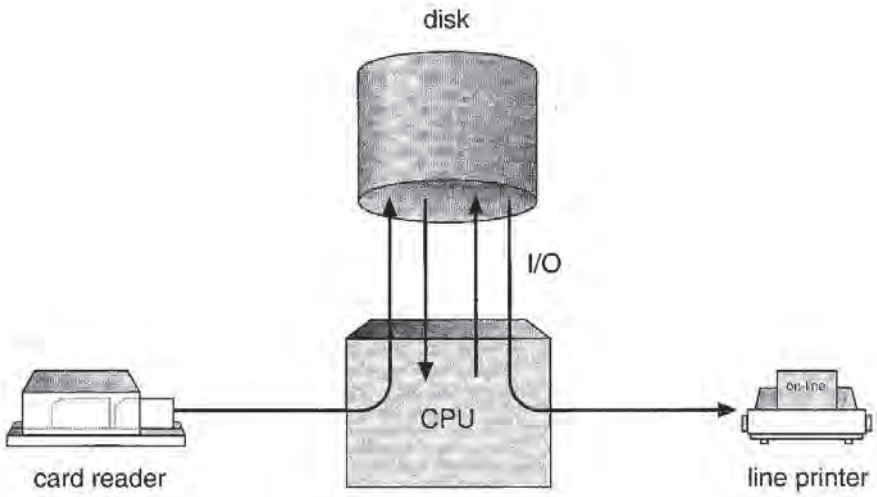


Figure 1.5 Spooling.

Spooling overlaps the I/O of one job with the computation of other jobs. Even in a simple system, the spooler may be reading the input of one job while printing the output of a different job. During this time, still another job (or jobs) may be executed, reading their "cards" from disk and "printing" their output lines onto the disk.

Spooling has a direct beneficial effect on the performance of the system. For the cost of some disk space and a few tables, the computation of one job can overlap with the I/O of other jobs. Thus, spooling can keep both the CPU and the I/O devices working at much higher rates.

1.4 ■ Multiprogrammed Batched Systems

Spooling provides an important data structure: a *job pool*. Spooling will generally result in several jobs that have already been read waiting on disk, ready to run. A pool of jobs on disk allows the operating system to select which job to run next, in order to increase CPU utilization. When jobs come in directly on cards or even on magnetic tape, it is not possible to run jobs in a different order. Jobs must be run sequentially, on a first-come, first-served basis. However, when several jobs are on a direct-access device, such as a disk, *job scheduling* becomes possible. We discuss job and CPU scheduling in greater detail in Chapter 5; a few important aspects are covered here.

The most important aspect of job scheduling is the ability to *multiprogram*. Off-line operation and spooling for overlapped I/O have their limitations. 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 something to execute.

The idea is as follows. The operating system keeps several jobs in memory at a time (Figure 1.6). 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 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 a tape to be mounted, a command to be typed on a keyboard, or an I/O operation to complete. In a nonmultiprogrammed 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 there is always some job to execute, the CPU will never be idle.

This idea is quite common in other life situations. A lawyer does not have only one client at a time. Rather, several clients may be in the process of being served at the same time. While one case is waiting to go to trial or to have papers typed, the lawyer can work on another case. With enough clients, a lawyer need never be idle. (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 mass storage awaiting allocation of main memory. If several jobs are ready to be brought into memory, and there is not enough room for all of them, then the system must choose among them. This decision is *job scheduling*, which

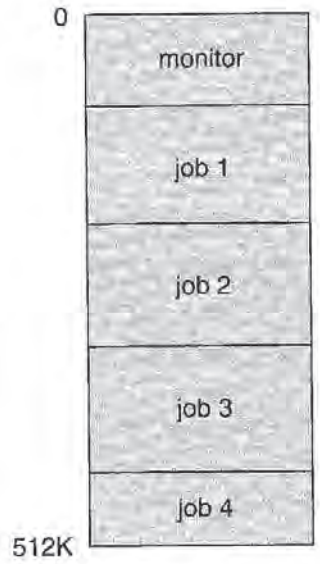


Figure 1.6 Memory layout for a multiprogramming system.

is discussed in Chapter 5. When the operating system selects a job from the job pool, it loads it into memory for execution. Having several programs in memory at the same time requires some form of memory management, which is covered in Chapters 8 and 9. In addition, if several jobs are ready to run at the same time, the system must choose among them. This decision is *CPU scheduling*, which is discussed in Chapter 5. 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.

1.5 ■ Time-Sharing Systems

Multiprogrammed batched systems provide an environment where the various system resources (for example, CPU, memory, peripheral devices) are utilized effectively. There are some difficulties with a batch system from the point of view of the programmer or user, however. Since the user cannot interact with the job when it is executing, the user must set up the control cards to handle all possible outcomes. In a multistep job, subsequent steps may depend on the result of earlier ones. The running of a program, for example, may depend on successful compilation. It can be difficult to define completely what to do in all cases.

Another difficulty is that programs must be debugged statically, from snapshot dumps. A programmer cannot modify a program as it executes to study its behavior. A long turnaround time inhibits experimentation with a program. (Conversely, this situation may instill a certain amount of discipline into the writing and testing of programs.)

Time sharing (or *multitasking*) is a logical extension of multiprogramming. Multiple jobs are executed by the CPU switching between them, but the switches occur so frequently that the users may interact with each program while it is running.

An *interactive*, or *hands-on*, computer system provides on-line communication between the user and the system. The user gives instructions to the operating system or to a program directly, and receives an immediate response. Usually, a keyboard is used to provide input, and a display screen (such as a cathode-ray tube (CRT), or monitor) is used to provide output. When the operating system finishes the execution of one command, it seeks the next "control statement" not from a card reader, but rather from the user's keyboard. The user gives a command, waits for the response, and decides on the next command, based on the result of the previous one. The user can easily experiment, and can see results immediately. Most systems have an interactive text editor for entering programs, and an interactive debugger for assisting in debugging programs.

If users are to be able to access both data and code conveniently, an *on-line file system* must be available. 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, such as text files, or may be rigidly formatted. In general, a file is a sequence of bits, bytes, lines, or records whose meaning is defined by its creator and user. The operating system implements the abstract concept of a file by managing mass-storage devices, such as tapes and disks. Files are normally organized into logical clusters, or *directories*, which makes them easier to use. Since multiple users have access to files, it is desirable to control by whom and in what ways files may be accessed.

Batch systems are quite appropriate for executing large jobs that need little interaction. The user can submit jobs and return later for the results; it is not necessary to wait while the job is processed. Interactive jobs tend to be composed of many short actions, where the results of the next command may be unpredictable. The user submits the command and then waits for the results. Accordingly, the *response* time should be quite short — on the order of seconds at most. An interactive system is used when a short response time is required.

Early computers were interactive systems. That is, the entire system was at the immediate disposal of the programmer/operator. This situation allowed the programmer great flexibility and freedom in program testing and development. But, as we saw, this arrangement resulted in substantial idle time while the CPU waited for some action to be taken by the programmer/operator. Because of the high cost of these early computers, idle CPU time was undesirable. Batch operating systems were developed to avoid this problem. Batch systems improved system utilization for the owners of the computer systems.

Time-sharing systems were developed to provide interactive use of a computer system at a reasonable cost. 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 that is loaded into memory and is 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. 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; five characters per second is fairly fast for people, but is very 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.

A time-shared operating system allows the 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 she has her own computer, whereas actually one computer is being shared among many users.

The idea of time sharing was demonstrated as early as 1960, but since time-shared systems are more difficult and expensive to build (due to the numerous I/O devices needed), they did not become common until the early 1970s. As the popularity of time sharing has grown, researchers have attempted to merge batch and time-shared systems. Many computer systems that were designed as primarily batch systems have been modified to create a time-sharing subsystem. For example, IBM's OS/360, a batch system, was modified to support the Time-Sharing Option (TSO). At the same time, time-sharing systems have often added a batch subsystem. Today, most systems provide both batch processing and time sharing, although their basic design and use tends to be one or the other type.

Time-sharing operating systems are even more complex than are multiprogrammed operating systems. As in multiprogramming, several jobs must be kept simultaneously in memory, which requires some form of memory management and protection (Chapter 8). So that a reasonable response time can be obtained, 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 9). The main visible advantage of this 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 frees programmers from concern over memory storage limitations. Time-sharing systems must also provide an on-line file system (Chapters 10 and 11). The file system resides on a collection of disks; hence, disk management must also be provided (Chapter 12). Also, time-sharing systems provide a mechanism for concurrent execution, which requires sophisticated CPU scheduling schemes (Chapter 5). To ensure orderly execution, the system must provide mechanisms for job synchronization and communication (Chapter 6), and must ensure that jobs do not get stuck in a deadlock, forever waiting for each other (Chapter 7).

Multiprogramming and time sharing are the central themes of modern operating systems, and are the central themes of this book.

1.6 ■ Personal-Computer Systems

As hardware costs have decreased, it has once again become feasible to have a computer system dedicated to a single user. These types of computer systems are usually referred to as *personal computers*, or just *PCs*.

The I/O devices have certainly changed, with panels of switches and card readers replaced with typewriter-like keyboards and mice. Line printers and card punches have succumbed to display screens and small, fast printers.

Personal computers appeared in the 1970s. They are microcomputers that are considerably smaller and less expensive than mainframe systems. Until recently, the CPUs of these types of computer systems have been lacking the features needed to protect an operating system from user programs. Their operating systems therefore have been neither multiuser nor multitasking. However, the goals of these operating systems have changed with time; instead of trying to maximize CPU and peripheral utilization, the systems opt for user convenience and responsiveness. These systems include both the IBM PC family of computers running the MS-DOS operating system, and the Apple Macintosh and its software. MS-DOS has been extended by Microsoft to include a window system, and IBM has upgraded MS-DOS with 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.

Operating systems for these computers have benefited from the development of operating systems for mainframes in several ways. Microcomputers were immediately able to adopt 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 that are made in operating systems for mainframes may not be appropriate for smaller systems. For example, file protection may not seem necessary on a personal machine. MS-DOS, the world's most common operating system, provides no such protection.

Some people have argued that the development of cheap microprocessors and cheap memory will make operating systems (and courses that teach them) obsolete. We do not believe that this prediction is true. Rather, the decrease in hardware costs will allow relatively sophisticated operating-system concepts (such as time sharing and virtual memory) to be implemented on an even greater number of systems. Thus, the decrease in the cost of computer hardware, such as microprocessors, will increase our need to understand the concepts of operating systems.

For example, although file protection may not seem necessary for isolated personal computers, these computers are often tied into other computers over telephone lines or local-area networks. When other computers and other users can access the files on a personal computer, file protection again becomes a necessary feature of an operating system. The lack of such protection enables 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 to disrupt entire companies or even worldwide networks. Protection and security are considered further in Chapters 13 and 14.

In fact, an examination of operating systems for mainframes and microcomputers shows that features that were at one time available only on mainframes have been adopted by microcomputers. The same concepts are appropriate for the various different classes of computers: mainframes, minicomputers, and microcomputers (Figure 1.7).

A good example of this movement occurred with the 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 and complex mainframe computer (the GE 645). Many of the ideas that were developed for MULTICS were subsequently used at Bell Labs (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 are being included in more mainstream operating systems such as Microsoft Windows, IBM OS/2, Macintosh Operating System version 7, and the recently released Microsoft Windows/NT. Thus, the features developed for a large mainframe system have moved to microcomputers, over time.

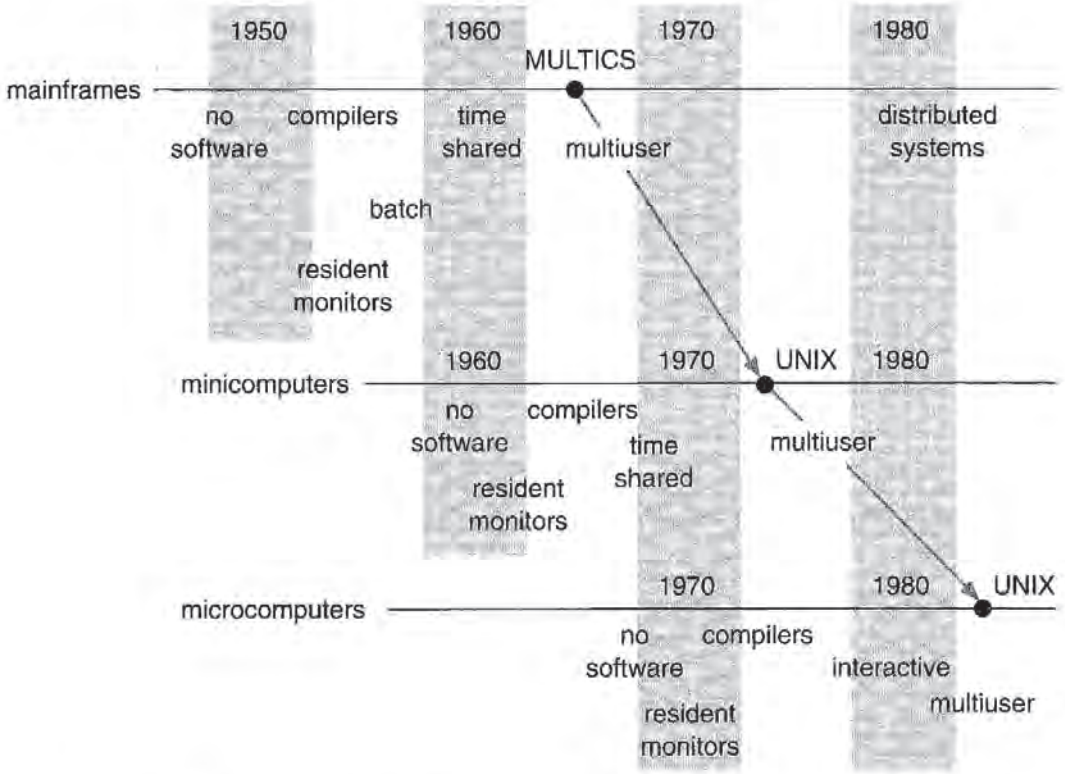


Figure 1.7 Migration of operating-system concepts and features.

At the same time that features of large operating systems were being scaled down to fit personal computers, more powerful, faster, and more sophisticated hardware systems were being developed. The *personal workstation* is a large personal computer, such as the Sun, HP/Apollo, or IBM RS/6000 computer. Many universities and businesses have large numbers of workstations tied together with local-area networks. As the PC systems gain more sophisticated hardware and software, and workstations become less expensive, the line dividing the two breeds is becoming blurry. In the future, the two may merge into one category.

1.7 ■ Parallel Systems

Most systems to date are single-processor systems; that is, they have only one main CPU. However, there is a trend toward multiprocessor systems. Such systems have more than one processor in close communication, sharing the computer bus, the clock, and sometimes memory and peripheral devices. These systems are referred to as *tightly coupled* systems.

There are several reasons for building such systems. One advantage is increased *throughput*. By increasing the number of processors, we would hope to get more work done in a shorter period of time. The speed-up ratio with n processors is not n, however, but rather is less than n. When multiple processors cooperate on a task, a certain amount of overhead is incurred in keeping everything 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.

Multiprocessors can also save money compared to multiple single systems because the processors can share peripherals, cabinets, and power supplies. If several programs are to 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, rather than to have many computers with local disks and many copies of the data.

Another reason for multiprocessor systems is that they increase reliability. If functions can be distributed properly among several processors, then the failure of one processor will not halt the system, but rather will only slow it down. If we have 10 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 nonfailed hardware is called *graceful degradation*. Systems that are designed for graceful degradation are also called *fail-soft*.

Continued operation in the presence of failures requires a mechanism to allow the failure to be detected, diagnosed, and corrected (if possible). 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 machine 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 obviously an expensive one, since there is considerable hardware duplication.

The most common multiple-processor systems now use the *symmetric multiprocessing* model, 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.

An example of the symmetric multiprocessing system is Encore's version of UNIX for the Multimax computer. This computer can be configured to employ dozens of processors, all running a copy of UNIX. The benefit of this model is that many processes can run at once (N processes if there are N CPUs) without causing a deterioration of performance. However, we must carefully control I/O to ensure that data reach the appropriate processor. Also, since the CPUs are separate, one may be sitting idle while another is overloaded, resulting in inefficiencies. To avoid these inefficiencies, the processors can share certain data structures. A multiprocessor system of this form will allow jobs and resources to be shared dynamically among the various processors, and can lower the variance among the systems. However, such a system must be written carefully, as we shall see in Chapter 6.

Asymmetric multiprocessing is more common in extremely large systems, where one of the most time-consuming activities is simply processing I/O. In older batch systems, small processors, located at some distance from the main CPU, were used to run card readers and line printers and to transfer these jobs to and from the main computer. These locations are called *remote job entry (RJE)* sites. In a time-sharing system, a main I/O activity is processing the I/O of characters between the terminals and the computer. If the main CPU must be interrupted for every character for every terminal, it may spend all its time simply processing characters. So that this situation is avoided, most systems have a separate front-end processor that handles all the terminal I/O. For example, a large IBM system might use an IBM Series/1 minicomputer as a front-end. The front-end acts as a buffer between the terminals and the main CPU, allowing the main CPU to handle lines and blocks of characters, instead of individual characters.

Such systems suffer from decreased reliability through increased specialization.

It is important to recognize that the difference between symmetric and asymmetric multiprocessing may be the result of either hardware or software. Special hardware may exist to differentiate the multiple processors, or the software may 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.

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. The IBM PC contains a microprocessor in its keyboard to convert the key strokes 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.

1.8 ■ Distributed Systems

A recent trend in computer systems is to distribute computation among several processors. In contrast to the tightly coupled systems discussed in Section 1.7, the processors 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.

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 different names, such as *sites, nodes, computers,* and so on, depending on the context in which they are mentioned.

There are a variety of reasons for building distributed systems, the major ones being these:

- **Resource sharing.** If a number of different sites (with different capabilities) are connected to one another, then a user at one site may be able to use the resources available at another. For example, a user at site A may be using a laser printer available only at site B. Meanwhile, a user at B may access a file that resides at A. In general, resource sharing in a distributed system provides mechanisms for sharing files at remote sites, processing information in a distributed database,

printing files at remote sites, using remote specialized hardware devices (such as a high-speed array processor), and performing other operations.

- **Computation speedup.** If a particular computation can be partitioned into a number of subcomputations that can run concurrently, then a distributed system may allow us to distribute the computation among the various sites — to run that computation concurrently. In addition, if a particular site is currently overloaded with jobs, some of them may be moved to other, lightly loaded, sites. This movement of jobs is called *load sharing*.
- **Reliability.** If one site fails in a distributed system, the remaining sites can potentially continue operating. If the system is composed of a number of large autonomous installations (that is, general-purpose computers), the failure of one of them should not affect the rest. If, on the other hand, the system is composed of a number of small machines, each of which is responsible for some crucial system function (such as terminal character I/O or the file system), then a single failure may effectively halt the operation of the whole system. In general, if enough redundancy exists in the system (in both hardware and data), the system can continue with its operation, even if some of its sites have failed.
- **Communication.** There are many instances in which programs need to exchange data with one another on one system. Window systems are one example, since they frequently share data or transfer data between displays. When a number of sites are connected to one another by a communication network, the processes at different sites have the opportunity to exchange information. Users may initiate file transfers or communicate with one another via *electronic mail*. A user can send mail to another user at the same site or at a different site.

Distributed systems are discussed in great detail in Chapter 15 through Chapter 18.

1.9 ■ Real-Time Systems

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

some display systems are real-time systems. Also included are some automobile-engine fuel-injection systems, home-appliance controllers, and weapon systems. A real-time operating 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 is considered to function correctly only if it returns the correct result within any 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, where there may be no time constraints at all.

There are two flavors of real-time systems. A *hard real-time* system guarantees that critical tasks complete on time. This goal requires that all delays in the system be bounded, from the retrieval of stored data to the time 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 a nonvolatile storage device that retains its content 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 further from the hardware, and that separation results in uncertainty as to the amount of time an operation will take. For instance, virtual memory (discussed in Chapter 9) 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, kernel delays still need to be bound. A real-time task cannot be kept waiting indefinitely for the kernel to run it. Soft real-time is an achievable goal that is amenable to mixing with other types of systems. Soft real-time systems, however, have more limited utility than do hard real-time systems. Given their lack of deadline support, they cannot be used for industrial control and robotics. There are several areas in which they are useful, however, 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 the two major versions of UNIX.

In Chapter 5, we consider the scheduling facility needed to implement soft real-time functionality in an operating system. In Chapter 9, we describe the design of memory management for real-time computing. Finally, in Chapter 20, we describe the real-time components of the Mach operating system.

1.10 ■ Summary

Operating systems have developed over the past 40 years for two main purposes. First, operating systems attempt to schedule computational activities to ensure good performance of the computing system. Second, they provide 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 monitor 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 provides a means to use multiple reader-to-tape and tape-to-printer systems for one CPU. Spooling allows the CPU to overlap the input of one job with the computation and output of other jobs.

To improve the overall performance of the system, developers introduced the concept of multiprogramming. With multiprogramming, several jobs are 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, which was developed to improve performance, 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.

Personal computer systems are microcomputers that are considerably smaller and less expensive than are mainframe systems. Operating systems for these computers have benefited from the development of operating systems for mainframes in several ways. However, since individuals have sole use of the computer, CPU utilization is no longer a prime concern. Hence, some of the design decisions that are made in operating systems for mainframes may not be appropriate for smaller systems.

Parallel systems have more than one CPU in close communication; the CPUs share the computer bus, and sometimes memory and peripheral devices. Such systems provide increased throughput and enhanced reliability.

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 various communication lines, such as high-speed buses or telephone lines. A distributed system provides the user with access to the various resources located at remote sites.

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.

We have shown the logical progression of operating-system development, driven by inclusion of features in the CPU hardware that are needed for advanced operating-system functionality. This trend can be seen today in the evolution of personal computers, with inexpensive hardware being improved enough 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 that are necessary to run a program on a completely dedicated machine.

1.3 An extreme method of spooling, known as *staging* a tape, is to read the entire contents of a magnetic tape onto disk before using it. Discuss the main advantage of such a scheme.

1.4 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.

1.5 What is the main advantage of multiprogramming?

1.6 What are the main differences between operating systems for mainframe computers and personal computers?

1.7 Define the essential properties of the following types of operating systems:

a. Batch

b. Interactive

c. Time-sharing

d. Real-time

e. Distributed

1.8 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 "waste" resources? Why is such a system not really wasteful?

1.9 Under what circumstances would a user be better off using a time-sharing system rather than a personal computer or single-user workstation?

1.10 Describe the differences between symmetric and asymmetric multiprocessing. What are the advantages and disadvantages of multiprocessor systems?

1.11 Why are distributed systems desirable?

1.12 What is the main difficulty a person must overcome in writing an operating system for a real-time environment?

Bibliographic Notes

Discussions concerning the historical evolution of computer hardware and software systems were presented by Rosen [1969], Denning [1971], and Weizer [1981].

Off-line systems (satellite processing) were used by the IBM FORTRAN Monitor system from the late 1950s to the middle of 1960. Spooling was pioneered on the Atlas computer system at Manchester University [Kilburn et al. 1961]. Spooling was also used on the Univac EXEC II system [Lynch 1972]. Spooling is now a standard feature of most systems, but it was not an integral part of IBM's OS/360 operating system for the 360 family of computers when OS/360 was introduced in the early sixties. Instead, spooling was a special feature added by the Houston computation center of the National Aeronautics and Space Administration (NASA). Hence, it is known as the Houston Automatic Spooling Priority (HASP) system.

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].

Tabak [1990] discussed operating systems for multiprocessor hardware. Forsdick et al. [1978] and Donnelley [1979] discussed operating systems for computer networks. A survey of distributed operating systems was offered by Tanenbaum and Van Renesse [1985]. Real time operating systems are discussed by Stankovic and Ramamrithan [1989]. A special issue on real-time operating systems was edited by Zhao [1989].

The MS-DOS and the IBM-PC family were described by Norton [1986], and Norton and Wilton [1988]. Apple provides an overview of the Apple Macintosh hardware and software [Apple 1987]. Microsoft covers the OS/2 operating system [Microsoft 1989]. More OS/2 information can be found in Letwin [1988] and Deitel and Kogan [1992]. Custer [1993] discussed the structure of Microsoft Windows/NT.

There are numerous up-to-date general textbooks on operating systems. These include Comer [1984], Maekawa et al. [1987], Milenkovic [1987], Bic and Shaw [1988], Finkel [1988], Krakowiak [1988], Pinkert and Wear [1989], Deitel [1990], Stallings [1992], and Tanenbaum [1992]. Useful bibliographies were presented by Metzner [1982] and Brumfield [1986].

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, several disparate parts of this structure are presented 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. Because an operating system is intimately tied to the I/O mechanisms of a computer, I/O is discussed first. The following sections discuss the data-storage structure.

The operating system must also ensure the correct operation of the computer system. So 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.

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.

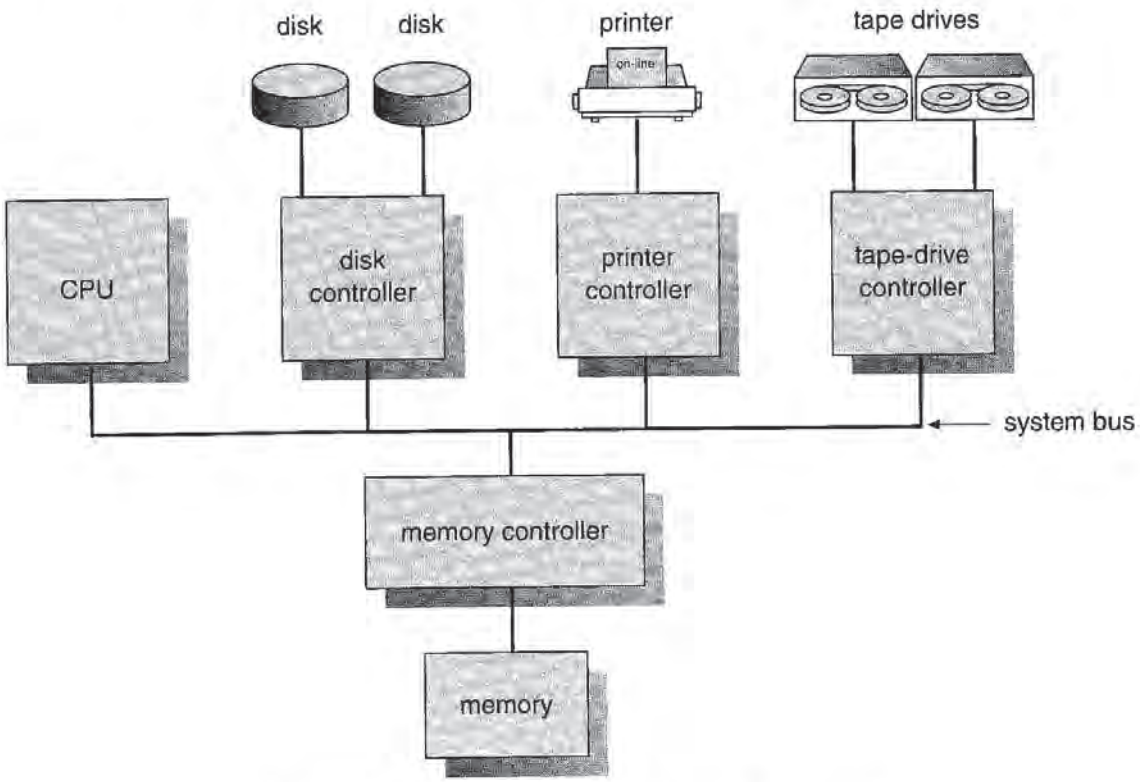


Figure 2.1 A modern computer system.

