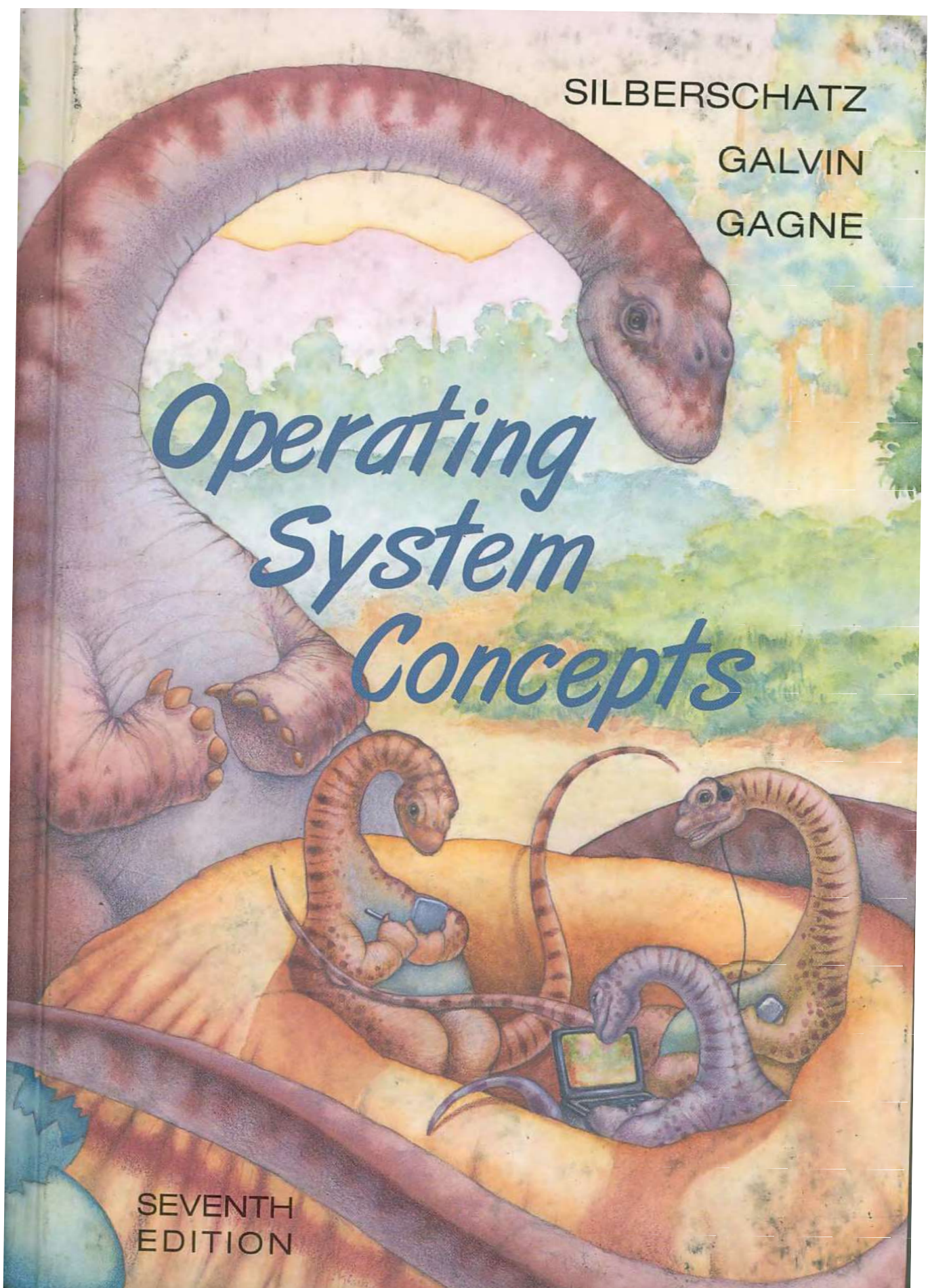




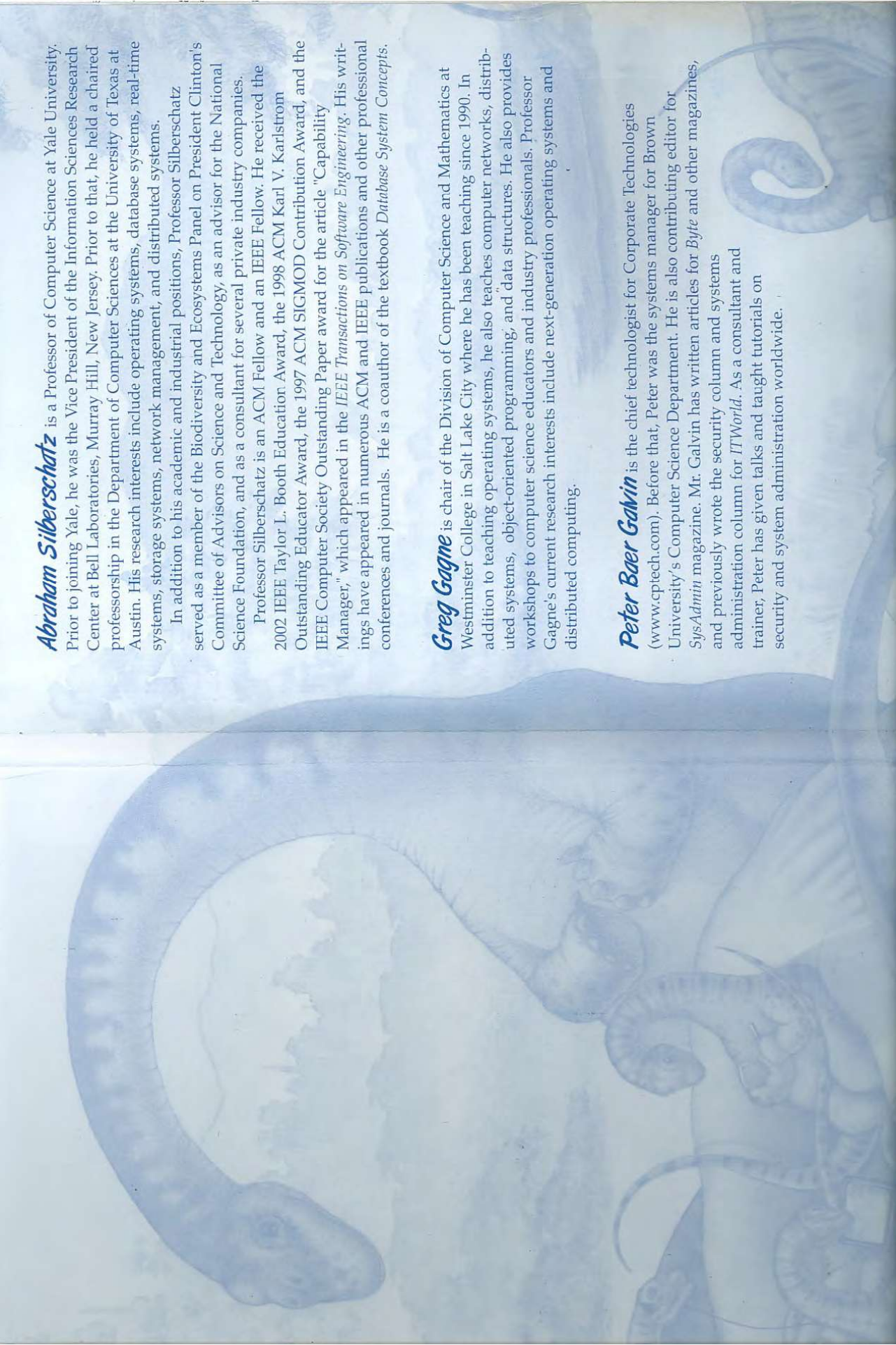
SILBERSCHATZ

GALVIN

GAGNE

# *Operating System Concepts*

SEVENTH