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. 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 it executing. To accomplish this goal, the bootstrap program must locate the operating system kernel and load it into memory. 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 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*).

There are many different types of events that may trigger an interrupt, for example, the completion of an I/O operation, division by zero, invalid memory access, and a request for some operating system service. For each such interrupt, a 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, and upon 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, and it, in turn, would call the interrupt-specific handler. However, interrupts must be handled very quickly, and given that there are a predefined number of possible interrupts, a table of pointers to interrupt routines may 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 architectures 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 it before returning. After the interrupt is serviced, the saved return address is loaded into the program counter, and the interrupted computation will resume as though the interrupt had not occurred.

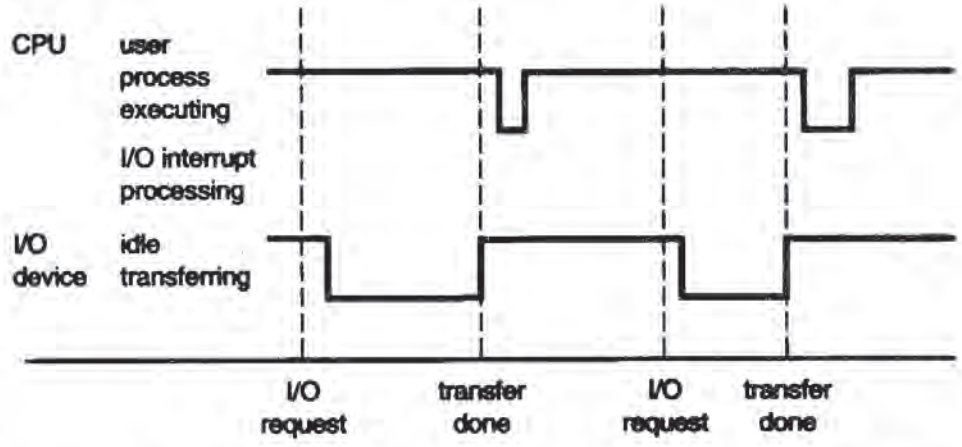


Figure 2.2 Interrupt time line for a single process doing output.

Usually, interrupts are *disabled* while an interrupt is being processed, delaying any incoming interrupts until the operating system is done with the current one, after which interrupts are *enabled*. If they were not thus disabled, the processing of the second interrupt while the first was being serviced would overwrite the first's data, and the first would be a *lost interrupt*. Sophisticated interrupt architectures allow for one interrupt to be processed during another. They often use a priority scheme in which request types are assigned priorities according to their relative importance, and interrupt processing information is stored separately for each priority. A higher-priority interrupt will be taken even if a lower-priority interrupt is active, but interrupts at the same or lower levels are *masked*, or selectively disabled, to prevent lost interrupts or unnecessary ones.

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. When an interrupt (or trap) occurs, the hardware transfers control to the operating system. First, the operating system preserves the state of the CPU by storing registers and the program counter. Then, it determines which type of interrupt has occurred. This determination may require *polling*, the querying of all I/O devices to detect which requested service, or it may be a natural result of a vectored interrupt system. For each type of interrupt, separate segments of code in the operating system determine what action should be taken.

2.2 ■ I/O Structure

As was discussed in Section 2.1, a general-purpose computer system consists of a CPU and a number of 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 device attached to it. For instance the *SCSI* (Small Computer Systems Interface) controller, found on the Macintosh and many small- to medium-sized computers, can have as many as seven devices attached to it. A device controller maintains some local buffer storage and a set of special-purpose registers. The controller is responsible for moving data between the peripheral device(s) 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 (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 causing 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 is known as *synchronous* I/O. The other possibility (*asynchronous* I/O) is to return control to the user program without waiting for the I/O to complete. The I/O then can continue while other system operations occur (see 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 very tight loop simply continues until an interrupt occurs, transferring

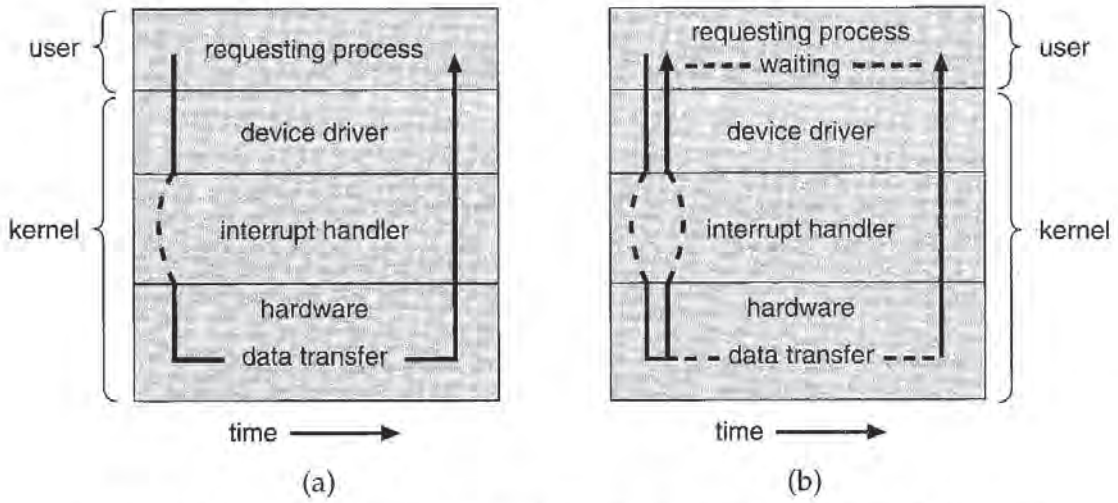


Figure 2.3 Two I/O methods. (a) synchronous. (b) asynchronous.

control to another part of the operating system. Such a loop might also need to poll any I/O devices which do not support the interrupt structure; instead they simply set a flag in one of their registers and expect the operating system to notice it. The **wait** instruction is probably better than a loop, since a wait loop generates a series of instruction fetches, which may cause significant contention for memory access. The contention is caused by the I/O device transferring information and the CPU transferring instructions.

One major advantage of always waiting for I/O completion is that 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 simultaneous I/O processing.

An alternative is to start the I/O and immediately to return control to the user program. A *system call* (a request to the operating system) is then needed to allow the user to wait for I/O completion, if desired. Hence, we still require the wait code that we needed 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, its address, and its 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 device. Since it is possible for other processes to issue requests to the same device, we may have a list or chain

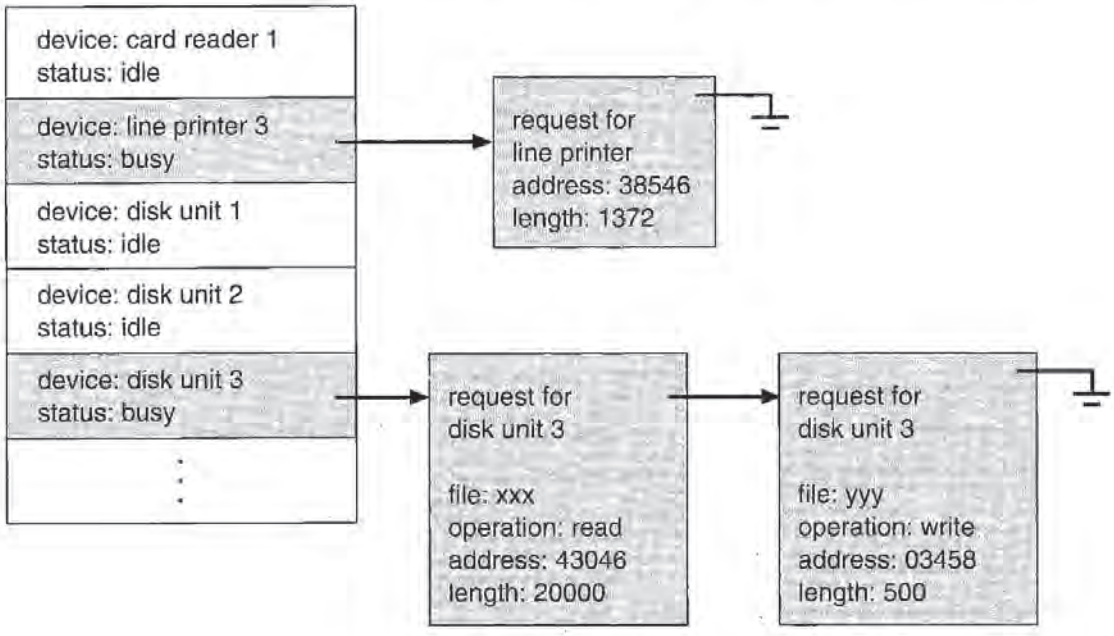


Figure 2.4 Device-status table.

of waiting requests. Thus, in addition to the I/O device table, an operating system may have a request list for each 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 is an additional request waiting for this device, the operating system starts processing that 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 may 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 others). In a time-sharing system, the operating system could switch to another ready-to-run process.

The schemes used by some input devices may vary from this one. Many interactive systems allow users to type ahead, or to enter data before the data are requested, on their terminal. 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 terminal.

The main advantage of asynchronous I/O is the increased system efficiency. While I/O is taking place, the system CPU can be used for processing, or even scheduling other I/O. Because I/O can be quite slow compared to processor speed, the system makes much better use of its facilities. In the next section, we shall see another mechanism for improving system performance.

2.2.2 DMA Structure

Consider 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 will interrupt the CPU. When the interrupt request from the terminal arrives, the CPU will be about to execute some instruction. (If the CPU is in the middle of executing an instruction, the interrupt is normally held pending until the instruction execution is complete.) 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 it will need to use. It checks for any error conditions that might have resulted from the last 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 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 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; the CPU would need 2 microseconds to respond to each interrupt, with interrupts arriving every 4 microseconds (for example). This would 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 to or from its own buffer storage to memory directly, 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 queue of buffers for the transfer. (A buffer is typically 128 to 4096 bytes, depending on the device type.) The DMA controller then has its registers set to the appropriate source and destination addresses, and transfer length. This register setting is usually done by a *device driver*, which knows exactly how this information is to be provided to the controller. The DMA controller is then instructed (via control bits in a control register) to start the I/O operation. Meanwhile, the CPU has been free to perform other tasks since it gave the transfer information to the controller. The DMA controller interrupts the CPU when the transfer has been completed.

As an example of the utility of DMA, consider a typical IBM PC. The PC supports interrupt-based I/O, as well as DMA channels. The backup program, which is included with the MS-DOS PC operating system, uses only interrupt-based I/O to copy data between a hard disk and a floppy disk. Several companies have written similar programs that take advantage of DMA data transfer, and the result is a several-fold increase in backup speed.

2.3 ■ Storage Structure

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. 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, while 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 *von Neumann* architecture system, will first fetch an instruction from memory and store it 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 (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.

Ideally, we would 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 lost.

Thus, most computer systems provide *secondary storage* as an extension of main memory. The main requirement of secondary storage is thus to be able to hold extremely large numbers of data permanently.

The most common secondary-storage device is a magnetic disk, which provides storage of both programs and data. Most programs (compilers, assemblers, sort routines, editors, formatters, and so on) are stored on a disk until loaded into memory. They then use the disk as both the source and destination of the information for their processing. Hence, the proper management of disk storage is of central importance to a computer system, as will be discussed in Chapter 12.

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, optical disks, magnetic tapes, and so on. Each storage system provides the basic functions of storing a datum, and 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 to 2.3.4, we describe the important storage systems.

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. (Consider that there are machine instructions which 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, some systems, including the IBM PC and Apple Macintosh, provide *memory-mapped I/O*. In this case, ranges of memory addresses are set aside, and are physically 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 with 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 frequently used devices, such as a serial port. So that a system can communicate with another computer over a modem and telephone line, characters are read and written to a one-byte location in memory. Note that polling is again needed in this circumstance, with the computer looping, constantly checking to see whether a character is available for reading. Interrupts for signaling the availability of new input could be used instead, depending on the hardware configuration.

Given that registers are built into the CPU, they are accessible within one cycle of the CPU clock. The CPU can decode an instruction and perform the given operation on a register's contents all in the same clock tick. The same cannot be said for main memory, which may be located on a system bus and take several cycles to access. In this case, the processor normally needs to *stall* while waiting for the access to complete. 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 is called a *cache*, as described in Section 2.4.1.

2.3.2 Magnetic Disks

Magnetic disks provide the bulk of secondary storage for modern computer systems. Physically, disks are relatively simple (Figure 2.5). Each disk *platter* has a flat circular shape, like a phonograph record. Its two surfaces are covered with a magnetic material, similar to magnetic tape. Information is recorded on the surfaces.

When the disk is in use, a drive motor spins it at high speed (usually 60 revolutions per second). There is a read–write head positioned just above the surface of the platter. The disk surface is logically divided into *tracks*, which are subdivided into *sectors*. We store information by

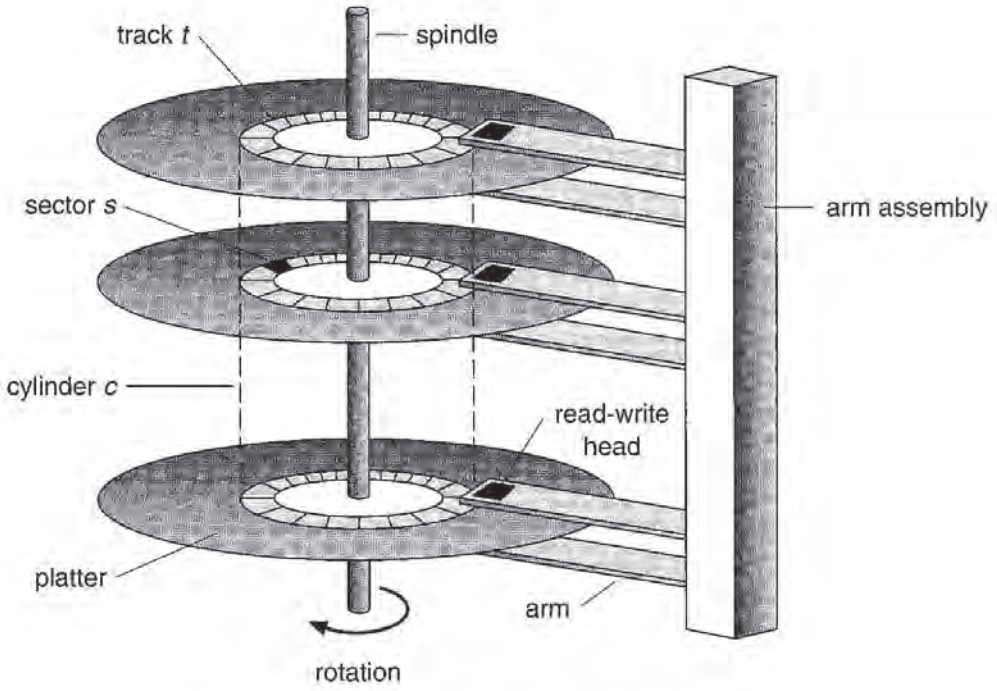


Figure 2.5 Moving-head disk mechanism.

recording it magnetically on the sector under the read–write head. There may be hundreds of concentric tracks on a disk surface, containing thousands of sectors. The platter itself may be between 1.8 inches and 14 inches wide. The larger sizes are common on large systems because of their higher storage capacities and transfer rates. The smaller sizes are found on PCs, since they have lower cost.

A *fixed-head disk* has a separate head for each track. This arrangement allows the computer to switch from track to track quickly, but it requires a large number of heads, making the device extremely expensive. Much more common, is only one head, which moves across the disk to access different tracks. This *moving-head disk*, or simply *hard disk*, requires hardware to move the head, but only a single head is needed, resulting in a much less expensive system. The disk platters, mounted on a spindle and surrounded by heads driven by a motor, are known as *head–disk assemblies* and come in a complete package.

Disks were originally designed for file storage, so the primary design criteria were cost, size, and speed. To provide additional storage capacity, developers took several approaches. They made the primary gain by improving the recording density, allowing more bits to be put on a surface. The density is reflected by the number of tracks per inch, sectors per track, and bits per sector. In addition, with separate heads on each side of the platter, disk capacity can be doubled at minimal cost. We can extend this approach by stacking several disks, each with two recording surfaces, on one spindle. Since all the disks rotate together, only one drive motor is needed, although each surface still needs its own read–write head. The most common disks, used in systems from portable PCs through mainframes, have this configuration. These disks vary in data transfer rate from 1 to 5 megabytes per second. The average access time, including the time for a head to be positioned over the requested data, is from 10 to 40 milliseconds. The capacities range from 10 to 7500 megabytes.

Finally, the disk can be *removable*, allowing different disks to be mounted as needed. Removable disk packs may consist of one or several platters on one spindle. Generally, they are held in hard plastic cases to prevent damage while they are not in the disk drive.

Disks are rigid metal or glass platters covered with magnetic recording material. Each platter is divided into small sections, and each such section may be changed by the disk head to be in a charged or not charged state. Each section represents a bit and is 0 or 1 depending on its charge state. The smaller the changeable sections are, the more bits can be put on a platter and thus the higher the density. The read–write heads are kept as close as possible to the disk surface to increase this density. Often, the head floats or flies only microns from the disk surface, supported by a cushion of air. Because the head floats so close to the surface, platters must be machined carefully to be flat.

Head crashes can be a problem. If the head contacts the disk surface (due to a power failure, for example), the head will scrape the recording medium off the disk, destroying the data that had been there. Usually, the head touching the surface causes the removed medium to become airborne and to come between the other heads and their platters, causing more crashes. Under normal circumstances, a head crash results in the entire disk failing and needing to be replaced.

Floppy disks take a different approach. The disks are coated with a hard surface, so the read–write head can sit directly on the disk surface without destroying the data. Thus, the disk itself is much less expensive to produce and use. The floppy disk must rotate much more slowly than a hard disk, due to the resulting friction. Also, the coating (and the read–write head) will wear with use, and need to be replaced eventually. Because they are more rugged, floppy disks are removable. The disks are not permanently mounted in a head–disk assembly. Instead, the disk is slipped manually into a slot which contains a spindle to rotate it, and a head and motor to access it. This arrangement keeps the cost of floppy disks low because one drive can be used to access hundreds or thousands of disks.

Floppy disks usually have a much lower capacity than do hard disks because they have much lower storage densities, have only one platter, and spin slower. They hold from 100 kilobytes (a kilobyte is 1024 bytes) to a few megabytes per disk. They come in many variations (single-sided, double-sided, single-density, and double-density) and sizes (5 1/4 inch, 3 1/2 inch, and so on). Generally, they are formatted and used in the same way as are hard disks, except that all floppy disks are removable and may therefore be used conveniently to transfer data between computers.

A disk drive has a *disk controller* that determines the logical interaction between the device and the computer. The controller takes instructions from the CPU and orders the disk drive to carry out the instruction. Some disk controllers have a built-in cache, which holds data recently read from or written to the disk. If data are currently in the cache, the need for a disk transfer is obviated.

2.3.3 Other Disk Types

There are many variations on magnetic disks among which we do not distinguish, from an operating-system point of view. Generally, they are treated as normal hard disks except at the device-driver level. There is the *drum*, which has a head over each track. Previously, these fast devices were used for *backing store* (see Chapters 8 and 9), but they are now used rarely due to the improved speed and price–performance ratio of standard hard disks. An up-and-coming device is the *optical disk*, which uses lasers to melt and fill holes on a plastic medium. These drives have been slower but less costly and more rugged than magnetic hard disks in the past, but

their performance is increasing. Because they are more rugged, they, like floppy disks, have the added advantage of removability.

2.3.4 Magnetic Tapes

Magnetic tape was used as an early secondary-storage media. Although it is relatively permanent, and can hold large numbers of data, magnetic tape is quite slow in comparison to the access time of main memory. Even more important, magnetic tape is limited to sequential access. Thus, it is unsuitable for providing the random access needed for most secondary-storage requirements. 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 rather than milliseconds; once positioned, however, tape drives can write data at densities and speeds approaching those of disk drives. Capacities vary depending on the length and width of the tape, and on the density at which the head can read and write. A tape drive is usually named by its width. Thus, there are 8-millimeter, 1/4-inch, and 1/2-inch (also known as 9-track) tape drives. The 8-millimeter tape drives have the highest density, due to the technology they use; they currently store 5 gigabytes of data (a gigabyte is one billion bytes) on a 350-foot tape.

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 their speed and their cost. The higher levels are expensive, but are fast. As we move down the hierarchy, the cost per bit decreases, whereas the access time 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.

In addition to the speed and cost of the various storage systems, there is also the issue of storage volatility. 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 system above disks is volatile, whereas the storage system below main memory is nonvolatile. The design of a complete memory system attempts to balance all these factors: It uses only as much expensive memory as

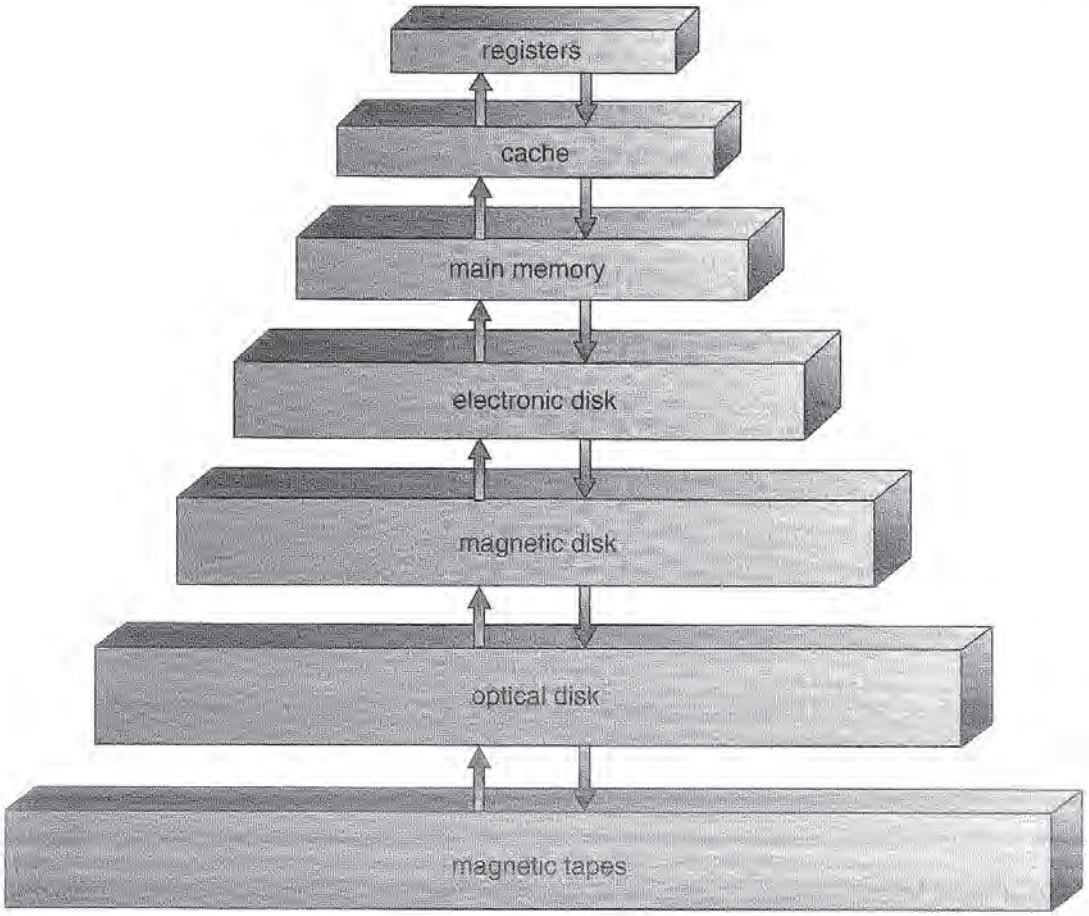


Figure 2.6 Storage-device hierarchy.

necessary, while providing as much inexpensive, nonvolatile, memory as possible. Caches can be installed to ameliorate performance differences where there is a large access-time or transfer-rate disparity between two components.

2.4.1 Caching

Caching is an important principle of computer systems, in both hardware and software. 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 there is a high probability that it will be needed again.

Extending this view, internal programmable registers, such as index registers and accumulators, are 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 was fetched from main memory. We are not concerned with these hardware-only caches in this text, since they are outside of the control of the operating system.

Since 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, resulting in extremely high performance. Various replacement algorithms for software-controlled caches are discussed in Chapter 9.

Main memory can be viewed as a fast *cache* for secondary memory, since data on 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 itself, which must reside on nonvolatile storage, may have several levels of storage. At the highest level, we have the electronic (or RAM) disk storage, which is backed up by the larger, but slower, magnetic-disk storage. The magnetic-disk storage, in turn, is backed up by the larger, but slower, tape storage. Optical disks are also efficient high-capacity but low-cost storage media. When compared to magnetic tape, they have the drawback of higher cost, but offer much greater speed and convenience. Transfers between these two storage levels are generally requested explicitly, but some systems now automatically archive a file that has not been used for a long time (for example, 1 month), and then automatically fetch back the file to disk when it is next referenced.

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 datum may appear in different storage systems. For example, consider an integer A located in file B that is to be incremented by 1. Suppose that 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 a possible copying of A to the cache, and by copying A to an internal register. Thus, the copy of A appears in several places. 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 back to the magnetic disk.

In an environment where there is only one single process executing 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 internal registers, the CPU 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 problem is called *cache coherency*, and is usually a hardware issue (handled below the operating system level).

In a distributed environment, the situation becomes even more complex. In such an environment, several copies (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, then all other replicas are brought up to date as soon as possible. There is a variety of ways for achieving this guarantee, as will be discussed in Chapter 17.

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. Starting with the resident monitor, the operating system began performing many of the functions, especially I/O, for which the programmer had been responsible previously.

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 created 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 earliest resident monitor, providing nothing more than automatic job sequencing (Section 1.3). Suppose 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 could 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 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 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 is dumped. In a batch system, the memory dump may be printed, allowing the user to try to find the cause of the error by examining the printed dump. In an interactive system, the memory dump may be written to a file. The user may then examine it on-line, and perhaps 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 is to provide hardware support to allow us to differentiate among various modes of executions. 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 an execution that is done on behalf of the operating system, and one that is done 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 be 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 to the operating system.

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 80486, do provide dual-mode operation. As a result, more recent operating systems, such as Microsoft Windows/NT, 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. Various mechanisms are used to ensure that such disruptions cannot take place in the system.

To prevent a user 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 the 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. Suppose 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.

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. Otherwise, a user program might overwrite instructions in the interrupt service routine with jumps to the user program, thus gaining control from the interrupt service routine that was executing in monitor mode. Even if the user did not 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, however, 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 shall see in Chapter 8. Here, we outline one such possible implementation.

What is needed to separate each program's memory space is an 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 2.7. The base register holds the smallest legal physical memory address, and the limit register contains the size of the range. For example, if the base register holds 300040 and limit register is 120900, then the

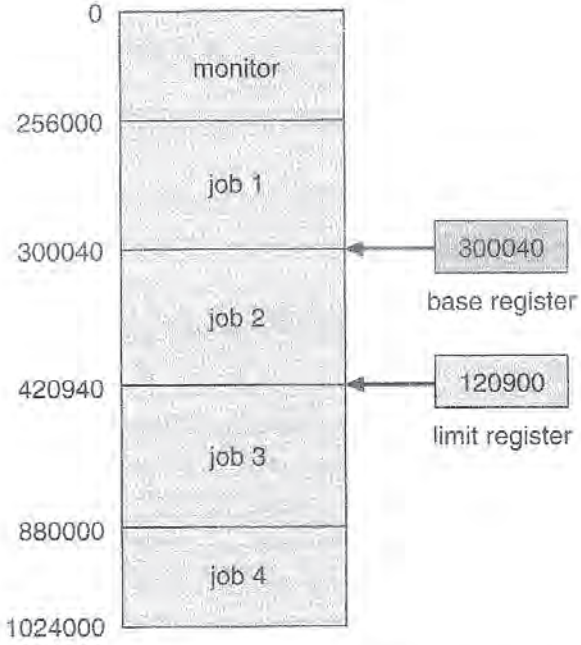


Figure 2.7 A base and a limit register define a logical address space.

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 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 2.8). 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 only in 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 them out in case of errors, to access and modify parameters of system calls, and so on.

2.5.4 CPU Protection

The third piece of the protection puzzle is ensuring that the operating system maintains control. We must prevent a user program from getting stuck in an infinite loop, and never returning control to the operating system. To accomplish this goal, we can use a *timer*. A timer can be set to

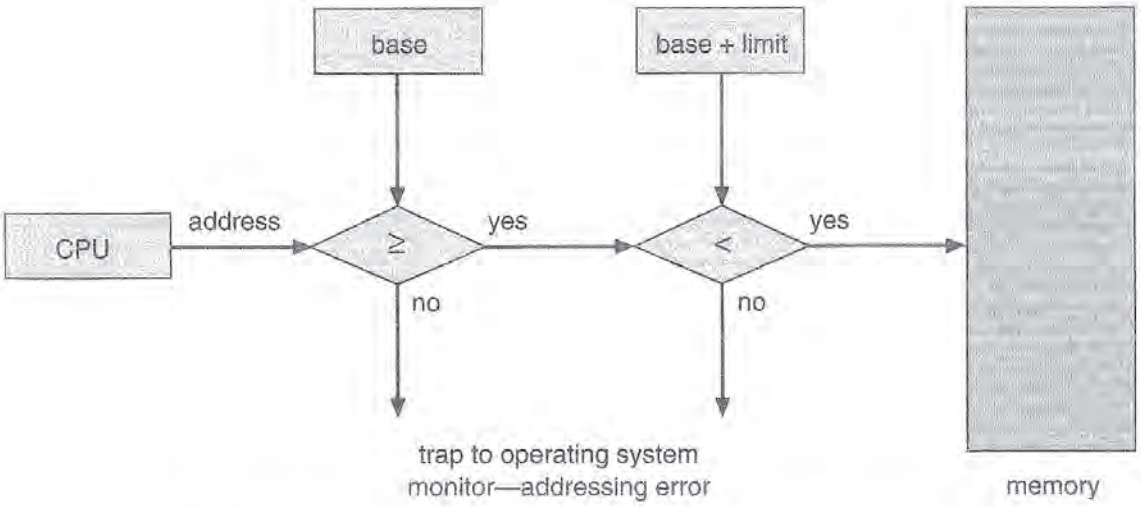


Figure 2.8 Hardware address protection with base and limit registers.

interrupt the computer after a specified period. The period may be fixed (for example, 1/60 second) or variable (for example, from 1 millisecond to 1 second, in increments of 1 millisecond). 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 zero, an interrupt occurs. For instance, a 10-bit counter with a 1-millisecond clock would allow interrupts at intervals from 1 millisecond to 1024 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 give the program more time. Instructions that modify the operation of the timer are clearly privileged.

Thus, the timer can be used 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 its 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* 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 resets 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*, and 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 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 ■ General-System Architecture

The desire to improve the utilization of the computer system led to the development of multiprogramming and time sharing, where the resources of the computer system are shared among many different programs and processes. Sharing led directly to modifications of the basic computer architecture, to allow the operating system to maintain control over the computer system, and especially over I/O. Control must be maintained if we are to provide continuous, consistent, and correct operation.

To maintain control, developers introduced a dual mode of execution (user mode and monitor mode). This scheme supports the concept of privileged instructions, which can be executed only in monitor mode. I/O instructions and instructions to modify the memory-management registers or the timer are privileged instructions.

As you can imagine, several other instructions are also classified as privileged. For instance, the **halt** instruction is privileged; a user program should never be able to halt the computer. The instructions to turn the interrupt system on and off are also privileged, since proper operation of the timer and I/O depends on the ability to respond to interrupts correctly. The instruction to change from user mode to monitor mode is privileged, and on many machines any change to the mode bit is privileged.

Because I/O instructions are privileged, they can be executed by only the operating system. Then how does the user program perform I/O? By making I/O instructions privileged, we have prevented user programs from doing any I/O, either valid or invalid. The solution to this problem is that, because only the monitor can do I/O, the user must *ask* the monitor to do I/O on the user's behalf.

Such a request is known as a *system call* (also called a *monitor call* or *operating system function call*). 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 (like the Mips R2000 family) have a specific **syscall** instruction.

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.

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 2.9). 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.

2.7 ■ Summary

Multiprogramming and time-sharing systems require the overlap of CPU and I/O operations on a single machine. Such an overlap requires that data transfer between the CPU and the I/O devices be handled by one or both of the following methods: (1) interrupt-initialized data transfer, (2) 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

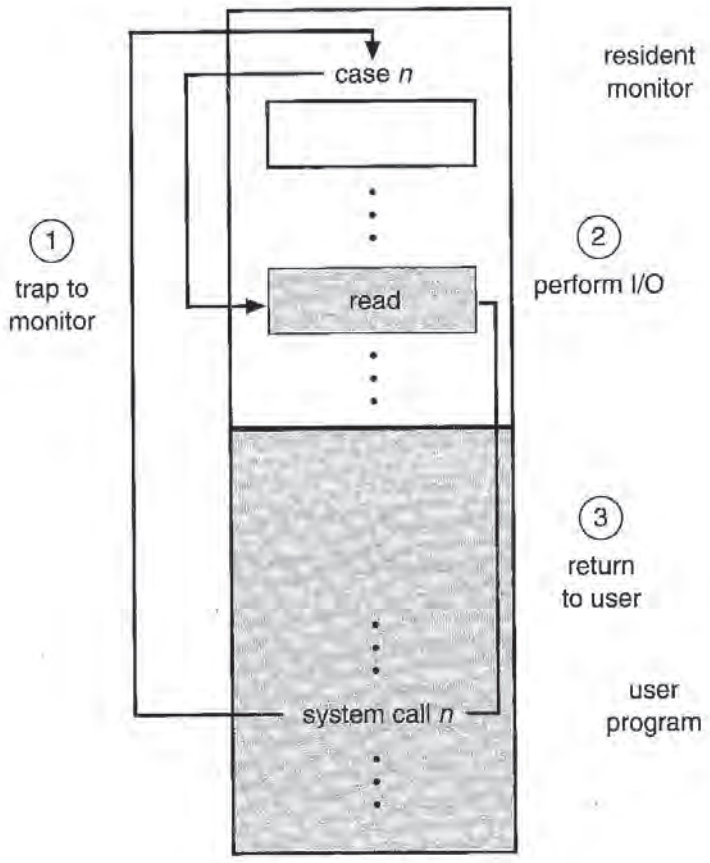


Figure 2.9 Use of a system call to perform I/O.

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 extremely large numbers 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 their cost. The higher levels are expensive, but are fast. As we move down the hierarchy, the cost per bit decreases, whereas the access time 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, developers modified the hardware to create 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. Once these changes (dual mode, privileged instructions, memory protection, timer interrupt) have been made to the basic computer architecture, it is possible to ensure the correct operation of the system. Chapter 3 continues this discussion with details of the facilities that operating systems provide.

■ Exercises

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

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

2.2 Show how a desire for control cards leads naturally to the creation of separate user and monitor modes of operation.

2.3 How does the distinction between monitor mode and user mode function as a rudimentary form of protection (security) system?

2.4 What are the differences between a trap and an interrupt? What is the use of each function?

2.5 For what types of operations is DMA useful? Why?

2.6 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.

2.7 Many computer systems do not provide dual-mode operation in hardware. Consider whether it is possible to construct a secure operating system for these computers. Give arguments both that it is and that it is not possible.

2.8 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.

2.9 Protecting the operating system is crucial to ensuring that the computer system operates correctly. Provision of this protection is the reason behind dual-mode operation, memory protection, and the timer. To allow maximum flexibility, however, we would also like to place minimal constraints on the user. The following is a list of operations 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.

2.10 When are caches useful? What problems do they solve? What problems do they cause? If a cache can be made as large as the device it is caching for (for instance, a cache as large as a disk) why not do so and eliminate the device?

2.11 Writing an operating system that can operate without interference from malicious or undebugged user programs requires some hardware assistance. Name three hardware aids for writing an operating system, and describe how they may be used together to protect the operating system.

Bibliographic Notes

A detailed description of I/O architectures such as channels and DMA on large systems appears in Baer [1980]. Hennessy and Patterson [1990] provide modern 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.

General discussions concerning multiprocessing are given by Jones and Schwarz [1980]. Multiprocessor hardware is discussed by Satyanarayanan [1980]. Performance of multiprocessor systems is presented by Maples [1985], Sanguinetti [1986], Agrawal et al. [1986], and Bhuyan et al. [1989]. A survey of parallel computer architectures is presented by Duncan [1990].

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

Cache memories, including associative memory, are described and analyzed by Smith [1982]. This paper also includes an extensive bibliography on the subject. Hennessy and Patterson [1990] discuss the hardware aspects of TLBs, caches, and MMUs.

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

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. It is important that the goals of the system be well defined before the design begins. The type of system desired is the foundation for choices among various algorithms and strategies that will be necessary.

There are several vantage points from which to view an operating system. One is by examining the services it provides. Another is by looking at the interface 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, to show them from 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 of these pieces should be a well-delineated portion of the system, with carefully defined inputs, outputs, and function. Obviously, not all systems have the same structure. However, many modern systems share the goal of supporting the types of 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. Typically, a batch job is a process. A time-shared user program is a process. A system task, such as spooling output to a printer, also is a process. For now, you can consider a process to be a job or a time-shared program, but the concept is actually more general. As we shall see in Chapter 4, it is possible to 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, some initialization data (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 the desired information and display it on the terminal. 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 progress in a sequential fashion. The CPU executes one instruction of the process after another, until the process completes. Further, at any point in 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:

- The creation and deletion of both user and system processes
- The suspension and resumption of processes
- The provision of mechanisms for process synchronization
- The provision of mechanisms for process communication
- The provision of mechanisms for deadlock handling

Process-management techniques will be discussed in great detail in Chapters 4 to 7.

3.1.2 Main-Memory Management

As discussed in Chapter 1, memory is central to the operation of a modern computer system. Memory is a large array of words or bytes, each with its own address. It 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 both reads and writes data from main memory during the data-fetch cycle. I/O implemented via DMA also reads and writes data in main memory. Main memory is generally the only storage device that the CPU is able to address 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 CPU and the speed of the computer's response to its users, we must keep several programs 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 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:

- Keep track of which parts of memory are currently being used and by whom
- Decide which processes are to be loaded into memory when memory space becomes available
- Allocate and deallocate memory space as needed

Memory-management techniques will be discussed in great detail in Chapters 8 and 9.

3.1.3 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 (*primary*

storage) during execution. Because main memory is too small to accommodate all data and programs, and its data 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 principle 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 disk subsystem and the algorithms which manipulate it. Techniques for secondary-storage management will be discussed in detail in Chapter 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 system*. The I/O system consists of

- A buffer-caching system
- 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 have already discussed in Chapter 2 how interrupt handlers and device drivers are used in the construction of efficient I/O systems. In Chapter 12, we shall discuss at great length how a particular device (the disk) is managed effectively.

3.1.5 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, with its own unique characteristics. These properties include speed, capacity, data transfer rate, and access method (sequential or random access method).

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, such as text files, or may be formatted rigidly. 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 tapes and disks, and the devices which control them. Also, files are normally organized into directories to ease their use. Finally, when multiple users have access to files, it may be desirable to control by whom and in what ways files may be accessed.

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

- The creation and deletion of files
- The creation and deletion of directories
- The support of primitives for manipulating files and directories
- The mapping of files onto secondary storage
- The backup of files on stable (nonvolatile) storage media

File-management techniques will be discussed in Chapters 10 and 11.

3.1.6 Protection System

If a system has multiple users and allows multiple concurrent processes, the various processes must be protected from one another's activities. For that purpose, mechanisms are provided to 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. Finally, users are not allowed to do their own I/O, so that the integrity of the various peripheral devices is protected.

Protection refers to 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, together with a means of 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 a 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 will be discussed in Chapter 13.

3.1.7 Networking

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 one another through various communication lines, such as high-speed buses or telephone lines. 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 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 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.

Discussions concerning network and distributed systems are presented in Chapters 15 to 18.

3.1.8 Command-Interpreter System

One of the most important system 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 variously called the *control-card interpreter*, the *command-line interpreter*, the *shell* (in UNIX), and so on. Its function is quite simple: Get the next command statement and execute it.

Operating systems are frequently differentiated in the area of command interpretation, with a user-friendly interpreter making the system more agreeable to some users. An example of a user-friendly interface is the Macintosh interpreter, a window and menu system that is almost exclusively mouse-based. The user uses the mouse to point with the cursor to images (or *icons*) on the screen that represent programs, files, and system functions. Depending on the cursor location, clicking the mouse's button can invoke a program, select a file or directory (known as a *folder*), or pull down a menu containing commands. More powerful, complex, and difficult-to-learn interpreters are appreciated by other, more sophisticated users. On these interpreters, 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 UNIX shells run in this mode.

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. The operating system provides certain services to programs and to the users of those programs. The specific services provided will, of course, differ from one operating system to another, but there are some common classes that we can identify. 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 it. 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 rewind a tape drive, or blank the screen on a CRT). For efficiency and protection, users cannot control I/O devices directly. Therefore, the operating system must provide some means to do I/O.

- **File-system manipulation:** The file system is of particular interest. It should be obvious that programs need to read and write files. They also need to create and delete files by name.
- **Communications:** There are many circumstances in which one process needs to exchange information with another process. There are two major ways in which such communication can occur. The first takes place between processes executing on the same computer; the second takes place between processes 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), or 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 rather 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 there are multiple users or multiple jobs 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 be used to allocate plotters, modems, and other peripheral devices.
- **Accounting:** We want to keep track of which users use how much and what kinds of computer resources. This record keeping may be 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 its use. When several disjoint 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 or herself 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 breakins. 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 are usually listed in the manuals used by assembly-language programmers.

Some systems may 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, Bliss, BCPL, and PL/360 — have been defined to replace assembly language for systems programming. These languages allow system calls to be made directly. Some Pascal systems also provide an ability to make system calls directly from a Pascal program to the operating system. Most FORTRAN systems provide similar capabilities, often by a set of library routines.

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. Another approach, frequently used for batch systems, is to specify the names of the files with control cards. In

this case, there must be a mechanism for passing these parameters from the control cards to the executing program. On mouse-based and icon-based 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 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. There are also possible error conditions for each operation. When the program tries to open the input file, it may find that there is no file of that name 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 that there is already 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). Another option, in an interactive system, is to ask the user (a sequence of system calls to output the prompting message and to read the response from the terminal) whether to replace the existing file or to abort.

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 there was a hardware failure in the read (such as a parity error). The write operation may encounter various errors, depending on the output device (no more disk space, physical end of tape, printer out of paper, and so on).

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 last system call). As we can see, programs may make heavy use of the operating system.

Most users never see this level of detail, however. The run-time support system for most programming languages provides a much simpler interface. For example, a *write* statement in Pascal or FORTRAN probably is 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 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). Parameters also can 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 they do not limit the number or length of parameters being passed.

System calls can be roughly grouped into five major categories: *process control*, *file manipulation*, *device manipulation*, *information maintenance*, and *communications*. In Sections 3.3.1 to 3.3.5, we discuss briefly the types of system calls that may be provided by an operating system. Unfortunately, our description may seem somewhat shallow, as 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 and Job 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 circumstances, the operating system must

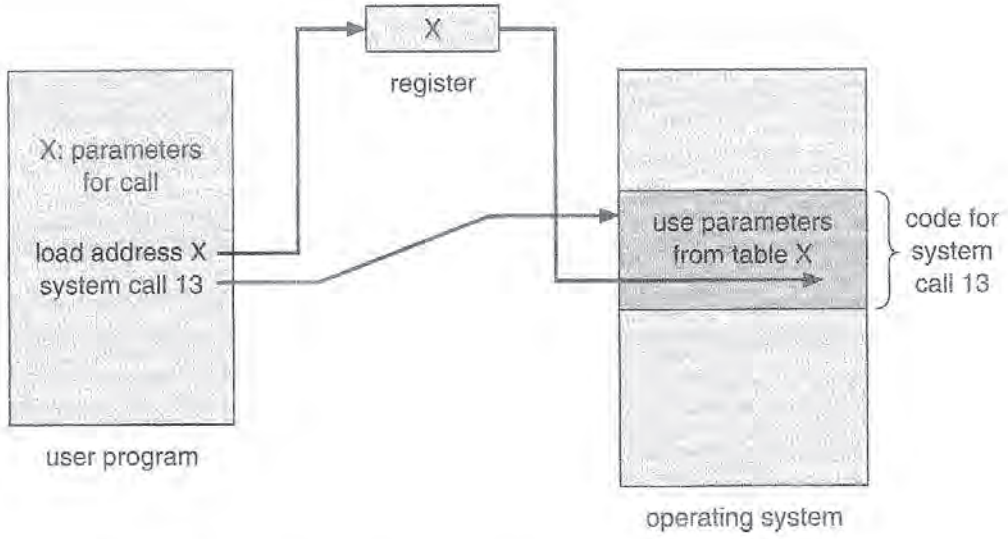


Figure 3.1 Passing of parameters as a table.

- 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 manipulation
  - create file, delete file
  - open, close
  - read, write, reposition
  - get file attributes, set file attributes
- Device manipulation
  - 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.

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, there is a system call 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 probably, 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 6.

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.

A time profile of a program is provided by many systems. 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.

There are so many facets of and variations in process and job control 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.

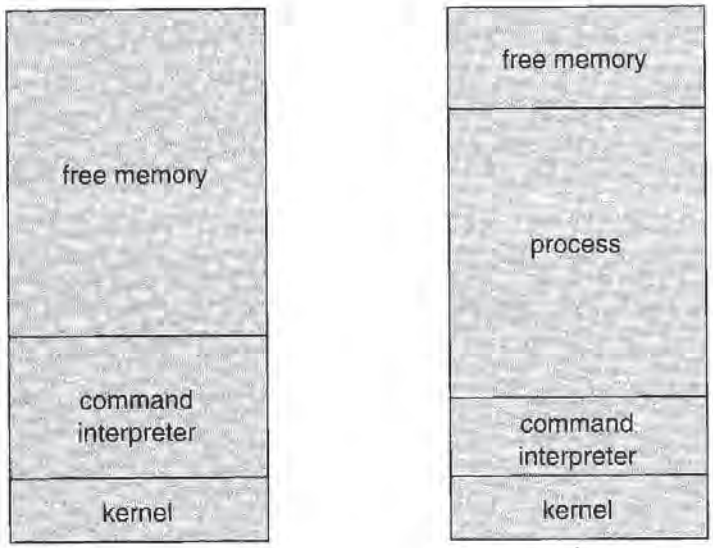


Figure 3.3 MS-DOS execution. (a) At system startup. (b) Running a program.

Berkeley UNIX, on the other hand, is an example of a multitasking system. When a user logs on to the system, a command interpreter (called a *shell*) of the user's choice is run. This shell is similar to the MS-DOS command interpreter (in fact, MS-DOS is modeled after UNIX) in that it accepts commands and executes programs that the user requests. However, since UNIX 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 was 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 user, because the shell is expecting input also. I/O is therefore done through files. 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.

3.3.2 File Manipulation

The file system will be discussed in more detail in Chapters 10 and 11. We can identify several common system calls dealing with files, however.

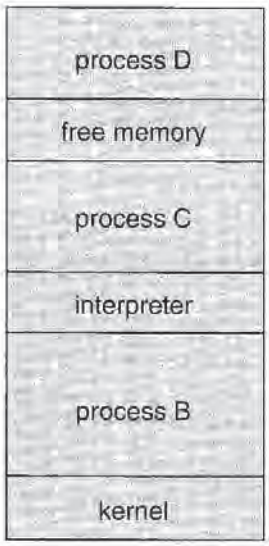


Figure 3.4 UNIX running multiple programs.

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** (rewinding or skipping 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 there are multiple users of the system, 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 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* 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,* which are system 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 two or more 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 6.

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 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, of course. Next is the operating system, then the systems programs, and finally the application programs. System programs provide a more convenient environment for program development and execution. Some of them are simply user interfaces to system calls, whereas others are considerably more complex. They can be divided into several categories:

- **File manipulation:** 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 FORTRAN, COBOL, Pascal, BASIC, C, and LISP) are often provided to the user with

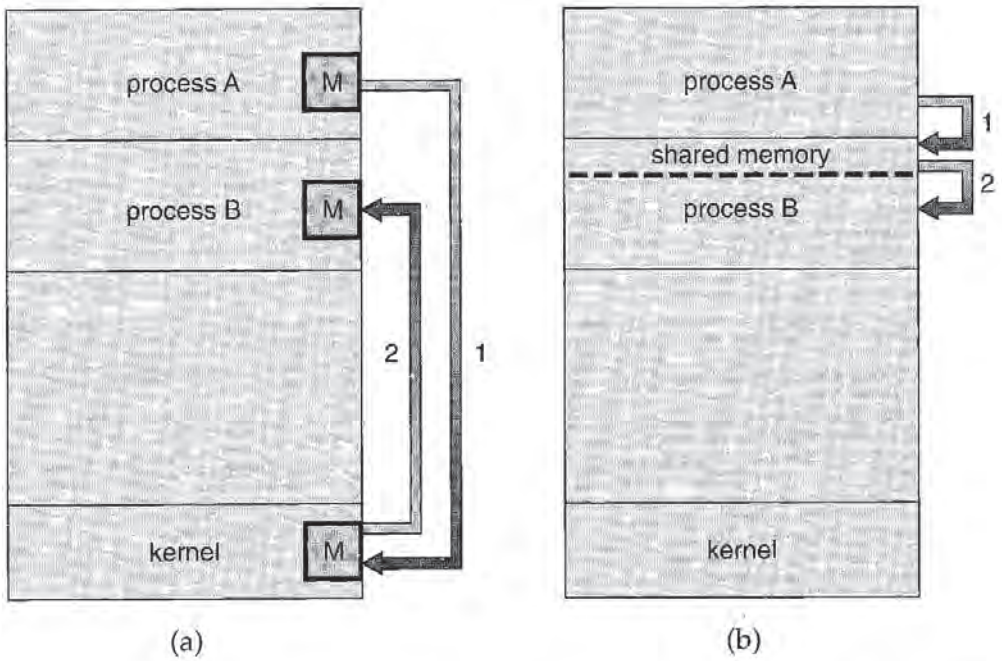


Figure 3.5 Communications models. (a) Message passing. (b) Shared memory.

the operating system. Many 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 each other's screens, to send larger messages as electronic mail, or to transfer files from one machine to another, and even to use other computers remotely as though these machines were local (known as *remote login*).
- **Application programs:** Most operating systems are supplied with programs that are useful to solve common problems, or to perform common operations. Such programs include compiler compilers, text formatters, plotting packages, database systems, spreadsheets, statistical-analysis packages, and games.

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. There are two general ways in which these commands can be implemented. 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 special systems 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, a command

delete *G*

would search for a file called *delete*, load the file into memory, and execute it with the parameter *G*. The function associated with the **delete** command would be defined completely by the code in the file *delete*. In this way, programmers can add new commands to the system easily by creating new

files of the proper name. The command-interpreter program, which can now be quite small, does not have to be changed for new commands to be added.

There are problems with this approach to the design of a command interpreter. Notice first that, 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 both 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 that were written at different times by different programmers.

The view of the operating system seen by most users is thus defined by the systems programs, rather than by the actual system calls. Consider IBM PC compatibles. Running the same MS-DOS operating system, the user could see a command-line–based command interpreter, or, could run the Windows program to invoke a graphical, Apple Macintosh–like interface. Both use the same set of system calls, but the calls look different and act in different ways. Consequently, this user 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 systems 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 to 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

There are numerous commercial systems that do not have a well-defined structure. Frequently, such operating systems started as small, simple,

and limited systems, and then grew beyond their original scope. An example of such a system is MS-DOS, the best-selling microcomputer operating system. MS-DOS 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 current structure.

Although MS-DOS does have some structure, its interfaces and levels of functionality are not well separated. For instance, applications programs are able to access the basic I/O routines to write directly to the display and disk drives. Such freedom leaves MS-DOS vulnerable to errant (or malicious) programs, causing entire system crashes or disk erasures when user programs fail. Of course, MS-DOS is also limited by the hardware on which it runs. Because the Intel 8088 for which it was written provides no dual mode and no hardware protection, the designers of MS-DOS had no choice but to leave the base hardware accessible.

Another example of limited structuring is the original UNIX operating system. UNIX is another system that initially was limited by hardware functionality. It consists of two separable parts: the kernel and the systems programs. The kernel is further separated into a series of interfaces and device drivers, which have been added and expanded over the years as UNIX has evolved. We can view the 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

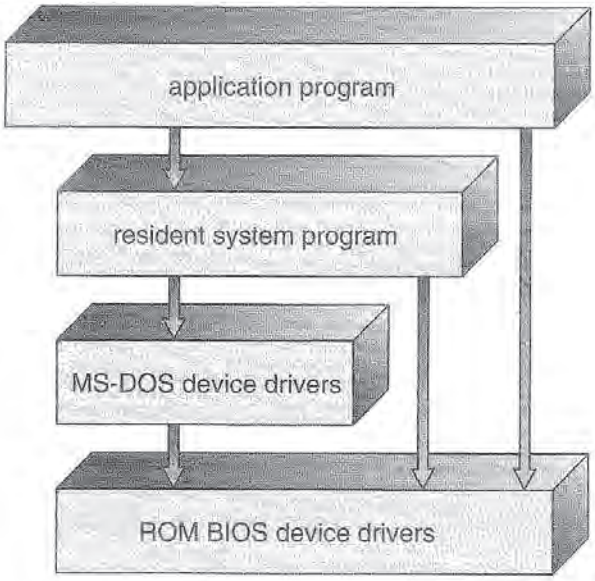


Figure 3.6 MS-DOS layer structure.

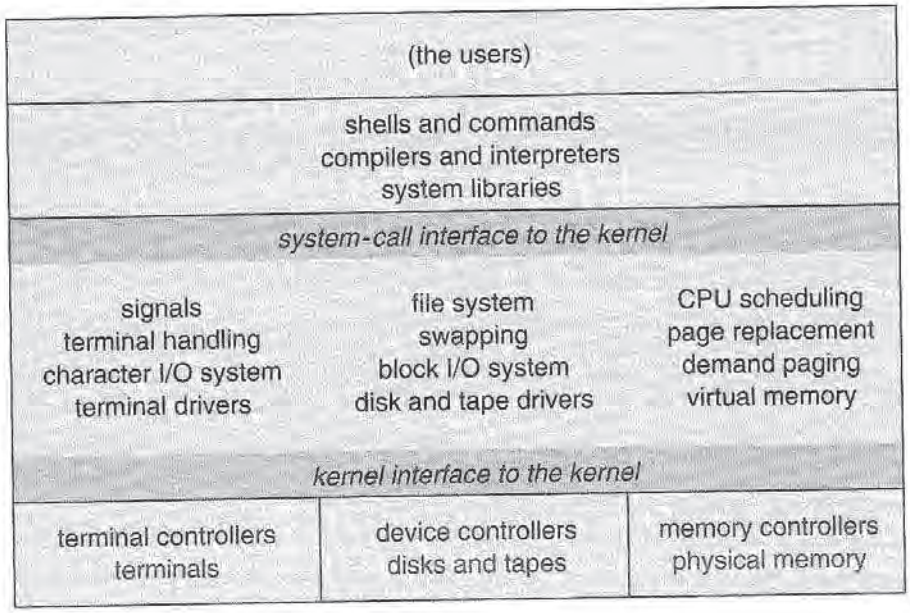


Figure 3.7 UNIX system structure.

of functionality to be combined into one level. Systems programs use the kernel-supported system calls to provide useful functions, such as compilation and file manipulation.

System calls define the *programmer interface* to UNIX; the set of systems programs commonly available defines the *user interface*. The programmer and user interfaces define the context that the kernel must support. Several versions of UNIX have been developed in which the kernel is partitioned further along functional boundaries. The AIX operating system, IBM's version of UNIX, separates the kernel into two parts. Mach, from Carnegie Mellon University, reduces the kernel to a small set of core functions by moving all nonessentials into systems and even into user-level programs. What remains is a *microkernel* operating system implementing only a small set of necessary primitives.

3.5.2 Layered Approach

These new UNIX versions are designed to use more advanced hardware. Given proper hardware support, operating systems may be broken into smaller, more appropriate pieces than those allowed by the original MS-DOS or UNIX. The operating system can then retain much greater control over the computer and the applications that make use of that computer. Implementors have more freedom to make changes to the inner workings of the system. Familiar techniques are used to aid in the creation of modular operating systems. Under the top-down approach, the overall functionality and features can be determined and separated into

components. Information hiding is also important, leaving programmers free to implement the low-level routines as they see fit, provided that the external interface of the routine stays unchanged and the routine itself performs the advertised task.

The modularization of a system can be done in many ways; the most appealing is the layered approach, which consists of breaking the operating system into a number of layers (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 operations that can manipulate those data. A typical operating-system layer — say layer M — is depicted in Figure 3.8. It consists of some data structures and a set of routines that can be invoked by higher-level layers. Layer M, in return, 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 (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 worked on, and so on. If an error is found during the debugging of a particular layer, we know that the error must be on that layer, because the layers below it are already debugged. Thus, the design and implementation of the system is simplified when the system is broken down into layers.

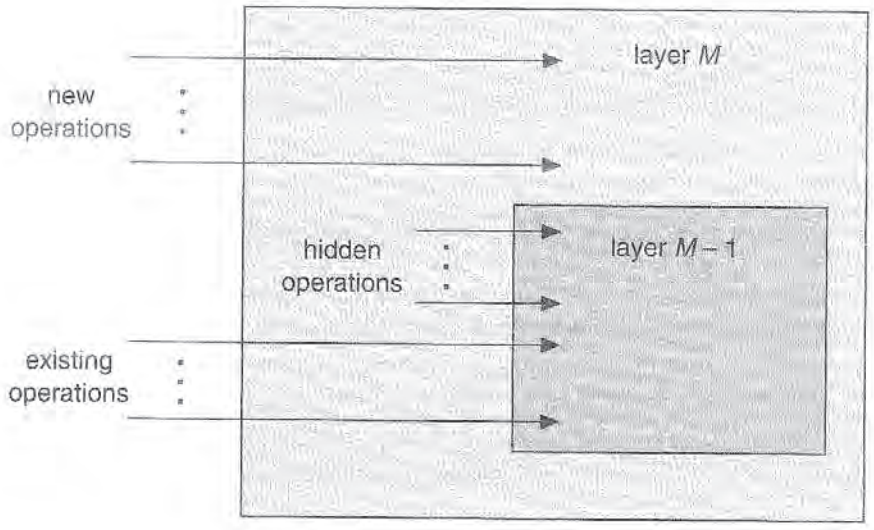


Figure 3.8 An operating-system layer.

Each layer is implemented using 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 layer approach to design was first used in the THE operating system at the Technische Hogeschool Eindhoven. The THE system was defined in six layers, as shown in Figure 3.9. The bottom layer was the hardware. The next layer implemented CPU scheduling. The next layer implemented memory management; the memory-management scheme was virtual memory (Chapter 9). Layer 3 contained the device driver for the operator's console. Because it, as well as the I/O buffering, was placed at layer 4, above memory management, the device buffers could be placed in virtual memory. The I/O buffering was also above the operator's console, so that I/O error conditions could be output to the operator's console.

This approach can be used in many ways. For example, the Venus system was also designed using a layered approach. The lower layers (0 to 4), dealing with CPU scheduling and memory management, were then put into microcode. This decision provided the advantages of additional speed of execution and a clearly defined interface between the microcoded layers and the higher layers (Figure 3.10).

The major difficulty with the layered approach involves the appropriate definition of the various layers. Because a layer can use only those layers that are at a lower level, careful planning is necessary. For example, the device driver for the backing store (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 backing store.

Other requirements may not be so obvious. The backing-store driver would normally be above the CPU scheduler, because the driver may need

layer 5:	user programs
layer 4:	buffering for input and output devices
layer 3:	operator-console device driver
layer 2:	memory management
layer 1:	CPU scheduling
layer 0:	hardware

Figure 3.9 THE layer structure.

layer 6:	user programs
layer 5:	device drivers and schedulers
layer 4:	virtual memory
layer 3:	I/O channel
layer 2:	CPU scheduling
layer 1:	instruction interpreter
layer 0:	hardware

Figure 3.10 Venus layer structure.

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, for a user program to execute an I/O operation, it executes a system call which is trapped to the I/O layer, which calls the memory-management layer, through to the CPU scheduling layer, and finally 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 and the net result is a system call that takes longer than one does 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. OS/2, a direct descendant of MS-DOS, was created to overcome the limitations of MS-DOS. OS/2 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. Contrast the MS-DOS structure to that shown in Figure 3.11. It should be clear that, 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.

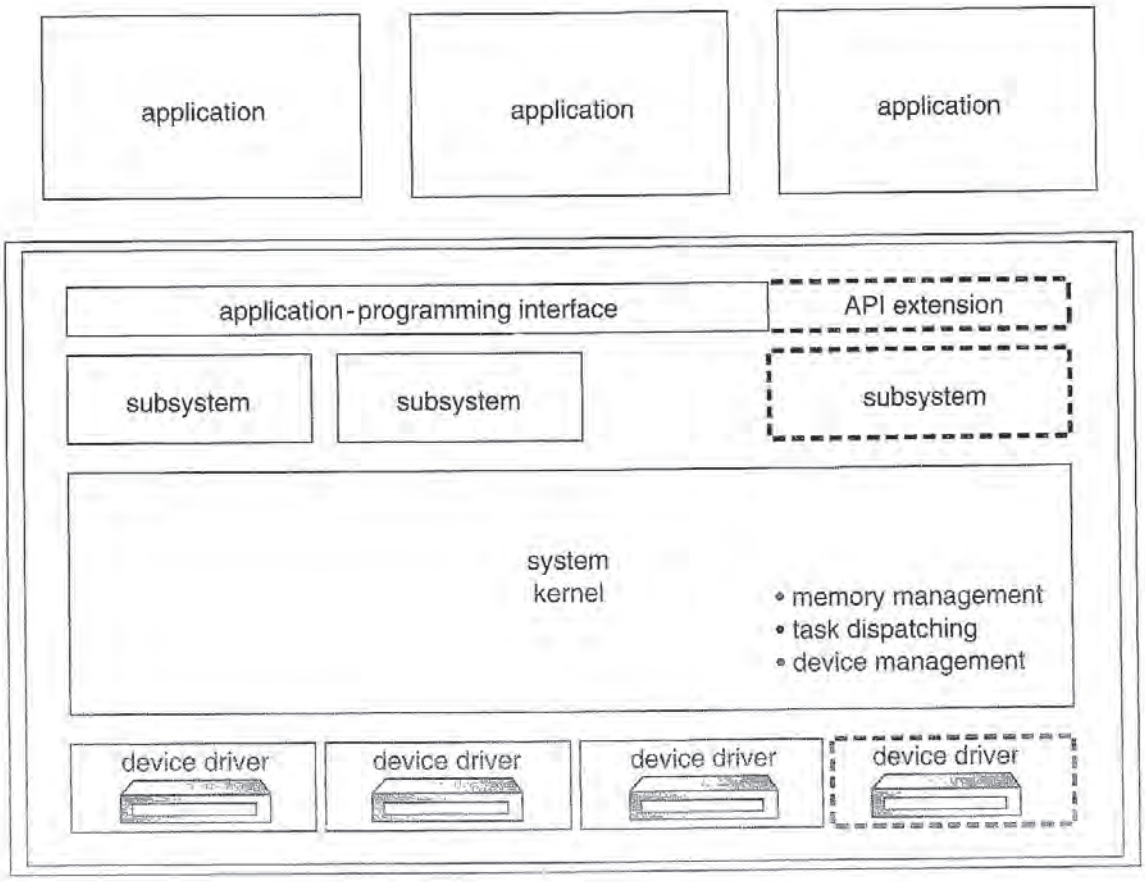


Figure 3.11 OS/2 layer structure.

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 systems 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 both are at the same level.

Some systems carry this scheme even 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 5) and virtual-memory techniques (Chapter 9), an operating system can create the illusion of multiple processes, each executing on its own processor with its own (virtual) memory. Of course, normally, the process has additional features, such as system calls and a file system, which are not provided by the bare hardware. The virtual-machine approach, on the other hand, does not provide any additional function, 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 3.12).

The resources of the physical computer are shared to create the virtual machines. CPU scheduling can be used to share the CPU and to create the appearance that users have their own processor. 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.

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

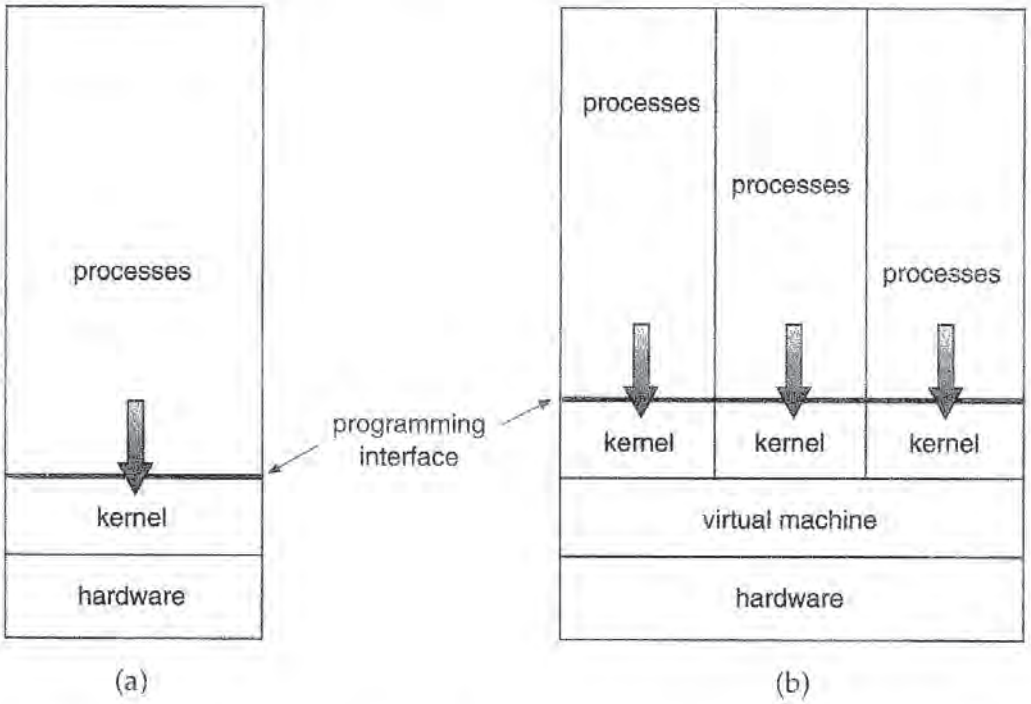


Figure 3.12 System models. (a) Nonvirtual machine. (b) Virtual machine.

drive to each virtual machine. Remember that the virtual-machine software itself will need substantial disk space to provide virtual memory and spooling. The solution is to provide virtual disks, which are identical in all respects except size. These are termed *minidisks* in IBM's VM operating system. The system implements each minidisk by allocating as many tracks as the minidisk needs on the physical disks. Obviously, the sum of the sizes of all minidisks must be less than the actual amount of physical disk space available.

Users thus are given their own virtual machine. 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 does not need to consider any user-support software. This arrangement may provide a useful partitioning of the problem of designing a multiuser interactive system into two smaller pieces.

Although the virtual machine concept is useful, it is difficult to implement. Much effort is required to provide an *exact* duplicate of the underlying machine. Remember, for example, 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. The virtual user mode is also a physical user mode. 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 is, of course, 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 (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.

The virtual-machine concept has several advantages. Notice that in this environment there is complete protection of the various system resources. Each virtual machine is completely isolated from all other virtual machines, so there are no security problems. On the other hand, there is no direct sharing of resources. To provide sharing, two approaches 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 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, it is difficult to be sure that a change in one point will not cause obscure bugs in some other part. This situation can be particularly dangerous because of the power of the operating system. 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 a physical machine. Normal system operation seldom needs to be disrupted for system development.

Virtual machines are coming back into fashion as a means of solving system compatibility problems. For instance, there are thousands of programs available for MS-DOS on Intel CPU-based systems. Computer vendors like Sun Microsystems and Digital Equipment Corporation (DEC) use other, faster processors, but would like their customers to be able to run these MS-DOS programs. The solution is to create a virtual Intel machine on top of the native processor. An MS-DOS program is run in this environment, and its Intel instructions are translated into the native instruction set. MS-DOS is also run in this virtual machine, so the program can make its system calls as usual. The net result is a program which appears to be running on an Intel-based system but is really executing on a very different processor. If the processor is sufficiently fast, the MS-DOS program will run quickly even though every instruction is being translated into several native instructions for execution.

3.7 ■ System Design and Implementation

In this section, we discuss the problems of designing and implementing a system. There are, of course, no complete solutions to the design problems, but there are approaches that 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 to use, easy to learn, easy to use, 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.

There is 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.

The specification and design of an operating system is a highly creative task. No mere textbook can solve that problem. There are, however, general principles that have been suggested. *Software engineering* is the general field for these principles; certain ideas from this field are especially applicable to operating systems.

3.7.2 Mechanisms and Policies

One important principle is the separation of *policy* from *mechanism*. Mechanisms determine *how* to do something. In contrast, policies decide *what* will be done. For example, a mechanism for ensuring CPU protection is the timer construct (see Section 2.5). The decision of for how long the timer is set for a particular user, on the other hand, is a policy decision.

The separation of policy and mechanism is important for flexibility. Policies are likely to change from place to place or time to 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 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 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 are important for all resource allocation and scheduling problems. Whenever it is necessary to decide whether or not to allocate a resource, a policy decision must be made. 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. However, that is generally no longer true. Operating systems can now be written in higher-level languages.

The first system that was 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. The major claimed disadvantages are reduced speed and increased storage requirements. Although no compiler can produce consistently more efficient code than can an expert assembly-language programmer, a compiler often can produce code at least as good as that written by the average assembly-language programmer. In addition, replacing the compiler with a better compiler will uniformly 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, which is written mostly in C, on the other hand, is available on a number of different CPUs, including Intel 80X86, Motorola 680X0, SPARC, and Mips RX000.

As with other systems, major performance improvements are more likely to be the result of better data structures and algorithms than of cleaner coding. In addition, although operating systems are large systems, 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 can be replaced with assembly-language equivalents.

To identify bottlenecks, we must be able to monitor the 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 could also be run as input for a simulation of a suggested improved system. Traces also can be useful in finding errors in operating-system behavior.

An alternative possibility is to compute and display performance measures in real time. This approach may allow the system operators to become more familiar with system behavior and to modify system operation in real time.

3.8 ■ System Generation

It is possible to 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. This process is known as *system generation (SYSGEN)*.

The operating system is normally distributed on tape or disk. To generate a system, we use a special program. The SYSGEN program reads from a file or asks the operator for information concerning the specific configuration of the hardware system:

- What CPU is to 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 CPU-scheduling algorithm is desired, what the maximum number of processes to be supported is, and so on.

Once this information is defined, it can be used in several ways. At one extreme, it can be used to modify a copy of the source code of the operating system. The operating system would then be completely compiled. Data declarations, initializations, and constants, along with conditional compilation, would 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 could cause the creation of tables and the selection of modules from a precompiled library. These modules would be linked together to form the generated operating system. Selection would allow the library to contain the device drivers for all supported I/O devices, but only those actually needed would be linked into the operating system. Because the system would not be recompiled, system generation would be faster, but might result in a system with more generality than was actually needed.

At the other extreme, it would be possible to construct a system that was completely table driven. All the code would always be a part of the system, and selection would occur at execution time, rather than at compile or link time. System generation involves simply creating the appropriate tables to describe the 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 it? The procedure of starting a computer by loading the kernel is known as *booting* the system. On most systems, there is 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 memory, and start its execution. Some systems, like IBM PCs running MS-DOS, turn this into a two-step process by having a very simple bootstrap loader load a more complex boot program, which in turn loads the kernel. Booting a system is further discussed in Section 12.3.2 and Chapter 19.

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 provides a mechanism for a user to issue a request without writing a program. Commands may come from cards (in a batch system) or directly from a terminal (in an interactive or time-shared system). Systems programs provide another mechanism for satisfying 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 manipulation. Higher-level requests, satisfied by the command interpreter or systems 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. It is important that the goals of the system be well defined before the design begins. The type of system desired is the foundation for choices among various algorithms and strategies that will be necessary.

Since an operating system is large, modularity is important. The design of a system as a sequence of layers is considered an important design technique. The virtual-machine concept takes the layered approach to heart and treats the kernel of the operating system and the hardware as though they were all hardware. Even other operating systems may be loaded on top of this virtual machine.

Throughout the entire operating-system design cycle, we must be careful to separate policy decisions from implementation details. 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 also 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 What is the purpose of system programs?

3.9 What is the main advantage of the layered approach to system design?

3.10 What is the main advantage for an operating-system designer of using a virtual-machine architecture? What is the main advantage for a user?

3.11 Why is the separation of mechanism and policy a desirable property?

3.12 Consider the experimental Synthesis operating system, which has an assembler incorporated within the kernel. To optimize system-call performance, the kernel assembles routines within kernel space to minimize the path 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 this approach to kernel design and to system-performance optimization.

Bibliographic Notes

Command languages can be seen as special-purpose programming languages. Brunt and Tuffs [1976] argued that a command language should provide a rich set of functions; Frank [1976] argued for a more limited, simpler command language. An excellent case study is the UNIX shell, as described by Bourne [1978].

Dijkstra [1968] advocated the layered approach to operating-system design. The THE system was described in [Dijkstra 1968]; the Venus system was described in [Liskov 1972].

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. A computer architecture for supporting level-structured operating systems was described by Bernstein and Siegel [1975].

The first operating system to provide a virtual machine was the CP/67 on an IBM 360/67 and is described by Meyer and Seawright [1970]. CP/67 provided each user with a virtual 360 Model 65, including I/O devices. The commercially available IBM VM/370 operating system was derived from CP/67 and is described by Seawright and MacKinnon [1979], Holley et al. [1979], and Creasy [1981]. Hall et al. [1980] promoted the use of virtual machines for increasing operating-system portability. Jones [1978] suggested the use of virtual machines to enforce the isolation of processes for protection purposes. General discussions concerning virtual machines have been

presented by Hendricks and Hartmann [1979], MacKinnon [1979], and Schultz [1988].

MS-DOS, Version 3.1, is described in [Microsoft 1986]. Windows/NT is described by Custer [1993]. The Apple Macintosh operating system is described in [Apple 1987]. Berkeley UNIX is described in [CSRG 1986]. The standard AT&T UNIX system V is described in [AT&T 1986]. A good description of OS/2 is given in [Iacobucci 1988]. Mach is introduced in [Accetta et al. 1986], and AIX is presented in [Loucks and Sauer 1987]. The experimental Synthesis operating system is discussed in [Massalin and Pu 1989].

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 can potentially execute concurrently.

The operating system is responsible for the following activities in connection with process 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 of 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 hindrance to the 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 MS-DOS and Macintosh OS, a user may be able to run several programs at one time: one interactive and several batch programs.

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 spooling. In many respects, all of 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 the term *process* has superseded it.

4.1.1 The Process

Informally, a *process* is a program in execution. The execution of a process must progress in a sequential fashion. That is, at any time, at most one instruction is executed on behalf of the process.

A process is more than the program code (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. A process generally also includes the process *stack*, containing temporary data (such as subroutine parameters, return addresses, and temporary variables), and a *data section* containing 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 copies of the mail program, or the same user may invoke many copies of the editor program. Each of these is

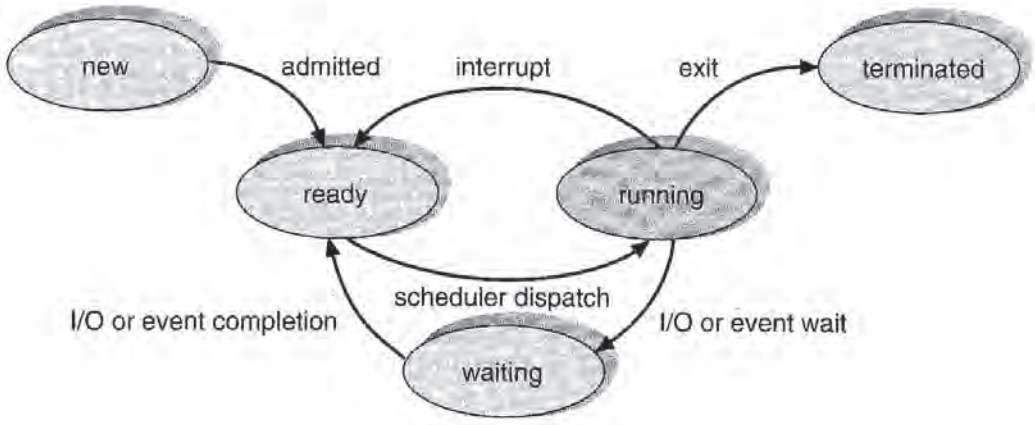


Figure 4.1 Diagram of process state.

a separate process, and, although the text sections are equivalent, the data sections will vary. It is also common to have a process that spawns many processes as it runs. This issue will be further discussed 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.
- **Terminated:** The process has finished execution.

These names are arbitrary, and vary between operating systems. The states that they represent are found on all systems, however. Certain operating systems also distinguish among more finely delineating process states. It is important to realize that only one process can be *running* on any processor at any instant. Many processes may be *ready* and *waiting*, however. 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:

- **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).

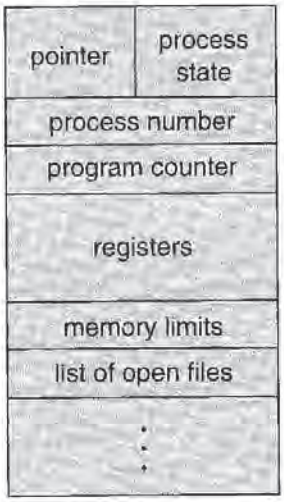


Figure 4.2 Process control block.

- **CPU scheduling information:** This information includes a process priority, pointers to scheduling queues, and any other scheduling parameters. (Chapter 5 describes process scheduling.)
- **Memory-management information:** This information may include 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 8).
- **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 (such as tape drives) 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.

4.2 ■ Process Scheduling

The objective of multiprogramming is to have some process running at all times, 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. For a uniprocessor system, there will never be more than one running process. If there are more processes, the rest will have to wait until the CPU is free and can be rescheduled.

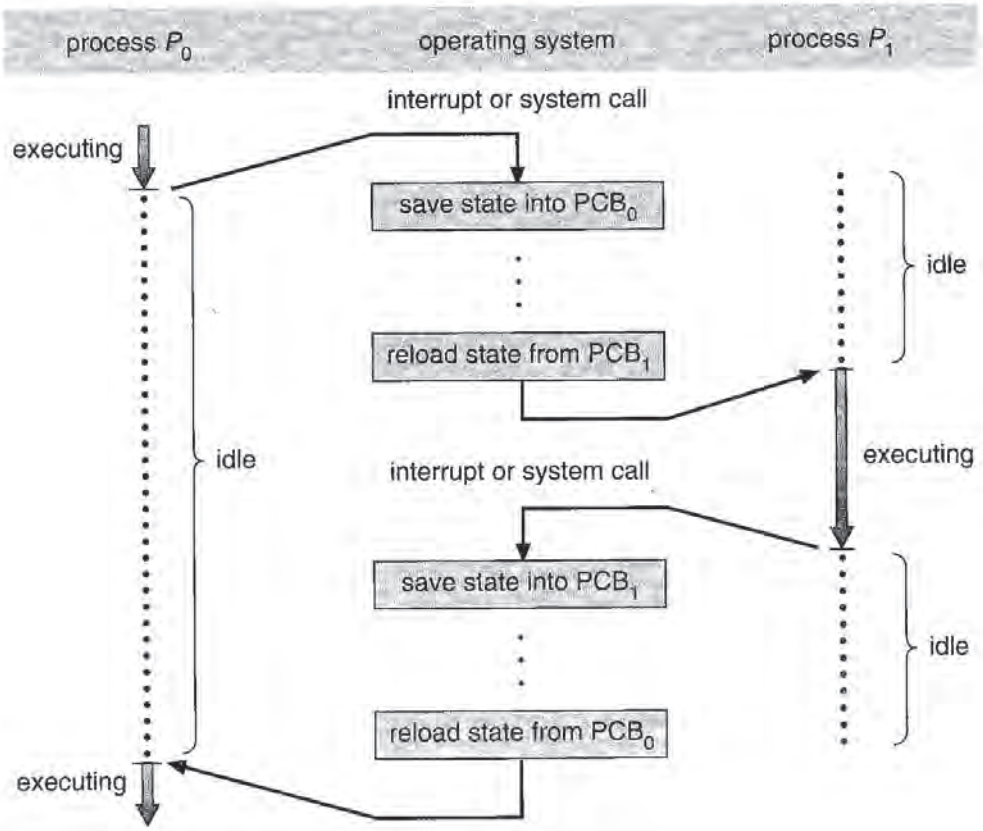


Figure 4.3 Diagram showing CPU switch from process to process.

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 will contain pointers to the first and last PCBs in the list. Each PCB has a pointer field that points to the next process in the ready queue.

There are also other queues in the system. When a process is allocated the CPU, it executes for awhile 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 there are many processes in the system, the disk 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).

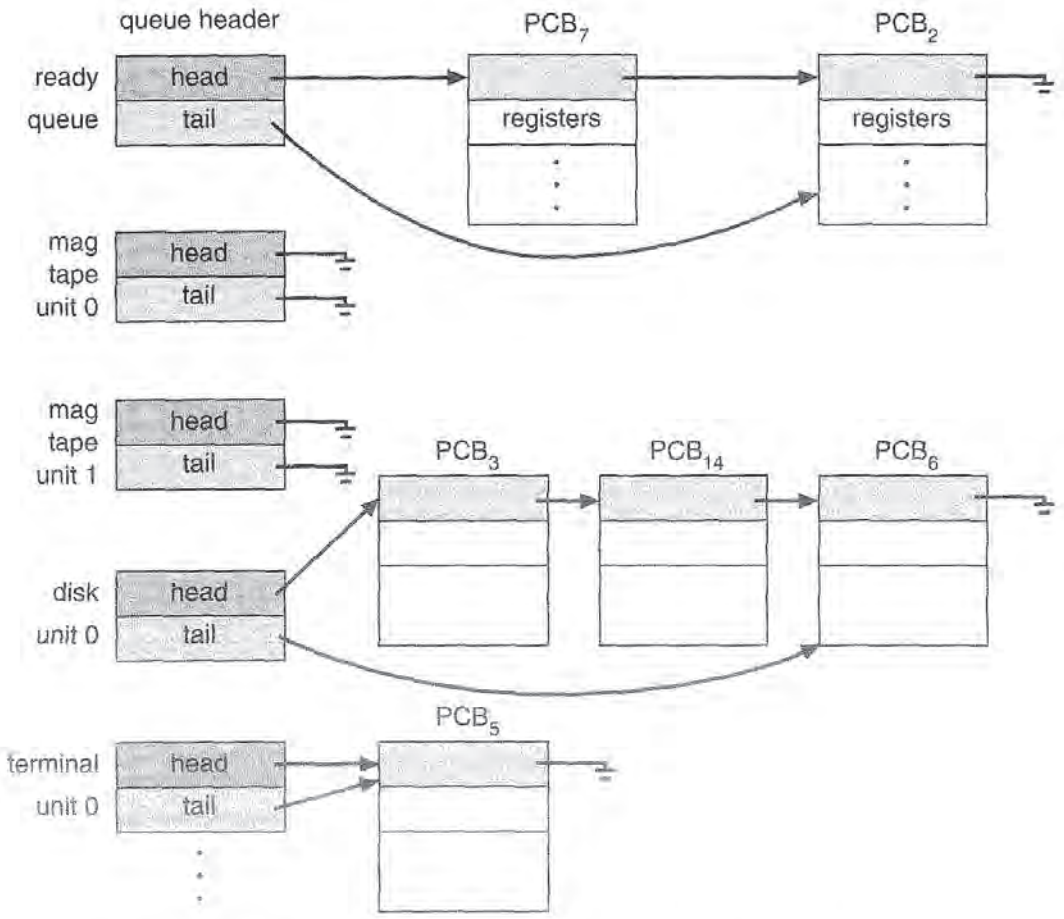


Figure 4.4 The ready queue and various I/O device queues.

A common representation for a discussion 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*) and is given the CPU. Once the process is allocated 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.

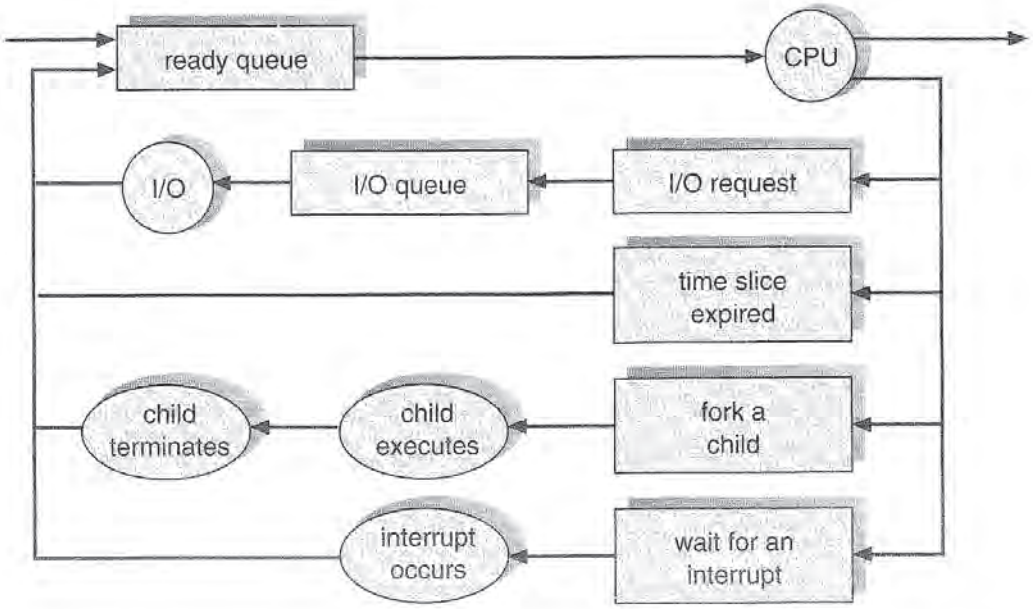


Figure 4.5 Queueing-diagram representation of process scheduling.

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 processes from these queues in some fashion. The selection process is carried out by the appropriate *scheduler*.

In a batch system, there are often more processes 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 quite 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 short duration of time between executions, the short-term scheduler must

be very 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 (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 decide which process should be selected for execution.

It is important that the long-term scheduler make a careful selection. In general, most processes can be described as either I/O bound or CPU bound. An *I/O-bound process* is one that spends more of its time doing I/O than it spends doing computations. A *CPU-bound process*, on the other hand, is one that generates I/O requests infrequently, using more of its time doing computation than an I/O-bound process uses. It is important that the long-term scheduler 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 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, and will do something else.

Some operating systems, such as time-sharing systems, may introduce an additional, intermediate level of scheduling. This *medium-term scheduler* is diagrammed in Figure 4.6. The key idea behind a medium-term scheduler is that sometimes it can be advantageous to remove processes from memory (and from active contention for the CPU), and thus to reduce 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 swapped in later 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 more detail in Chapter 8.

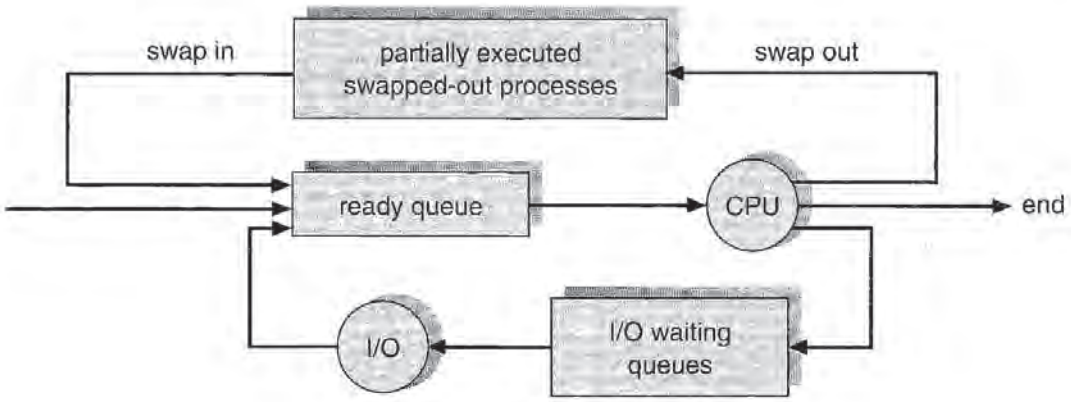


Figure 4.6 Addition of medium-term scheduling to the queueing diagram.

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*. 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 which must be copied, and the existence of special instructions (such as a single instruction to load or store all registers). Typically, the speed ranges from 1 to 1000 microseconds.

Context-switch times are highly dependent on hardware support. For instance, some processors (such as the DECSYSTEM-20) provide multiple sets of registers. A context switch simply includes changing the pointer to the current register set. Of course, if there are more active processes than there are 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 shall see in Chapter 8, 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 the amount of work needed to do it, depend on the memory-management method of the operating system. As we shall see in Section 4.5, context switching has become such a performance bottleneck that new structures (threads) are being used to avoid it whenever possible.

4.3 ■ Operation on Processes

The processes in the system can execute concurrently, and must be created and deleted dynamically. Thus, the operating system must provide a mechanism 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 (CPU time, memory, files, I/O devices) to accomplish its task. When a process creates a subprocess, the 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.

In addition to the various physical and logical resources that a process obtains when it is created, initialization data (input) may be passed along by the parent process to the child process. For example, consider a process whose function is to display the status of a file, say *F1*, 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 *F1*, and it will execute using that datum to obtain the desired information. It may also get the name of the

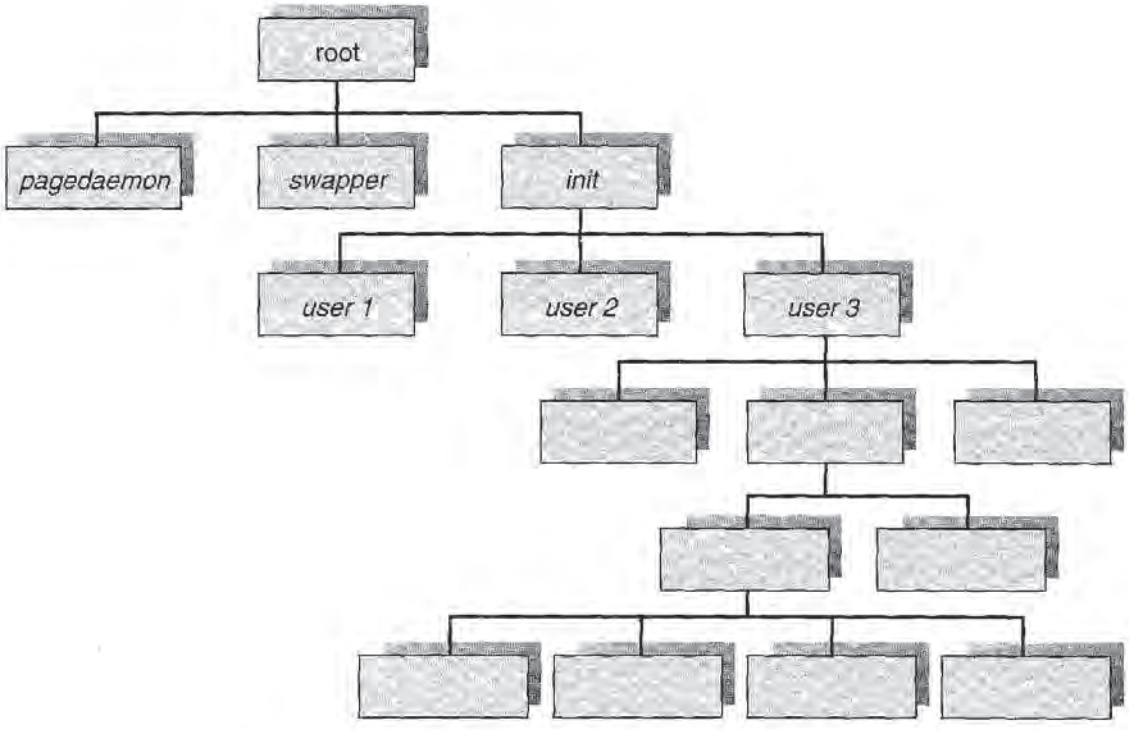


Figure 4.7 A tree of processes on a typical UNIX system.

output device. Some operating systems pass resources to child processes. On such a system, the new process may get two open files, *F1* 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:

- The parent continues to execute concurrently with its children.
- 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:

- The child process is a duplicate of the parent process.
- 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** with one difference: The return code for the **fork** is zero for the new (child) process, whereas the (nonzero) process identifier of the child is returned to the parent.

Typically, the **execve** system call is used after a **fork** by one of the two processes to replace the process' memory space with a new program. The **execve** system call loads a binary file into memory (destroying the memory image of the program containing the **execve** 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 DEC VMS operating system, in contrast, creates a new process, loads a specified program into that process, and starts it running. The Microsoft 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 last 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

fork system call). All of the resources of the process, including physical and virtual memory, open files, and I/O buffers, are deallocated by the operating system.

There are additional circumstances when termination occurs. A process can cause the termination of another process via an appropriate system call (for example, **abort**). Usually, such a system call can be invoked by only the parent of the process that is to be terminated. Otherwise, users could arbitrarily kill each other's jobs. Note that a parent 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:

- The child has exceeded its usage of some of the resources it has been allocated.
- 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.

To determine the first case, the parent must have a mechanism to inspect the state of its children.

Many systems, including VMS, do not allow a child to exist if its parent has terminated. In such systems, if a process terminates (either normally or abnormally), then all its children must also be terminated. This phenomenon is referred to as *cascading termination* and is normally initiated by the operating system.

To illustrate process execution and termination, let us consider again the UNIX system. In UNIX, a process may terminate by using the **exit** system call, and its parent process may wait for that event 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 the possibly many children has terminated. If the parent terminates, however, all the children are terminated by the operating system. Without a parent, UNIX does not know to whom to report the activities of a child.

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.

There are several reasons for providing an environment that allows process cooperation:

- **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. Notice that 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, as was discussed in Chapter 3.
- **Convenience:** Even an individual user may have many tasks to work on at one time. For instance, a user may be editing, printing, and compiling in parallel.

Concurrent execution that requires cooperation among the processes requires mechanisms to allow processes to communicate with each other (Section 4.6), and to synchronize their actions (Chapter 6).

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 that there is 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 be either provided by the operating system through the use of IPC (Section 4.6), or 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:

```
var n;
type item = ... ;
var buffer: array [0..n−1] of item;
in, out: 0..n−1;
```

with *in, out* initialized to the value 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 **mod** *n* = *out*.

The code for the producer and consumer processes follows. The *no-op* is a do-nothing instruction. Thus, **while** *condition* **do** *no-op* simply tests the condition repetitively until it becomes false.

The producer process has a local variable *nextp,* in which the new item to be produced is stored:

```
repeat
    ...
    produce an item in nextp
    ...
    while in+1 mod n = out do no-op;
    buffer[in] := nextp;
    in := in+1 mod n;
until false;
```

The consumer process has a local variable *nextc,* in which the item to be consumed is stored:

```
repeat
    while in = out do no-op;
    nextc := buffer[out];
    out := out+1 mod n;
    ...
    consume the item in nextc
    ...
until false;
```

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

In Chapter 6, we shall discuss in great detail how synchronization among cooperating processes can be implemented effectively in a shared-memory environment.

4.5 ■ Threads

Recall that a process is defined by the resources it uses and by the location at which it is executing. There are many instances, however, in which it would be useful for resources to be shared and accessed concurrently. This situation is similar to the case where a **fork** system call is invoked with a new program counter, or thread of control, executing within the same address space. This concept is so useful that several new operating systems are providing a mechanism to support it through a *thread* facility.

4.5.1 Thread Structure

A *thread,* sometimes called a *lightweight process (LWP),* is a basic unit of CPU utilization, and consists of a program counter, a register set, and a stack space. It shares with peer threads its code section, data section, and operating-system resources such as open files and signals, collectively known as a *task.* A traditional or *heavyweight* process is equal to a task with one thread. A task does nothing if no threads are in it, and a thread must be in exactly one task. The extensive sharing makes CPU switching among peer threads and the creation of threads inexpensive, compared with context switches among heavyweight processes. Although a thread context switch still requires a register set switch, no memory-management–related work need be done.

Also, some systems implement *user-level threads* in user-level libraries, rather than via system calls, so thread switching does not need to call the operating system, and to cause an interrupt to the kernel. Switching between user-level threads can be done independently of the operating system and, therefore, very quickly. Thus, blocking a thread and switching to another thread is a reasonable solution to the problem of how a server can handle many requests efficiently. User-level threads do have disadvantages, however. For instance, if the kernel is single-threaded, then any user-level thread executing a system call will cause the entire task to wait until the system call returns.

We can grasp the functionality of threads by comparing multiple-thread control with multiple-process control. With multiple processes, each process operates independently of the others; each process has its

own program counter, stack pointer, and address space. This type of organization is useful when the jobs performed by the processes are unrelated. For instance, in single-CPU operating systems, a file server may have to block while waiting for disk access. System performance would improve if another server process could operate while the first one was blocked, yet because they would have to occupy the same address space, it is not possible to create a second, independent server process.

Threads operate, in many respects, in the same manner as processes. Threads can be in one of several states: ready, blocked, running, or terminated. Like processes, threads share the CPU, and only one thread at a time is active (running). A thread within a process executes sequentially, and each thread has its own stack and program counter. Threads can create child threads, and can block waiting for system calls to complete; if one thread is blocked, another thread can run. However, unlike processes, threads are not independent of one another. Because all threads can access every address in the task, a thread can read or write over any other thread's stacks. This structure does not provide protection between threads. Such protection, however, should not be necessary. Whereas processes may originate from different users, and may be hostile to one another, only a single user can own an individual task with multiple threads. The threads, in this case, probably would be designed to assist one another, and therefore would not require mutual protection. Figure 4.8 depicts a task with multiple threads.

Let us return to our example of the blocked file-server process in the single-process model. In this scenario, no other server process can execute until the first process is unblocked. By contrast, in the case of a task that contains multiple threads, while one server thread is blocked and waiting, a second thread in the same task could run. In this application, the cooperation of multiple threads that are part of the same job confers the advantages of higher throughput and improved performance. Other applications, such as the producer–consumer problem, require sharing a common buffer and so also benefit from this feature of thread utilization: The producer and consumer could be threads in a task. Little overhead is needed to switch between them, and, on a multiprocessor system, they could execute in parallel on two processors for maximum efficiency.

Threads provide a mechanism that allows sequential processes to make blocking system calls while also achieving parallelism. To illustrate the advantage of this mechanism, we shall consider writing a file server in a system where threads are not available. We have already seen that, in a single-threaded file server, the server process must carry a request to completion before acquiring new work. If the request involves waiting for disk access, the CPU is idle during the wait. Hence, the number of requests per second that can be processed is much less than with parallel execution. Without the option of multiple threads, a system designer seeking to minimize the performance slowdown of single-threaded

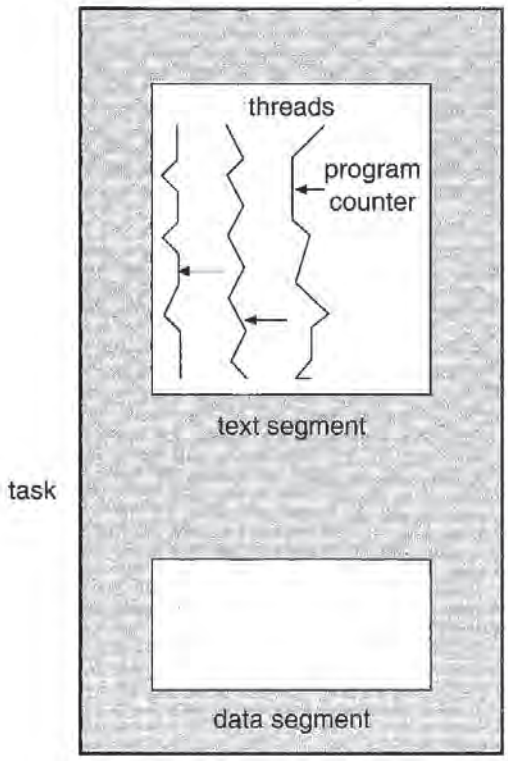


Figure 4.8 Multiple threads within a task.

processes would need to mimic the parallel structure of threads with the use of heavyweight process. She could do so, but at the cost of a complex nonsequential program structure.

The abstraction presented by a group of lightweight processes is that of multiple threads of control associated with several shared resources. There are many alternatives regarding threads; we mention a few of them briefly. Threads can be supported by the kernel (as in the Mach and OS/2 operating systems). In this case, a set of system calls similar to those for processes is provided. Alternatively, they can be supported above the kernel, via a set of library calls at the user level (as is done in Project Andrew from CMU).

Why should an operating system support one version or the other? User-level threads do not involve the kernel, and therefore are faster to switch among than kernel-supported threads. However, any calls to the operating system cause the entire process to wait, because the kernel schedules only processes (having no knowledge of threads), and a process which is waiting gets no CPU time. Scheduling can also be unfair. Consider two processes, one with 1 thread (process *a*) and the other with 100 threads (process *b*). Each process generally receives the same number of time slices, so the thread in process *a* runs 100 times as fast as a thread in process *b*. On systems with kernel-supported threads, switching among the threads is more time-consuming because the kernel (via an interrupt)

must do the switch. Each thread may be scheduled independently, however, so process *b* could receive 100 times the CPU time that process *a* receives. Additionally, process *b* could have 100 system calls in operation concurrently, accomplishing far more than the same process would on a system with only user-level thread support.

Because of the compromises involved in each of these two approaches to threading, some systems use a hybrid approach in which both user-level and kernel-supported threads are implemented. Solaris 2 is such a system and is described below.

Threads are gaining in popularity because they have some of the characteristics of heavyweight processes but can execute more efficiently. There are many applications where this combination is useful. For instance, the UNIX kernel is usually single tasking: Only one task can be executing code in the kernel at a time. Many problems, such as synchronization of data access (locking of data structures while they are being modified) are avoided, because only one process is allowed to be doing the modification. Mach, on the other hand, is multithreaded, allowing the kernel to service many requests simultaneously. In this case, the threads themselves are synchronous: another thread in the same group may run only if the currently executing thread relinquishes control. Of course, the current thread would relinquish control only when it was not modifying shared data. On systems on which threads are asynchronous, some explicit locking mechanism must be used, just as in systems where multiple processes share data. Process synchronization is discussed in Chapter 6.

4.5.2 Example: Solaris 2

An examination of the thread system in a current operating system should help us to clarify many issues. For this purpose, we choose Solaris 2, a version of UNIX, which until 1992 supported only traditional heavyweight processes. It has been transformed into a modern operating system with support for threads at the kernel and user levels, symmetric multiprocessing, and real-time scheduling.

Solaris 2 supports user-level threads, as described in Section 4.5.1. They are supported by a library for their creation and scheduling, and the kernel knows nothing of these threads. Solaris 2 expects potentially thousands of user-level threads to be vying for CPU cycles.

Solaris 2 defines an intermediate level of threads as well. Between user-level threads and kernel-level threads are lightweight processes. Each task (still called a "process" in SunOS nomenclature) contains at least one LWP. These LWPs are manipulated by the thread library. The user-level threads are multiplexed on the LWPs of the process, and only user-level

threads currently connected to LWPs accomplish work. The rest are either blocked or waiting for an LWP on which they can run.

All operations within the kernel are executed by standard kernel-level threads. There is a kernel-level thread for each LWP, and there are some kernel-level threads which run on the kernel's behalf and have no associated LWP (for instance, a thread to service disk requests). The entire thread system is depicted in Figure 4.9. Kernel-level threads are the only objects scheduled within the system (see Chapter 5). Some kernel-level threads are multiplexed on the processors in the system, whereas some are tied to a specific processor. For instance, the kernel thread associated with a device driver for a device connected to a specific processor will run only on that processor. By request, a thread can also be *pinned* to a processor. Only that thread runs on the processor, with the processor allocated to only that thread (see the rightmost thread in Figure 4.9).

Consider this system in operation. Any one task may have many user-level threads. These user-level threads may be scheduled and switched among kernel-supported lightweight processes without the intervention of the kernel. No context switch is needed for one user-level thread to block and another to start running, so user-level threads are extremely efficient.

These user-level threads are supported by lightweight processes. Each LWP is connected to exactly one kernel-level thread, whereas each user-level thread is independent of the kernel. There may be many LWPs in a task, but they are needed only when threads need to communicate with

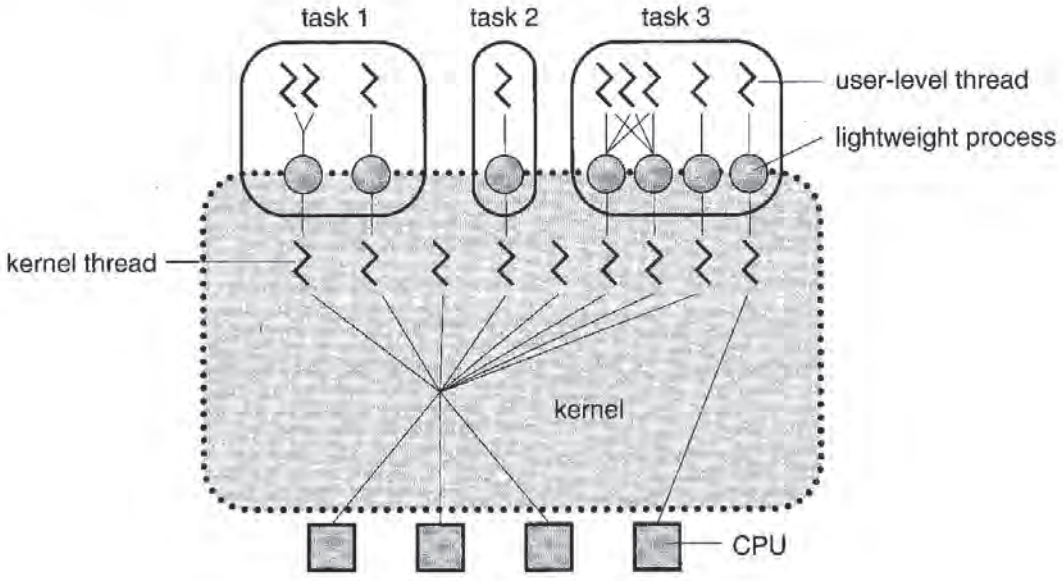


Figure 4.9 Threads in Solaris 2.

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 could be occurring 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 us 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 (usually 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 the task containing that thread has only one LWP, the whole task blocks until the I/O completes. This behavior is the same as that of a process under an older version of the operating system.

With Solaris 2, a task no longer must block while waiting for I/O to complete. The task may have multiple LWPs; if one blocks, the others can continue to execute within the task.

We conclude this example by examining the resource needs of each of these thread types.

- A kernel thread has only a small data structure and a stack. Switching between kernel threads does not require changing memory access information, and therefore is relatively fast.
- An LWP contains a process control block with register data, accounting information, and memory information. Switching between LWPs therefore requires quite a bit of work and is relatively slow.
- A user-level thread needs only a stack and a program counter: no kernel resources are required. The kernel is not involved in scheduling these user-level threads; therefore, switching among them is fast. There may be thousands of these user-level threads, but all the kernel will ever see is the LWPs in the process that support these user-level threads.

4.6 ■ 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 explicitly written 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. Interprocess-communication is best provided by a message system. Message systems can be defined in many different ways. Message-passing systems also have other advantages, as will be shown in Chapter 16.

Note that the shared-memory and message-system communication schemes are not mutually exclusive, and could be used simultaneously within a single operating system or even a single process.

4.6.1 Basic Structure

The function of a message system is to allow processes to communicate with each other without the need to resort to shared variables. 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 physical implementation is straightforward. This restriction, however, makes the task of programming more difficult. On the other hand, variable-sized messages require a more complex physical 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 the issues of its logical implementation, such as its logical properties. Some basic implementation questions are these:

- How are links established?
- Can a link be associated with more than two processes?
- How many links can there be between every pair of processes?
- What is the capacity of a link? That is, does the link have some buffer space? If it does, how much?
- What is the size of messages? Can the link accommodate variable-sized or only fixed-sized messages?
- Is a link unidirectional or bidirectional? That is, if a link exists between P and Q, can messages flow in only one direction (such as only from P to Q) or in both directions?

The definition of *unidirectional* must be stated more carefully, since a link may be associated with more than two processes. Thus, we say that a link is unidirectional only if each process connected to the link can either send or receive, but not both, and each link has at least one receiver process connected to it.

In addition, there 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

For the remainder of this section, we elaborate on these types of message systems.

4.6.2 Naming

Processes that want to communicate must have a way to refer to each other. They can use either *direct communication* or *indirect communication*, as we shall discuss in the next two subsections.

4.6.2.1 Direct Communication

In the direct-communication discipline, 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 follows:

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.
- Between each pair of processes, there exists exactly one link.
- The link may be unidirectional, but is usually bidirectional.

To illustrate, let us present a solution to the producer–consumer problem. To allow the producer and consumer processes to run concurrently, we allow the producer to produce one item while the consumer is consuming another item. When the producer finishes generating an item, it **sends** that item to the consumer. The consumer gets that item via the **receive** operation. If an item has not been produced yet, the consumer process must wait until an item is produced. The *producer* process is defined as

```
repeat
    ...
    produce an item in nextp
    ...
    send(consumer,nextp);
until false;
```

The *consumer* process is defined as

```
repeat
    receive(producer,nextc);
    ...
    consume the item in nextc
    ...
until false;
```

This scheme exhibits a symmetry in addressing; that is, both the sender and the receiver processes have to name each 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 of these schemes (symmetric and asymmetric) 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.

4.6.2.2 Indirect Communication

With indirect communication, the messages are sent to and received from *mailboxes* (also referred to as *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 the processes have a shared 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 they have a shared mailbox.
- A link may be associated with more than two processes.
- Between each pair of communicating processes, there may be a number of different links, each link corresponding to one mailbox.
- A link may be either unidirectional or bidirectional.

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? This question can be resolved in a variety of ways:

- 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 system. If the mailbox is owned by a process (that is, the mailbox is attached to or defined as part of the process), then we distinguish between the owner (who can only receive messages through this mailbox) and the user of the mailbox (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 (via exception handling, described in Section 4.6.4).

There are various ways to designate the owner and users of a particular mailbox. One possibility is to allow a process to declare variables of type *mailbox*. The process that declares a mailbox is that mailbox's owner. Any other process that knows the name of this mailbox can use this mailbox.

On the other hand, a mailbox that is owned by the operating system has an existence of its own. It is independent, and is not attached to any particular process. The operating system provides a mechanism that allows a process:

- To create a new mailbox
- To send and receive messages through the mailbox
- To destroy 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. Processes may also share a mailbox through the process-creation facility. For example, if process P created mailbox A, and then created a new process Q, P and Q may share mailbox A. Since all processes with access rights to a mailbox may ultimately terminate, after some time a mailbox may no longer be accessible by any process. In this case, the operating system should reclaim whatever space was used for the mailbox. This task may require some form of *garbage collection* (see Section 10.3.5), in which a separate operation occurs to search for and deallocate memory that is no longer in use.

4.6.3 Buffering

A link has some capacity that determines the number of messages that can reside in it temporarily. This property can be viewed as a queue of messages attached to the link. Basically, there are three ways that such a queue can be implemented:

- **Zero capacity:** The queue has maximum length 0; thus, the link cannot have any messages waiting in it. In this case, the sender must wait until the recipient receives the message. The two processes must be synchronized for a message transfer to take place. This synchronization is called a *rendezvous*.

- **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 be delayed 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 is never delayed.

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

We note that, in the nonzero-capacity cases, a process does not know whether a message has arrived at its destination after the **send** operation is completed. If this information is crucial for the computation, the sender must communicate explicitly with the receiver to find out whether the latter received the message. For example, suppose process P sends a message to process Q and can continue its execution only after the message is received. Process P executes the sequence

send(*Q* ,*message*);
receive(*Q*, *message*);

Process Q executes

receive(*P*, *message*);
send(*P*, "acknowledgment");

Such processes are said to communicate *asynchronously*.

There are special cases that do not fit directly into any of the categories that we have discussed:

- The process sending a message is never delayed. However, if the receiver has not received the message before the sending process sends another message, the first message is lost. The advantage of this scheme is that large messages do not need to be copied more than once. The main disadvantage is that the programming task becomes more difficult. Processes need to synchronize explicitly, to ensure both that messages are not lost and that the sender and receiver do not manipulate the message buffer simultaneously.
- The process sending a message is delayed until it receives a reply. This scheme was adopted in the *Thoth* operating system. In this system, messages are of fixed size (eight words). A process P that sends a message is blocked until the receiving process has received the

message and has sent back an eight-word reply by the **reply**(*P*, *message*) primitive. The reply message overwrites the original message buffer. The only difference between the **send** and **reply** primitives is that a **send** causes the sending process to be blocked, whereas the **reply** allows both the sending process and the receiving process to continue with their executions immediately.

This synchronous communication method can be expanded easily into a full-featured *remote procedure call (RPC)* system. An RPC system is based on the realization that a subroutine or procedure call in a single-processor system acts exactly like a message system in which the sender blocks until it receives a reply. The message is then like a subroutine call, and the return message contains the value of the subroutine computed. The next logical step, therefore, is for concurrent processes to be able to call each other as subroutines using RPC. In fact, we shall see in Chapter 16 that RPCs can be used between processes running on separate computers to allow multiple computers to work together in a mutually beneficial way.

4.6.4 Exception Conditions

A message system is particularly useful in a distributed environment, where processes may reside at different sites (machines). In such an environment, the probability that an error will occur during communication (and processing) is much larger than in a single-machine environment. In a single-machine environment, messages are usually implemented in shared memory. If a failure occurs, the entire system fails. In a distributed environment, however, messages are transferred by communication lines, and the failure of one site (or link) does not necessarily result in the failure of the entire system.

When a failure occurs in either a centralized or distributed system, some error recovery (exception-condition handling) must take place. Let us discuss briefly some of the exception conditions that a system must handle in the context of a message scheme.

4.6.4.1 Process Terminates

Either a sender or a receiver may terminate before a message is processed. This situation will leave messages that will never be received or processes waiting for messages that will never be sent. We consider two cases here:

1. A receiver process P may wait for a message from a process Q that has terminated. If no action is taken, P will be blocked forever. In this case, the system may either terminate P or notify P that Q has terminated.
2. Process P may send a message to a process Q that has terminated. In the automatic-buffering scheme, no harm is done; P simply continues

with its execution. If P needs to know that its message has been processed by Q, it must program explicitly for an acknowledgment. In the no-buffering case, P will be blocked forever. As in case 1, the system may either terminate P or notify P that Q has terminated.

4.6.4.2 Lost Messages

A message from process P to process Q may become lost somewhere in the communications network, due to a hardware or communication-line failure. There are three basic methods for dealing with this event:

1. The operating system is responsible for detecting this event and for resending the message.
2. The sending process is responsible for detecting this event and for retransmitting the message, if it so wants.
3. The operating system is responsible for detecting this event; it then notifies the sending process that the message has been lost. The sending process can proceed as it chooses.

It is not always necessary to detect lost messages. In fact, some network protocols specify that messages are unreliable, whereas some guarantee reliability (see Chapter 15). The user must specify (that is, either notify the system, or program this requirement itself) that such a detection should take place.

How do we detect that a message is lost? The most common detection method is to use *timeouts*. When a message is sent out, a reply message, acknowledging reception of the message, is always sent back. The operating system or a process may then specify a time interval during which it expects the acknowledgment message to arrive. If this time period elapses before the acknowledgment arrives, the operating system (or process) may assume that the message is lost, and the message is resent. It is possible, however, that a message did not get lost, but simply took a little longer than expected to travel through the network. In this case, we may have multiple copies of the same message flowing through the network. A mechanism must exist to distinguish between these various types of messages. This problem is discussed in more detail in Chapter 16.

4.6.4.3 Scrambled Messages

The message may be delivered to its destination, but be scrambled on the way (for example, because of noise in the communications channel). This case is similar to the case of a lost message. Usually, the operating system will retransmit the original message. Checksums (such as parity or CRC) are commonly used to detect this type of error.

4.6.5 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 mailbox is used by the kernel 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*. RPCs are executed via *msg_rpc*, which sends a message and waits for exactly one return message from the sender.

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 quite 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 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 actually be put in the mailbox, a 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 last 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 (1) any mailbox in a mailbox set, or (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 18, 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 mailer to the receiver. The Mach message system attempts to avoid double copy operations by using virtual-memory management techniques (Chapter 9). 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 for only intrasystem messages. The Mach operating system is discussed in detail in Chapter 20.

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's current activity. Each process may be in one of the following states: *new*, *ready*, *running*, *waiting*,

or *halted*. 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. There are two major classes of queues in an operating system: 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 (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 (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 a mechanism 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, there exist two complementary communication schemes: 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 then rests with the operating system itself. These two schemes are not mutually exclusive, and could be used simultaneously within a single operating system.

Cooperating processes that directly share a logical address space can be implemented as lightweight processes or threads. A *thread* is a basic unit of CPU utilization, and it shares with peer threads its code section, data section, and operating-system resources, collectively known as a *task*. A task does nothing if no threads are in it, and a thread must be in exactly one task. The extensive sharing makes CPU switching among peer threads and thread creation inexpensive, compared with context switches among heavyweight processes.

■ Exercises

4.1 Several popular microcomputer operating systems provide little or no means of concurrent processing. Discuss the 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 a register set, and all the register sets are in use?

4.4 What two advantages do threads have over multiple processes? What major disadvantage do they have? Suggest one application that would benefit from the use of threads, and one that would not.

4.5 What resources are used when a thread is created? How do they differ from those used when a process is created?

4.6 Describe the actions taken by a kernel to context switch

a. Among threads.

b. Among processes.

4.7 What are the differences between user-level threads and kernel-supported threads? Under what circumstances is one type "better" than the other?

4.8 The correct producer–consumer algorithm presented in Section 4.6 allows only $n - 1$ buffers to be full at any time. Modify the algorithm to allow all the buffers to be utilized fully.

4.9 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 either from mailbox A or from mailbox B (or from both)?

c. A **receive** operation makes a process wait until the mailbox is nonempty. Either devise a scheme that allows a process to wait until a mailbox is empty, or explain why such a scheme cannot exist.

4.10 Consider an operating system that supports both the IPC and RPC schemes. Give examples of problems that could be solved with each type of scheme. Explain why each problem is best solved by the method that you specify.

Bibliographic Notes

Doeppner [1987] discussed early work in implementing threads at the user level. Thread performance issues were discussed in Anderson et al. [1989]. Anderson et al. [1991] continued this work 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,

The IBM OS/2 operating system is a multithreaded operating system that runs on personal computers [Kogan and Rawson 1988]. The *Synthesis* high-performance kernel uses threads as well [Massalin and Pu 1989]. The implementation of threads in Mach was described in Tevanian et al. [1987a]. Birrell [1989] discussed programming with threads. Debugging multithreaded applications continues to be a difficult problem that is under investigation. Caswell and Black [1990] implemented a debugger in Mach.

Sun Microsystem's Solaris 2 thread structure was described in Eykholt et al. [1992]. The user-level threads were detailed in Stein and Shaw [1992]. Peacock [1992] discussed the multithreading of the file system in Solaris 2.

The subject of interprocess communication was discussed by Brinch Hansen [1970] with respect to the RC 4000 system. The interprocess communication facility in the Thoth operating system was discussed by Cheriton et al. [1979]; the one for the Accent operating system was discussed by Rashid and Robertson [1981]; the one for the Mach operating system was discussed by Accetta et al. [1986]. Schlichting and Schneider [1982] discussed asynchronous message-passing primitives. The IPC facility implemented at the user level was described in Bershad et al. [1990].

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 the issues of procedure calls versus message-passing communication.

CHAPTER 5



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.

5.1 ■ Basic Concepts

The objective of multiprogramming is to have some process running at all times, to maximize CPU utilization. For a uniprocessor system, there will never be more than one running process. If there are more processes, the rest will have to 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 just sit idle. All this waiting time is wasted; no useful work is accomplished. 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. Every time one process has to wait, another process may take over the use of the CPU.

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.

5.1.1 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 back and forth between these two states. Process execution begins with a *CPU burst*. That is followed by an *I/O burst*, which is followed by 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 5.1).

The durations of these CPU bursts have been measured. Although they vary greatly from process to process and computer to computer, they tend to have a frequency curve similar to that shown in Figure 5.2. The curve is generally characterized as exponential or hyperexponential. There is a large

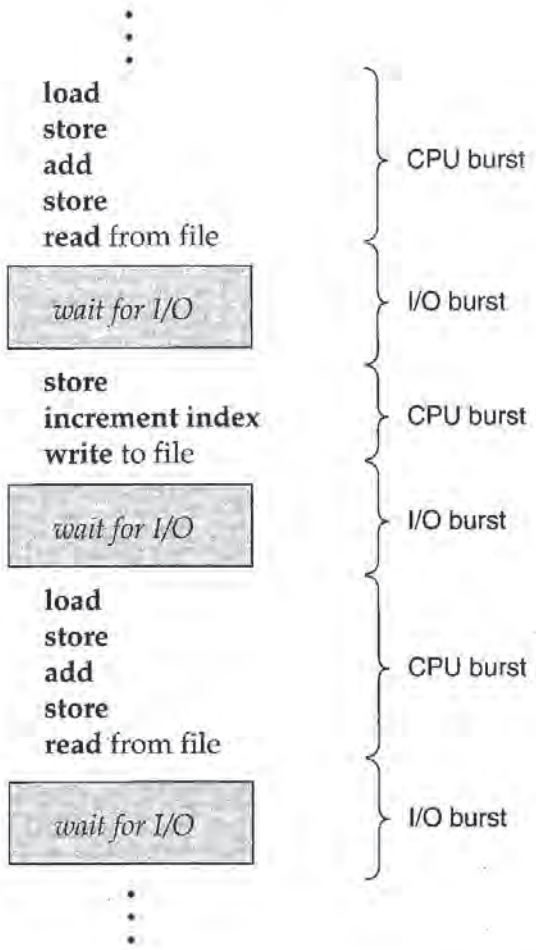


Figure 5.1 Alternating sequence of CPU and I/O bursts.

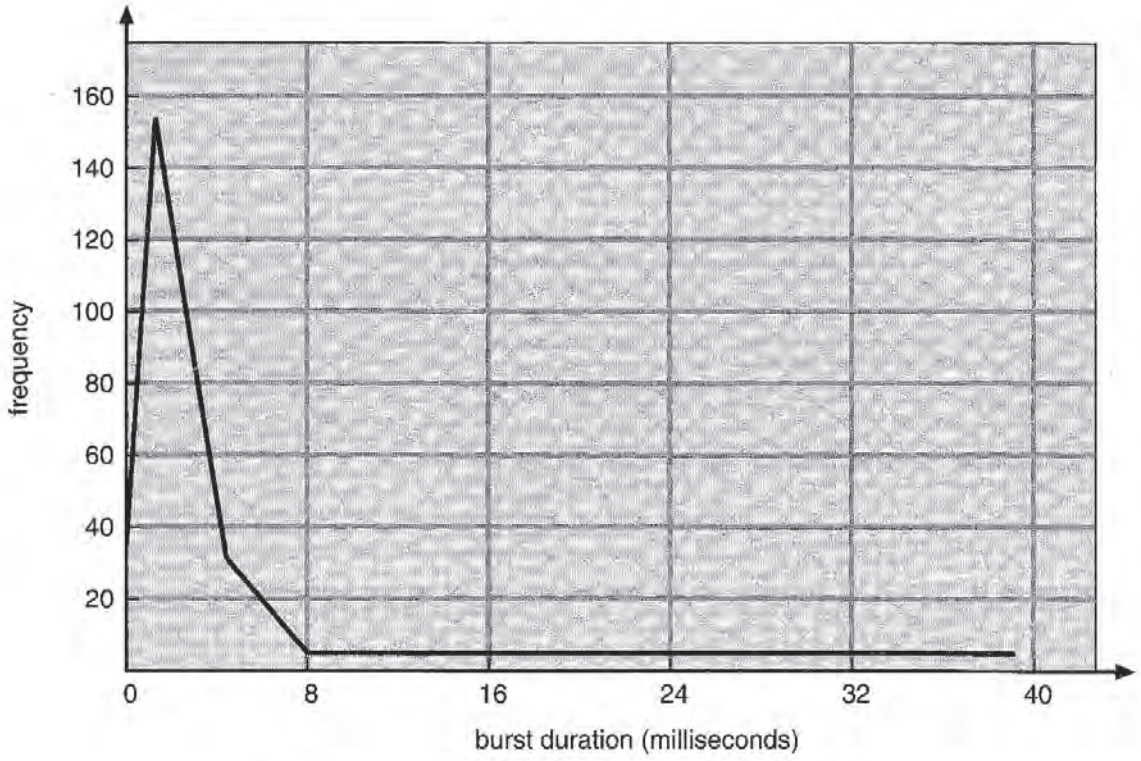


Figure 5.2 Histogram of CPU-burst times.

number of short CPU bursts, and there is a small number of 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 CPU bursts. This distribution can be important in the selection of an appropriate CPU scheduling algorithm.

5.1.2 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.

Note that 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 PCBs of the processes.

5.1.3 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

For 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, for 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 environment. 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 6.

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 ensues. 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 5.4 and 5.5.

In the case of UNIX, there are still sections of code at risk. Because interrupts can, by definition, occur at any time, and because interrupts 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, since otherwise input might be lost or output overwritten. So that these sections of code are not accessed concurrently by several processes, they disable interrupts at entry and reenable interrupts at exit.

5.1.4 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*.

5.2 ■ 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. Which characteristics are used for comparison can make a substantial difference in the determination of the best algorithm. Criteria that are used 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, then work is being done. One measure of work is the number of processes that are completed per time unit, called *throughput*. For long processes, this rate may be one 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 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.

It is desirable 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, there are circumstances when it is desirable 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.

It has also been suggested that, for interactive systems (such as time-sharing systems), it is more important to minimize the *variance* in the response time than it is to minimize the average response time. A system with reasonable and *predictable* response time may be considered more desirable than is 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 5.6.

5.3 ■ Scheduling Algorithms

CPU scheduling deals with the problem of deciding which of the processes in the ready queue is to be allocated the CPU. There are many different CPU scheduling algorithms. In this section, we describe several of these algorithms.

5.3.1 First-Come, First-Served Scheduling

By far the simplest CPU scheduling algorithm is the *first-come, first-served scheduling (FCFS)* 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 it is important that each user 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.

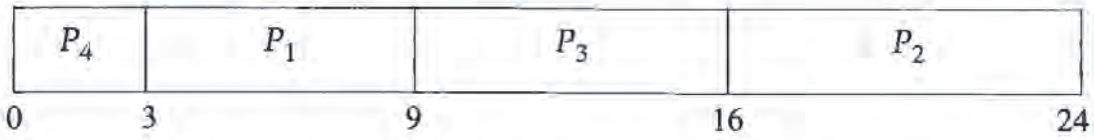
5.3.2 Shortest-Job-First Scheduling

A different approach to CPU scheduling is the *shortest-job-first (SJF)* 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, 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 (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 nth 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

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

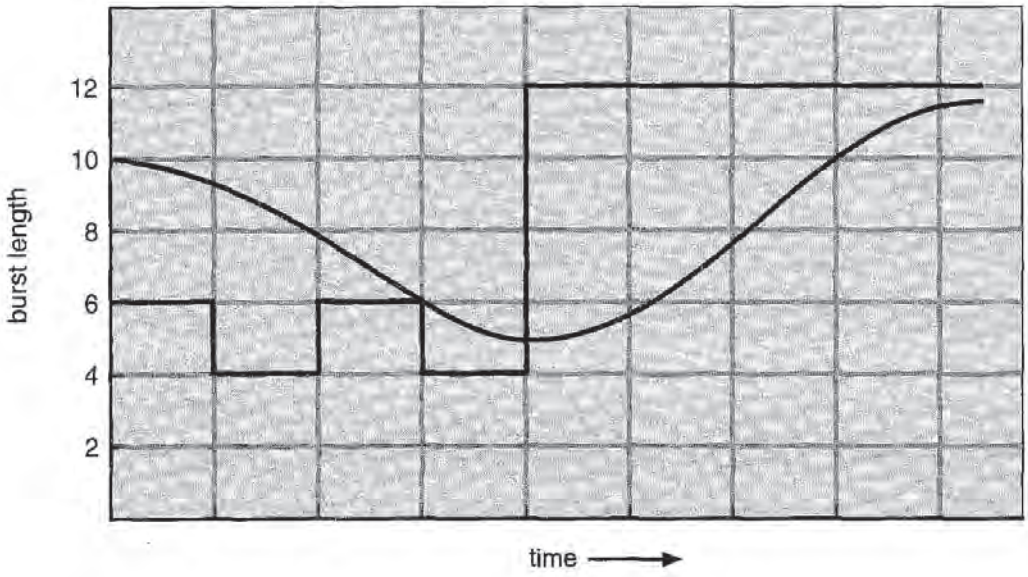
This formula defines an *exponential average*. The value of t_n contains our most recent information; τ_n stores the past history. The parameter α controls the relative weight of recent and past history in our prediction. If $\alpha = 0$, then $\tau_{n+1} = \tau_n$, and recent history has no effect (current conditions are assumed to be transient); if $\alpha = 1$, then $\tau_{n+1} = t_n$, and only the most recent CPU burst matters (history is assumed to be old and irrelevant). More commonly, $\alpha = 1/2$, so recent history and past history are equally weighted. Figure 5.3 shows an exponential average with $\alpha = 1/2$. The initial τ_0 can be defined as a constant or as an overall system average.

To understand the behavior of the exponential average, we can expand the formula for τ_{n+1} by substituting for τ_n, to find

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

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



CPU burst (t_i)	6	4	6	4	13	13	13	...
"guess" (τ_i)	10	8	6	6	5	9	11	12	...

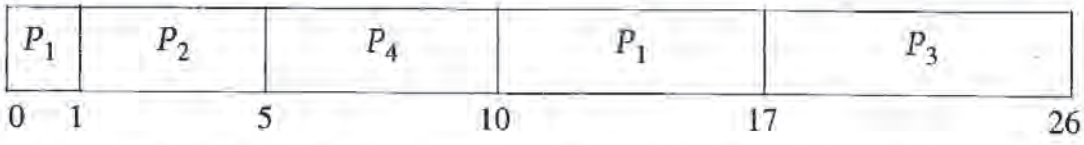
Figure 5.3 Prediction of the length of the next CPU burst.

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 5.75 milliseconds.

5.3.3 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 (τ): $p = 1/\tau$. 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 4095. 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 assume that low numbers 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, ..., 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	3
P_4	1	4
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 is 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 0 (low) to 127 (high), we could increment the priority of a waiting process by 1 every 15 minutes. Eventually, even a process with an initial priority of 0 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 0 process to age to a priority 127 process.

5.3.4 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.

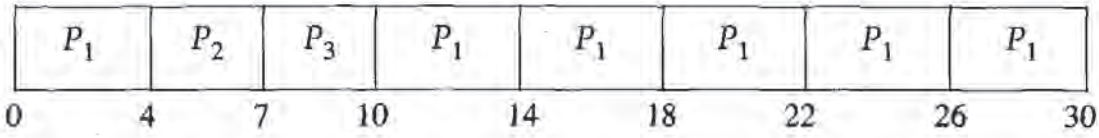
One of two things will then happen. The process may have a CPU burst of less than 1 time quantum. In this case, the process itself will release the CPU voluntarily. The scheduler will then proceed to the next process in the ready queue. Otherwise, if the CPU burst of the currently running process is 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 one. (Actually, since the processor was much faster than memory and each instruction referenced memory, the processors were not much slower than a single processor would have been.)

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 a context switch. If the time quantum is 1 time unit, then nine context switches will occur, slowing the execution of the process accordingly (Figure 5.4).

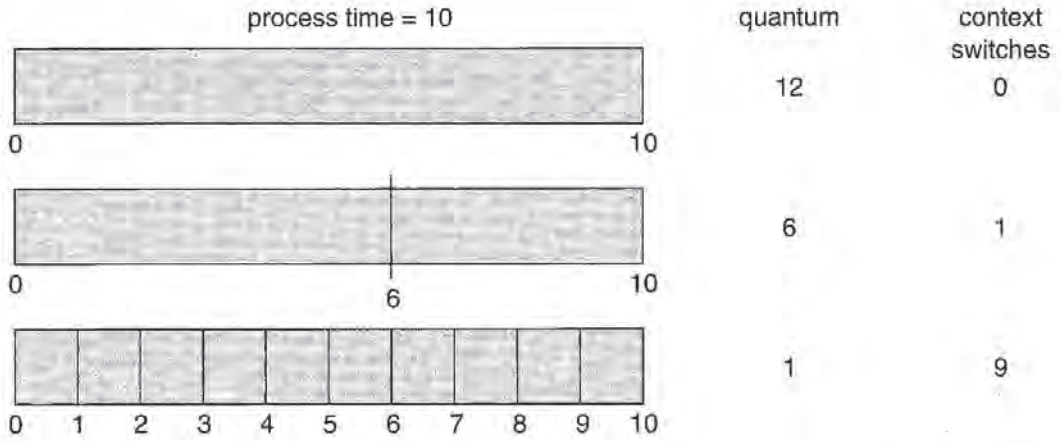


Figure 5.4 Showing how a smaller time quantum increases context switches.

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.

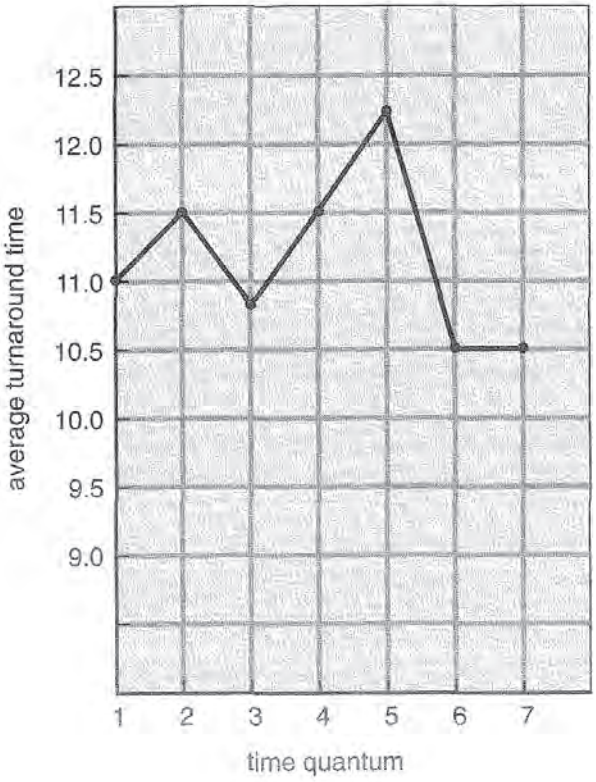
Turnaround time also depends on the size of the time quantum. As we can see from Figure 5.5, the average turnaround time of a set of processes does not necessarily improve as the time-quantum size increases. In general, the average turnaround time can be improved if most processes finish their next CPU 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.

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.

5.3.5 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* (interactive) processes and *background* (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 (externally defined) over background processes.

A *multilevel queue-scheduling algorithm* partitions the ready queue into several separate queues (Figure 5.6). The processes are permanently



process	time
P_1	6
P_2	3
P_3	1
P_4	7

Figure 5.5 Showing how turnaround time varies with the time quantum.

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 between the queues, which is commonly implemented as a fixed-priority preemptive scheduling. For example, the foreground queue may have absolute priority over the background queue.

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

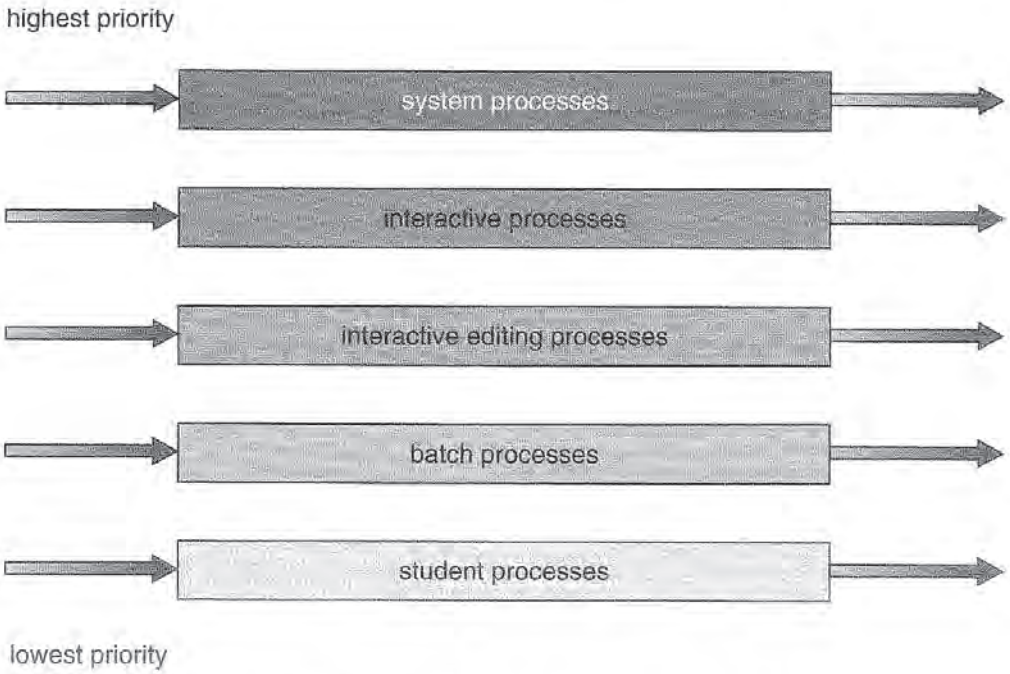


Figure 5.6 Multilevel queue scheduling.

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.

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.

5.3.6 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 is 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 5.7). 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 only be executed if queues 0 and 1 are empty. A process that arrives for queue 1 will preempt a process in queue 2. A process in queue 1 will in turn be preempted by a process arriving for queue 0.

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:

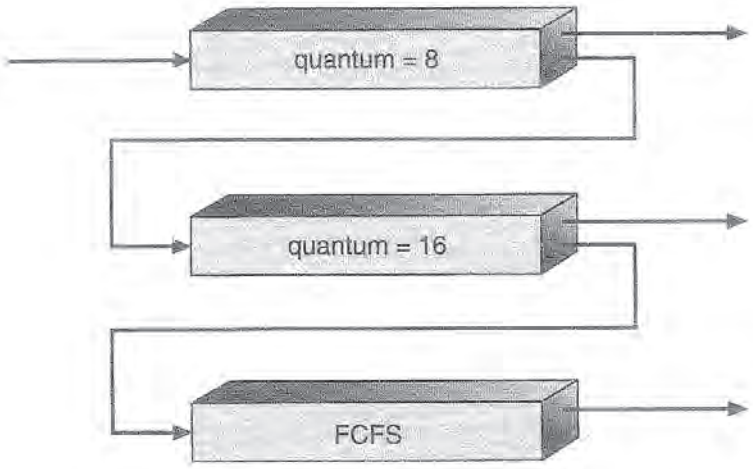


Figure 5.7 Multilevel feedback queues.

- 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.

5.4 ■ 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. A complete coverage is beyond the scope of this text.

The processors within a multiprocessor are identical (*homogeneous*) in terms of their functionality. Any available processor can then be used to run any processes in the queue. If the processors were different (a *heterogeneous* system), only programs compiled for a given processor's instruction set could be run on that processor. This is sometimes the case with distributed systems, as we shall see in Chapters 15 though 18. There are sometimes limitations on scheduling even within homogeneous multiprocessors. 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 6, 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.

5.5 ■ 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 runable.

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 for either 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, which 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 6. 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 method used in Solaris 2.

But what happens if the higher-priority process needs to read or modify kernel data that are 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, there could be a chain of processes, all accessing resources that the high-priority process needs. This problem can be solved via the *priority-inheritance protocol*, in which all these processes (the processes that are

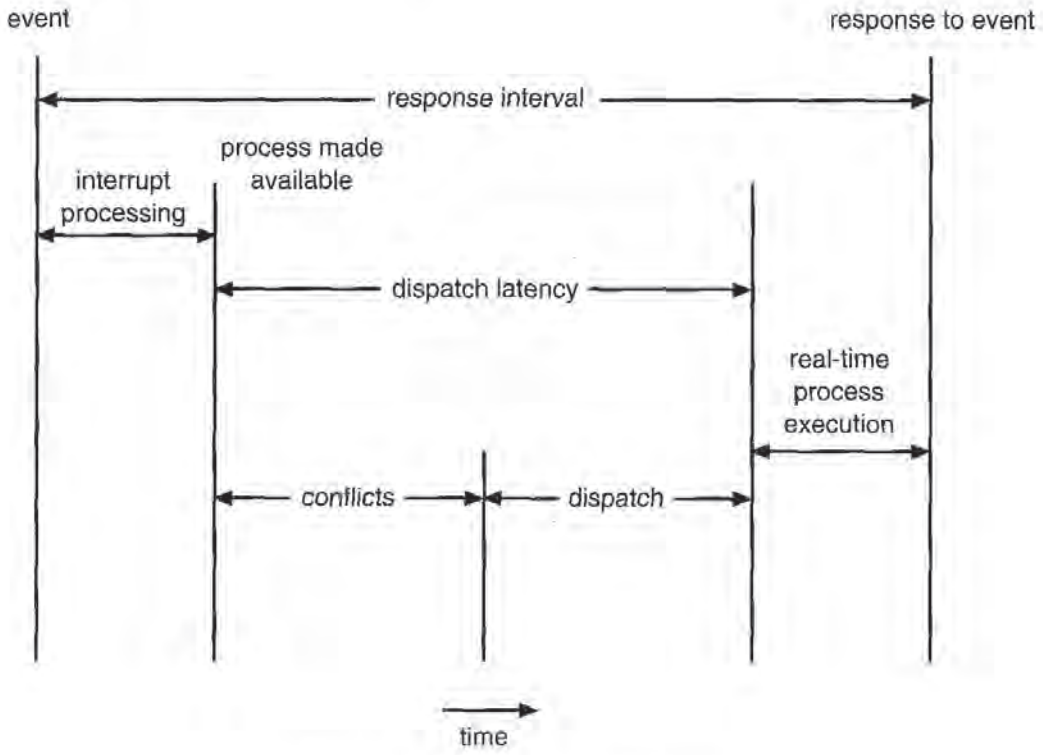


Figure 5.8 Dispatch latency.

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 natural value.

In Figure 5.8, we show the makeup of dispatch latency. The *conflict phase* of dispatch latency has three components:

1. Preemption of any process running in the kernel
2. Low-priority processes releasing resources needed by the high-priority process
3. Context switching from the current process to 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.

5.6 ■ Algorithm Evaluation

How do we select a CPU scheduling algorithm for a particular system? As we saw in Section 5.3, 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 5.2, 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. There are a number of different evaluation methods, which we describe in Sections 5.6.1 through 5.6.4.

5.6.1 Deterministic modeling

One major class of evaluation methods is called *analytic evaluation*. Analytic evaluation uses the algorithm and the system workload to produce a formula or number that evaluates the performance of the algorithm for that workload.

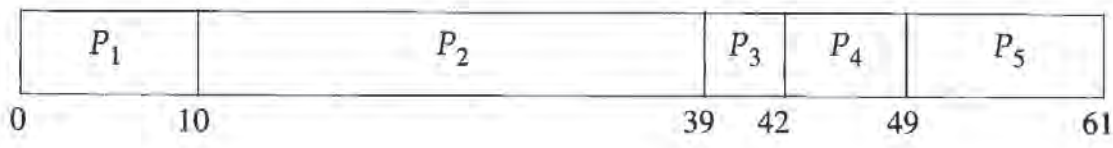
One type of analytic evaluation is *deterministic modeling*. This method takes a particular predetermined workload and defines the performance of each algorithm for that workload.

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

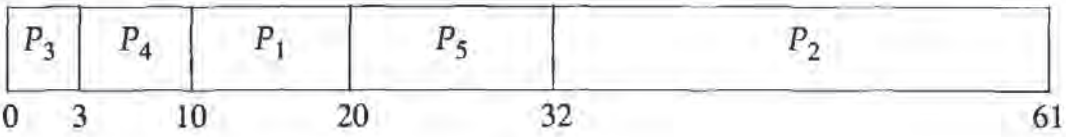
Consider the FCFS, SJF, and RR (quantum = 10 milliseconds) scheduling algorithms for this set of processes. Which algorithm would give the minimum average waiting time?

For the FCFS algorithm, we would execute the processes as



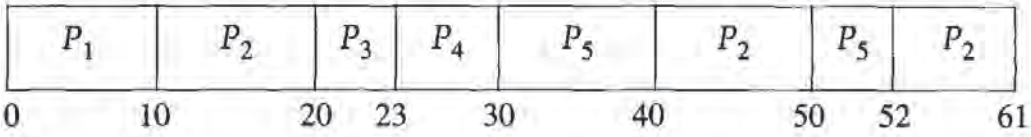
The waiting time is 0 milliseconds for process P_1, 10 milliseconds for process P_2, 39 milliseconds for process P_3, 42 milliseconds for process P_4, and 49 milliseconds for process P_5. Thus, the average waiting time is (0 + 10 + 39 + 42 + 49)/5 = 28 milliseconds.

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 start process P_2, but preempt it after 10 milliseconds, putting it in the back of the queue:



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.

5.6.2 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. From these two distributions, it is possible to compute the average throughput, utilization, waiting time, and so on for most algorithms.

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, which 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.

5.6.3 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.

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 5.9). 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.

5.6.4 Implementation

Even a simulation is of limited accuracy. The only completely accurate way to evaluate a scheduling algorithm is to code it up, to put it in the operating system, and to 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

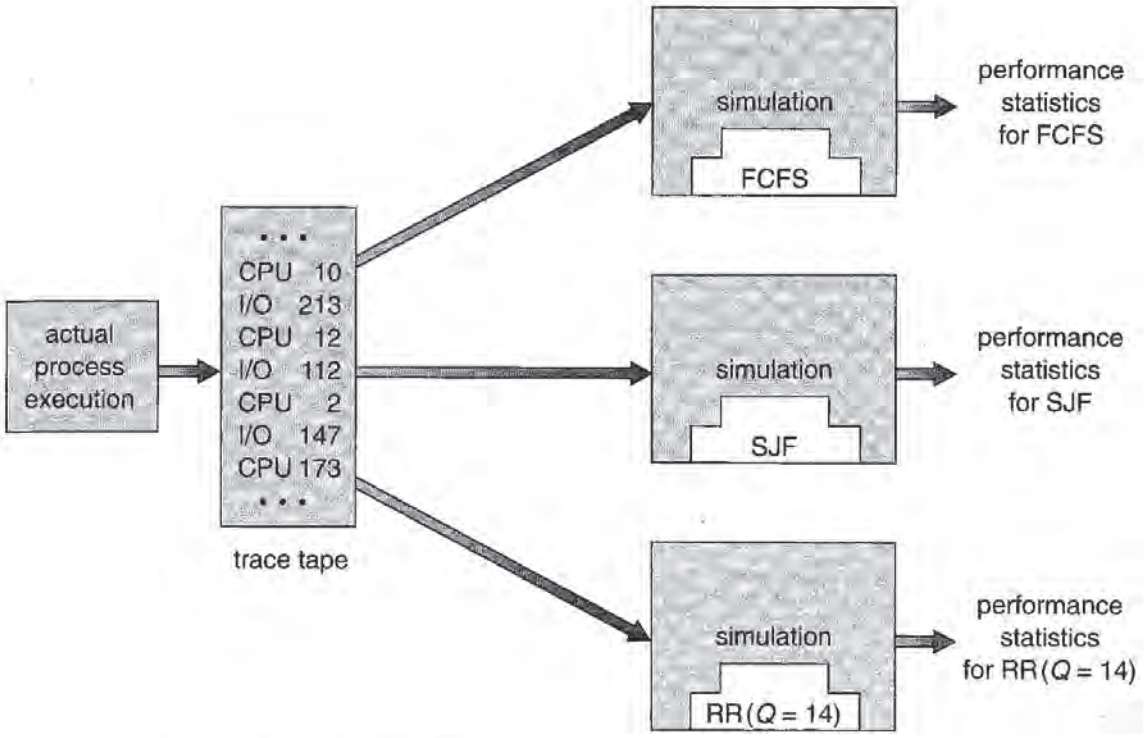


Figure 5.9 Evaluation of CPU schedulers by simulation.

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.

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, researchers tried designing one system to classify 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-second 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 second. 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. 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.

5.7 ■ 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 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, the CPU 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.

The wide variety of scheduling algorithms demands that we have methods to select among algorithms. 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.

■ Exercises

5.1 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 possible different schedules are there? Give a formula in terms of n.

5.2 Define the difference between preemptive and nonpreemptive scheduling. State why strict nonpreemptive scheduling is unlikely to be used in a computer center.

5.3 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)?

5.4 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.

5.5 Consider a variant of the RR scheduling algorithm where the entries in the ready queue are pointers to the PCBs.

a. What would be the effect of putting two pointers to the same process in the ready queue?

b. What would be the major advantages and disadvantages of this scheme?

c. How would you modify the basic RR algorithm to achieve the same effect without the duplicate pointers?

5.6 What advantage is there in having different time-quantum sizes on different levels of a multilevel queueing system?

5.7 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.

a. What is the algorithm that results from $\beta > \alpha > 0$?

b. What is the algorithm that results from $\alpha < \beta < 0$?

5.8 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

5.9 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?

5.10 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

Bibliographic 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 Ruschitzka 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 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]. The

preemptive priority-scheduling algorithm of Exercise 5.9 was suggested by Kleinrock [1975].

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].

Discussions concerning scheduling policies used in the MVS operating system were presented by Samson [1990]; the one for the OS/2 operating system was presented by Iacobucci [1988]; the one for the UNIX V operating system was presented by Bach [1987]; and the one for the Mach operating system was presented by Black [1990].

CHAPTER 6



PROCESS SYNCHRONIZATION

A *cooperating* process is one that can affect or be affected by the 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 4.5. 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.

6.1 ■ 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 have 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 4.6. As we pointed out, our solution allows at most $n - 1$ items in the buffer at the same time. Suppose that we wanted 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:

```
repeat
    ...
    produce an item in nextp
    ...
    while counter = n do no-op;
    buffer[in] := nextp;
    in := in+1 mod n;
    counter := counter + 1;
until false;
```

The code for the consumer process can be modified as follows:

```
repeat
    while counter = 0 do no-op;
    nextc := buffer[out];
    out := out+1 mod n;
    counter := counter - 1;
    ...
    consume the item in nextc
    ...
until false;
```

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* := *counter* + 1" and "*counter* := *counter* − 1" 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* := *counter*+1" may be implemented in machine language (on a typical machine) as

```
register1 := counter;
register1 := register1+1;
counter := register1
```

where $register_1$ is a local CPU register. Similarly, the statement "*counter* := *counter* − 1" is implemented as follows:

```
register2 := counter
register2 := register2 - 1;
counter := register2
```

where again $register_2$ is a local CPU register. Even though $register_1$ and $register_2$ may be the same physical registers (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 the statements "*counter* := *counter* + 1" and "*counter* := *counter* − 1" 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:	*producer*	**execute**	$register_1 := counter$	$\{register_1 = 5\}$
T_1:	*producer*	**execute**	$register_1 := register_1 + 1$	$\{register_1 = 6\}$
T_2:	*consumer*	**execute**	$register_2 := counter$	$\{register_2 = 5\}$
T_3:	*consumer*	**execute**	$register_2 := register_2 - 1$	$\{register_2 = 4\}$
T_4:	*producer*	**execute**	$counter := register_1$	$\{counter = 6\}$
T_5:	*consumer*	**execute**	$counter := register_2$	$\{counter = 4\}$

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.

6.2 ■ The Critical-Section Problem

Consider a system consisting of n processes $\{P_0, P_1, \ldots, P_{n-1}\}$. 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.

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 there exist some processes that wish to enter their critical sections, then only those processes that are not executing in their remainder section can participate in the decision of which will enter its critical section next, and this selection cannot be postponed indefinitely.
3. **Bounded Waiting:** There must exist 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 6.2.1 and 6.2.2, 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,

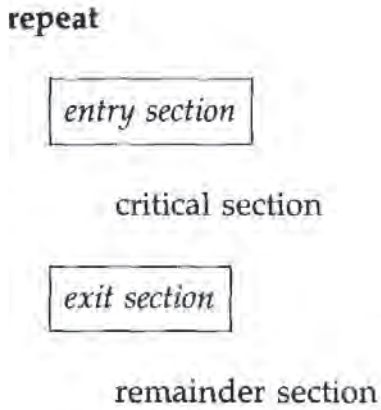


Figure 6.1 General structure of a typical process P_i.

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 6.1. The *entry section* and *exit section* are enclosed in boxes to highlight these important segments of code.

6.2.1 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, $j = 1 - i$.

6.2.1.1 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 6.2.

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.

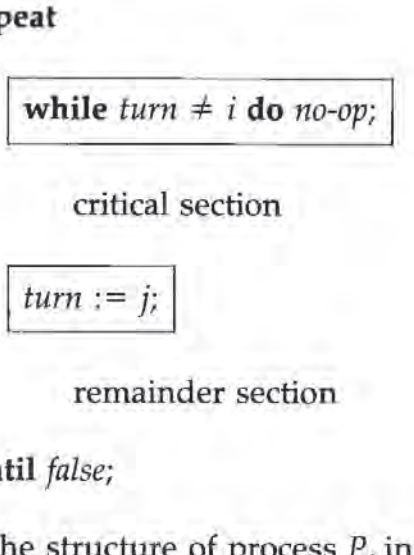


Figure 6.2 The structure of process P_i in algorithm 1.

6.2.1.2 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 that process' critical section. To remedy this problem, we can replace the variable *turn* with the following array:

var *flag*: **array** [0..1] **of** *boolean*;

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 6.3.

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 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 its *flag* 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.

repeat

flag[*i*] := *true*;
while *flag*[*j*] **do** *no-op*;

critical section

flag[*i*] := *false*;

remainder section

until *false*;

Figure 6.3 The structure of process P_i in algorithm 2.

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.

6.2.1.3 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:

```
var flag: array [0..1] of boolean;
    turn: 0..1;
```

Initially *flag*[0] = *flag*[1] = *false*, and the value of *turn* is immaterial (but is either 0 or 1). The structure of process P_i is shown in Figure 6.4.

To enter the critical section, process P_i first sets *flag*[*i*] to be *true*, and then asserts that it is the other process' turn to enter if appropriate (*turn* = *j*). 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

```
repeat

    flag[i] := true;
    turn := j;
    while (flag[j] and turn=j) do no-op;

        critical section

    flag[i] := false;

        remainder section

until false;
```

Figure 6.4 The structure of process P_i in algorithm 3.

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 *flag*[*j*] = *false* or *turn* = *i*. Also note that, if both processes can be executing in their critical sections at the same time, then *flag*[0] = *flag*[1] = *true*. These two observations imply that P_0 and P_1 could not have executed successfully 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 executed successfully the **while** statement, whereas P_i had to execute at least one additional statement ("*turn* = *j*"). However, since, at that time, *flag*[*j*] = *true*, and *turn* = *i*, and this condition will persist as long as P_j is in its critical section, the result follows: Mutual exclusion is preserved.

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 *flag*[*j*] = *true* and *turn* = *j*; this loop is the only one. If P_j is not ready to enter the critical section, then *flag*[*j*] = *false*, and P_i can enter its critical section. If P_j has set *flag*[*j*] = *true* and is also executing in its **while** statement, then either *turn* = *i* or *turn* = *j*. If *turn* = *i*, then P_i will enter the critical section. If *turn* = *j*, then P_j will enter the critical section. However, once P_j exits its critical section, it will reset *flag*[*j*] to *false*, allowing P_i to enter its critical section. If P_j resets *flag*[*j*] to *true*, it must also set *turn* = *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).

6.2.2 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, meat markets, 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

```
var choosing: array [0..n−1] of boolean;
    number: array [0..n−1] of integer;
```

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 $b < d$.
- $max(a_0, \ldots, a_{n-1})$ is a number, k, such that $k \geq a_i$ for $i = 0, \ldots, n-1$.

The structure of process P_i is shown in Figure 6.5.

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 $(k \neq i)$ has already chosen its

```
repeat

    choosing[i] := true;
    number[i] := max(number[0], number[1], ..., number[n−1]) + 1;
    choosing[i] := false;
    for j := 0 to n−1
        do begin
            while choosing[j] do no-op;
            while number[j] ≠ 0
                and (number[j],j) < (number[i],i) do no-op;
        end;

        critical section

    number[i] := 0;

        remainder section

until false;
```

Figure 6.5 The structure of process P_i in the bakery algorithm.

$number[k] \neq 0$, then $(number[i],i) < (number[k],k)$. The proof of this algorithm is left to you in Exercise 6.2.

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

- $number[i] \neq 0$
- $(number[i],i) < (number[k],k)$.

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.

6.3 ■ Synchronization Hardware

As with other aspects of software, features of the hardware 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.

The critical-section problem could be solved simply in a uniprocessor environment if we could disallow 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. 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 *Test-and-Set* instruction can be defined as follows:

```
function Test-and-Set (var target: boolean): boolean;
    begin
        Test-and-Set := target;
        target := true;
    end;
```

The important characteristic is that this instruction is executed atomically — that is, as one uninterruptible unit. Thus, if two *Test-and-Set* instructions are executed simultaneously (each on a different CPU), they will be executed sequentially in some arbitrary order.

If the machine supports the *Test-and-Set* 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 6.6.

The *Swap* instruction swaps the contents of two words, atomically, and is defined as follows:

```
procedure Swap (var a, b: boolean);
    var temp: boolean;
    begin
        temp := a;
        a := b;
        b := temp;
    end;
```

As in the case of the *Test-and-Set* instruction, the *Swap* instruction is also executed atomically.

```
repeat

    while Test-and-Set(lock) do no-op;

        critical section

    lock := false;

        remainder section

until false;
```

Figure 6.6 Mutual-exclusion implementation with *Test-and-Set*.

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 6.7.

These algorithms do not satisfy the bounded-waiting requirement. We present an algorithm that uses the *Test-and-Set* instruction in Figure 6.8. This algorithm satisfies all the critical-section requirements. The common data structures are

```
var waiting: array [0..n−1] of boolean
    lock: boolean
```

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 only if either *waiting*[*i*] = *false* or *key* = *false*. *Key* can become *false* only if the *Test-and-Set* is executed. The first process to execute the *Test-and-Set* 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, 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.

```
repeat

    key := true;
    repeat
        Swap(lock,key);
    until key = false;

        critical section

    lock := false;

        remainder section

until false;
```

Figure 6.7 Mutual-exclusion implementation with the *Swap* instruction.

```
var j: 0..n-1;
   key: boolean;
repeat

    waiting[i] := true;
    key := true;
    while waiting[i] and key do key := Test-and-Set(lock);
    waiting[i] := false;

        critical section

    j := i+1 mod n;
    while (j ≠ i) and (not waiting[j]) do j := j+1 mod n;
    if j = i then lock := false
          else waiting[j] := false;

        remainder section

until false;
```

Figure 6.8 Bounded-waiting mutual exclusion with *Test-and-Set*.

To prove bounded waiting, we note that, when a process leaves its critical section, it scans the array *waiting* in the cyclic ordering ($i + 1, i + 2, \ldots, n - 1, 0, \ldots, i - 1$). 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 test-and-set instructions on multiprocessors is not a trivial task. Such implementations are discussed in books on computer architecture.

6.4 ■ Semaphores

The solutions to the critical-section problem presented in Section 6.3 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 definitions of *wait* and *signal* are

```
wait(S): while S ≤ 0 do no-op;
         S := S − 1;

signal(S): S := S + 1;
```

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 ($S := S - 1$), must also be executed without interruption. We shall see how these operations can be implemented in Section 6.4.2; first, let us see how semaphores can be used.

6.4.1 Usage

We can use semaphores to deal with the n-process critical-section problem. The n processes share a semaphore, *mutex* (standing for *mut*ual *ex*clusion), initialized to 1. Each process P_i is organized as shown in Figure 6.9.

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

```
S1;
signal(synch);
```

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.

6.4.2 Implementation

The main disadvantage of the mutual-exclusion solutions of Section 6.2, 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

```
repeat
    wait(mutex);
        critical section
    signal(mutex);
        remainder section
until false;
```

Figure 6.9 Mutual-exclusion implementation with semaphores.

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, as shown in Chapter 20. 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 record:

```
type semaphore = record
                     value: integer;
                     L: list of process;
                 end;
```

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 semaphore operations can now be defined as

```
wait(S): S.value := S.value - 1;
         if S.value < 0
           then begin
                  add this process to S.L;
                  block;
                end;

signal(S): S.value := S.value + 1;
           if S.value ≤ 0
             then begin
                    remove a process P from S.L;
                    wakeup(P);
                  end;
```

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, which ensures bounded waiting, would be to use a first-in, first-out (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 6.2), 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 applications programs. Furthermore, we have limited it 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 applications programs whose critical sections may be long (hours) or may be almost always occupied. In this case, busy waiting is extremely inefficient.

6.4.3 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 by only 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
wait(S);	*wait*(Q);
wait(Q);	*wait*(S);
.	.
.	.
.	.
signal(S);	*signal*(Q);
signal(Q);	*signal*(S);

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 7. 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.

6.4.4 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:

```
var S1: binary-semaphore;
    S2: binary-semaphore;
    S3: binary-semaphore;
    C: integer;
```

Initially $S1 = S3 = 1$, $S2 = 0$, 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(S3);
wait(S1);
C := C - 1;
if C < 0
then begin
        signal(S1);
        wait(S2);
     end
else signal(S1);
signal(S3);
```

The *signal* operation on the counting semaphore S can be implemented as follows:

```
wait(S1);
C := C + 1;
if C ≤ 0 then signal(S2);
signal(S1);
```

The *S3* semaphore has no effect on *signal*(*S*), it merely serializes the *wait*(*S*) operations.

6.5 ■ Classical Problems of Synchronization

In this section, we present a number of different synchronization problems that are important mainly because they are 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.

6.5.1 The Bounded-Buffer Problem

The bounded-buffer problem was introduced in Section 6.1; 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. (Semaphores with initial values other than 1 are sometimes known as *counting semaphores*.) 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 6.10; the code for the consumer process is shown in Figure 6.11. 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.

6.5.2 The Readers and 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

```
repeat
    ...
    produce an item in nextp
    ...
    wait(empty);
    wait(mutex);
    ...
    add nextp to buffer
    ...
    signal(mutex);
    signal(full);
until false;
```

Figure 6.10 The structure of the producer process.

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 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

```
repeat
    wait(full);
    wait(mutex);
    ...
    remove an item from buffer to nextc
    ...
    signal(mutex);
    signal(empty);
    ...
    consume the item in nextc
    ...
until false;
```

Figure 6.11 The structure of the consumer process.

```
wait(wrt);
    ...
    writing is performed
    ...
signal(wrt);
```

Figure 6.12 The structure of a writer process.

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.

We note that 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 Bibliographic 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:

```
var mutex, wrt: semaphore;
    readcount : integer;
```

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. *Readcount* keeps track of how many processes are currently reading the object. The semaphore *wrt* functions as a mutual exclusion semaphore for the writers. It also is 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 6.12; the code for a reader process is shown in Figure 6.13. 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.

6.5.3 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 there is a

```
wait(mutex);
    readcount := readcount + 1;
    if readcount = 1 then wait(wrt);
signal(mutex);
        ...
     reading is performed
        ...
wait(mutex);
    readcount := readcount - 1;
    if readcount = 0 then signal(wrt);
signal(mutex);
```

Figure 6.13 The structure of a reader process.

bowl of rice, and the table is laid with five single chopsticks (Figure 6.14). 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 releasing 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 for a large class of concurrency-control problems. It is a simple

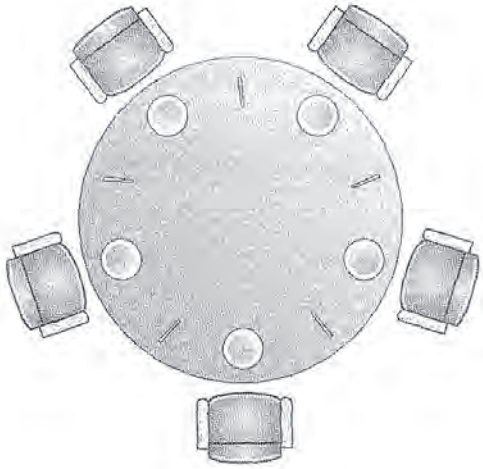


Figure 6.14 The situation of the dining philosophers.

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

var *chopstick*: **array** [0..4] **of** *semaphore*;

where all the elements of *chopstick* are initialized to 1. The structure of philosopher i is shown in Figure 6.15.

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 6.7, 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 (note that 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.

```
repeat
    wait(chopstick[i]);
    wait(chopstick[i+1 mod 5]);
        ...
        eat
        ...
    signal(chopstick[i]);
    signal(chopstick[i+1 mod 5]);
        ...
        think
        ...
until false;
```

Figure 6.15 The structure of philosopher i.

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.

6.6 ■ 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 6.1). 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 even if 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 6.5.

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 6.7, 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 synchronization construct requires that a variable v of type T, which is to be shared among many processes, be declared as

var 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 do S1;
region v when true do 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

```
var buffer: shared record
                pool: array [0..n-1] of item;
                count,in,out: integer;
            end;
```

The producer process inserts a new item *nextp* into the shared buffer by executing

```
region buffer when count < n
   do begin
        pool[in] := nextp;
        in := in+1 mod n;
        count := count + 1;
      end;
```

The consumer process removes an item from the shared buffer and puts it in *nextc* by executing

```
region buffer when count > 0
   do begin
        nextc := pool[out];
        out := out+1 mod n;
        count := count - 1;
      end;
```

Let us illustrate how the conditional critical region could be implemented by a compiler. With each shared variable, the following variables are associated:

```
var mutex, first-delay, second-delay: semaphore;
    first-count, second-count: integer;
```

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.

When a process leaves the critical section, it may have changed the value of some Boolean condition *B* that prevented another process from

```
wait(mutex);
while not B
   do begin
         first-count := first-count + 1;
         if second-count > 0
               then signal(second-delay)
               else signal(mutex);
         wait(first-delay);
         first-count := first-count - 1;
         second-count := second-count + 1;
         if first-count > 0
              then signal(first-delay)
              else signal(second-delay);
         wait(second-delay);
         second-count := second-count - 1;
        end;
S;
if first-count > 0
     then signal(first-delay);
     else if second-count > 0
               then signal(second-delay);
               else  signal(mutex);
```

Figure 6.16 Implementation of the conditional-region construct.

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 **do** S;

can be implemented as shown in Figure 6.16. 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. There are various optimization methods that we can use to reduce this overhead. Refer to the Bibliographic Notes for relevant references.

6.7 ■ 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

```
type monitor-name = monitor
    variable declarations

    procedure entry P1 ( ... );
        begin ... end;

    procedure entry P2 ( ... );
        begin ... end;
            .
            .
            .
    procedure entry Pn ( ... );
        begin ... end;

    begin
        initialization code
    end.
```

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 the 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 6.17). 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* construct. A programmer who needs to write her own tailor-made synchronization scheme can define one or more variables of type *condition:*

var *x,y: condition;*

The only operations that can be invoked on a condition variable are *wait* and *signal*. The operation

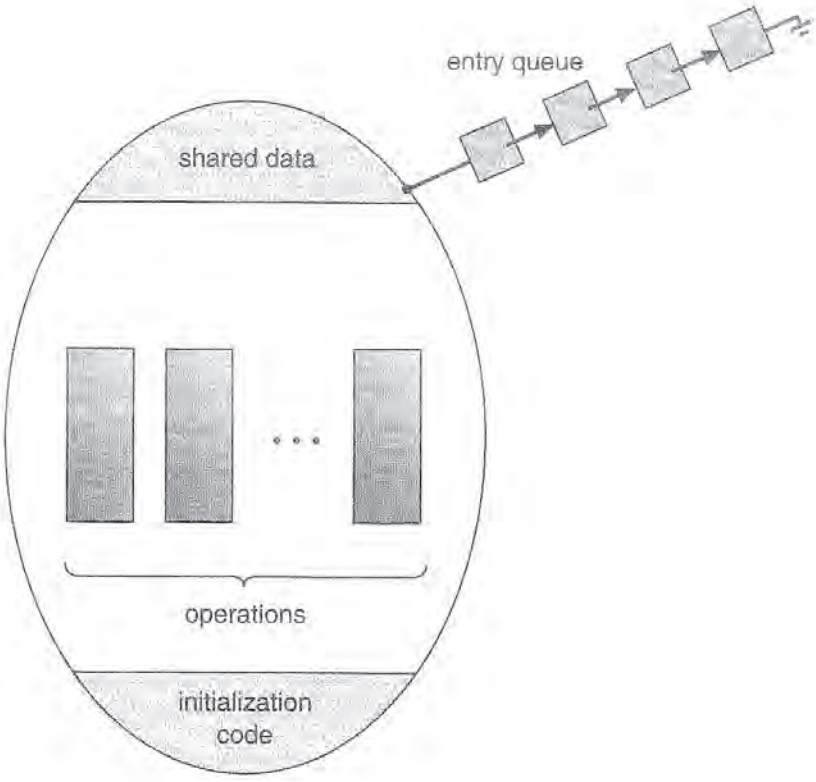


Figure 6.17 Schematic view of a monitor.

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 was never executed (Figure 6.18). 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 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.

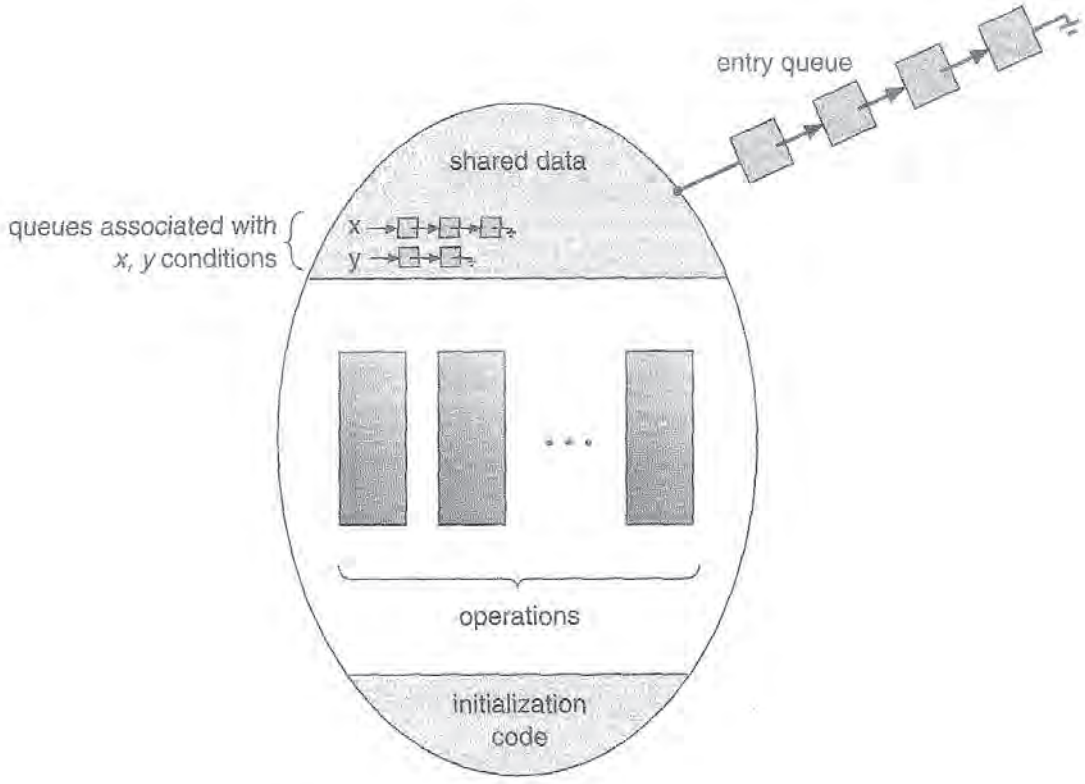


Figure 6.18 Monitor with condition variables.

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 Pascal. When process P executes the *signal* operation, it immediately leaves the monitor. Hence, 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 between three states in which a philosopher may be. For this purpose, we introduce the following data structure:

var *state*: **array** [0..4] **of** (*thinking, hungry, eating*);

Philosopher i can set the variable $state[i] = eating$ only if her two neighbors are not eating ($state[i+4 \bmod 5] \neq eating$ **and** $state[i+1 \bmod 5] \neq eating$).

We also need to declare

var *self*: **array** [0..4] **of** *condition*;

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. The distribution of the chopsticks is controlled by the monitor shown in Figure 6.19. Philosopher i must invoke the operations *pickup* and *putdown* on an instance *dp* of the *dining-philosophers* monitor in the following sequence:

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.

```
type dining-philosophers = monitor
    var state : array [0..4] of (thinking, hungry, eating);
    var self : array [0..4] of condition;

    procedure entry pickup (i: 0..4);
        begin
            state[i] := hungry;
            test (i);
            if state[i] ≠ eating then self[i].wait;
        end;

    procedure entry putdown (i: 0..4);
        begin
            state[i] := thinking;
            test (i+4 mod 5);
            test (i+1 mod 5);
        end;

    procedure test (k: 0..4);
        begin
            if state[k+4 mod 5] ≠ eating
              and state[k] = hungry
              and state[k+1 mod 5] ≠ eating
              then begin
                        state[k] := eating;
                        self[k].signal;
                   end;
        end;

    begin
        for i := 0 to 4
            do state[i] := thinking;
    end.
```

Figure 6.19 A monitor solution to the dining-philosopher problem.

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
   then 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 := x-count + 1;
if next-count > 0
   then signal(next)
   else signal(mutex);
wait(x-sem);
x-count := x-count - 1;
```

The operation *x.signal* can be implemented as

```
if x-count > 0
   then begin
           next-count := next-count + 1;
           signal(x-sem);
           wait(next);
           next-count := next-count - 1;
        end;
```

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 6.12.

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 an FCFS ordering, so that the process waiting the longest

is resumed first. There are, however, many circumstances in which 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 6.20, 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:

```
R.acquire(t);
...
    access the resource;
...
R.release;
```

where R is an instance of type *resource-allocation*.

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).

Note that 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

```
type resource-allocation = monitor
    var busy: boolean;
        x: condition;

    procedure entry acquire (time: integer);
        begin
            if busy then x.wait(time);
            busy := true;
        end;

    procedure entry release;
        begin
            busy := false;
            x.signal;
        end;

    begin
        busy := false;
    end.
```

Figure 6.20 A monitor to allocate a single resource.

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 *resource-allocation* 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 *resource-allocation* 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 13.

6.8 ■ Synchronization in Solaris 2

To solidify this discussion, we now return to Solaris 2. Before the advent of Solaris 2, SunOS used critical sections to guard important data structures. The system implemented the critical sections by setting the interrupt level to as high as or higher than any interrupt that could modify the same data. Thus, no interrupt would occur that would allow a change to the same data.

In Section 5.5, we described the changes needed to support real-time computing on a time-sharing system. Solaris 2 was designed to provide real-time capabilities, be multithreaded, and support multiprocessors. Continuing to use critical sections would have caused a large performance degradation, as the kernel bottlenecked waiting for entry into critical sections. Further, critical sections could not have been implemented via interrupt elevation because interrupts could occur on other processors on a multiprocessor system. To avoid these problems, Solaris 2 uses *adaptive mutexes* to protect 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, the thread waits 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.

For more complex synchronization situations, Solaris 2 uses condition variables and readers–writers locks. The adaptive mutex method described above is used 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 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 are

semaphores, because multiple threads may be reading data concurrently, whereas semaphores would always serialize access to the data. Readers–writers locks are expensive to implement, so again they are used on only long sections of code.

6.9 ■ 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, there are many cases where 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 6.9.2 to 6.9.4, we describe what these database techniques are, and how they can be used by operating systems.

6.9.1 System Model

A collection of instructions (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 Section 6.9, 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 disk 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 (see Section 12.6).

Here, we are concerned only with ensuring transaction atomicity in an environment where failures result in the loss of information on volatile storage.

6.9.2 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
- **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 $<T_i$ **starts**> 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 $<T_i$ **commits**> 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 record <T_i **starts**>, but does not contain the record <T_i **commits**>.
- Transaction T_i needs to be redone if the log contains both the record <T_i **starts**> and the record <T_i **commits**>.

6.9.3 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, which 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 <T_i **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 <T_i **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 <T_k **commits**> appears in the log, execute **redo**(T_k).
- For all transactions T_k in T that have no <T_k **commits**> record in the log, execute **undo**(T_k).

6.9.4 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 nevertheless is too restrictive. As we shall see, there are many cases where we can allow transactions to overlap their execution, while maintaining serializability. There are a number of different *concurrency-control* algorithms to ensure serializability. These are described below.

6.9.4.1 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

T_0	T_1
read(A)	
write(A)	
read(B)	
write(B)	
	read(A)
	write(A)
	read(B)
	write(B)

Figure 6.21 Schedule 1: A serial schedule in which T_0 is followed by T_1.

sequence, which is called a *schedule*, is represented in Figure 6.21. In schedule 1 of Figure 6.21, 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 O_j *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 6.22. 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 6.22. As the **write**(A) operation of T_1 does not conflict with the

T_0	T_1
read(A)	
write(A)	
	read(A)
	write(A)
read(B)	
write(B)	
	read(B)
	write(B)

Figure 6.22 Schedule 2: A concurrent serializable schedule.

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 6.21, 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.

6.9.4.2 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 it 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 6.5.2.

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 (see Exercise 6.21). 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.

6.9.4.3 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 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 $TS(T_i) <$ W-timestamp(Q), then this state implies that T_i needs to read a value of Q which was already overwritten. Hence, the **read** operation is rejected, and T_i is rolled back.
  - If $TS(T_i) \geq$ W-timestamp(Q), then the **read** operation is executed, and R-timestamp(Q) is set to the maximum of R-timestamp(Q) and $TS(T_i)$.

- Suppose that transaction T_i issues **write**(Q):
  - If TS(T_i) < 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 TS(T_i) < 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, which is rolled back by the concurrency-control scheme as 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 6.23 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, TS(T_2) < TS(T_3), and the schedule is possible under the timestamp protocol.

We note that this execution can also be produced by the two-phase locking protocol. There are, however, schedules that are possible under the two-phase locking protocol but are not possible under the timestamp protocol, and vice versa (see Exercise 6.22).

The timestamp-ordering protocol ensures conflict serializability. This capability follows from the fact that conflicting operations are processed in timestamp order. The protocol ensures freedom from deadlock, because no transaction ever waits.

6.10 ■ 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

T_2	T_3
read(B)	
	read(B)
	write(B)
read(A)	
	read(A)
	write(A)

Figure 6.23 Schedule 3: A schedule possible under the timestamp protocol.

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 which 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.

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

6.1 What is the meaning of the term *busy waiting*? What other kinds of waiting are there? Can busy waiting be avoided altogether? Explain your answer.

6.2 Prove that, in the bakery algorithm (Section 6.2.2), 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 $(number[i],i) < (number[k],k)$.

6.3 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:

```
var flag: array [0..1] of boolean;  (* initially false *)
    turn: 0..1;
```

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 6.24.

Prove that the algorithm satisfies all three requirements for the critical-section problem.

```
repeat

    flag[i] := true;
    while flag[j]
        do if turn = j
            then begin
                    flag[i] := false;
                    while turn = j do no-op;
                    flag[i] := true;
                 end;

            critical section

    turn := j;
    flag[i] := false;

            remainder section

until false;
```

Figure 6.24 The structure of process P_i in Dekker's algorithm.

```
var j: 0..n;
repeat
```

```
repeat
    flag[i] := want-in;
    j := turn;
    while j ≠ i
        do if flag[j] ≠ idle
                then j := turn
                else j := j+1 mod n;
    flag[i] := in-cs;
    j := 0;
    while (j < n) and (j = i or flag[j] ≠ in-cs) do j := j+1;
until (j ≥ n) and (turn = i or flag[turn] = idle);
turn := i;
```

critical section

```
j := turn+1 mod n;
while (flag[j] = idle) do j := j+1 mod n;
turn := j;
flag[i] := idle;
```

remainder section

```
until false;
```

Figure 6.25 The structure of P_i in Eisenberg and McGuire's algorithm.

6.4 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:

```
var flag: array [0..n−1] of (idle, wa0-in, in-cs);
    turn: 0..n−1;
```

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 6.25.

Prove that the algorithm satisfies all three requirements for the critical-section problem.

6.5 In Section 6.3, we mentioned that disabling interrupts frequently could affect the system's clock. Explain why it could, and how such effects could be minimized.

6.6 Show that, if the *wait* and *signal* operations are not executed atomically, then mutual exclusion may be violated.

6.7 *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.

6.8 *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 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.

6.9 Demonstrate that monitors, conditional critical regions, and semaphores are all equivalent, insofar as the same types of synchronization problems can be implemented with them.

6.10 Write a bounded-buffer monitor in which the buffers (portions) are embedded within the monitor itself.

6.11 The strict mutual exclusion within a monitor makes the bounded-buffer monitor of Exercise 6.10 mainly suitable for small portions.

a. Explain why this assertion is true.

b. Design a new scheme that is suitable for larger portions.

6.12 Suppose that the *signal* statement can appear as only the last statement in a monitor procedure. Suggest how the implementation described in Section 6.7 can be simplified.

6.13 Consider a system consisting of processes P_1, P_2, ..., 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.

6.14 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.

6.15 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.

a. Write a monitor using this scheme to implement the readers–writers problem.

b. Explain why, in general, this construct cannot be implemented efficiently.

c. 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].)

6.16 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, which invokes a procedure *tick* in your monitor at regular intervals.

6.17 Why does Solaris 2 implement multiple locking mechanisms? Under what circumstances does it use spinlocks, blocking semaphores, conditional variables, and readers–writers locks? Why does it use each mechanism?

6.18 Explain the differences, in terms of cost, among the three storage types: volatile, nonvolatile, and stable.

6.19 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?

6.20 Explain the concept of transaction atomicity.

6.21 Show that the two-phase locking protocol ensures conflict serializability.

6.22 Show that there are schedules that are possible under the two-phase locking protocol but are not possible under the timestamp protocol, and vice versa.

Bibliographic Notes

The mutual-exclusion algorithms 1 to 2 for two processes were first discussed in the classical 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 that process 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, 1991]. A collection of algorithms for mutual exclusion were 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 limitation 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, 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 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 Khanna et al. [1992], Powell et al. [1991], and especially Eykholt et al. [1992]. 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.

The write-ahead log scheme was first introduced in System R [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].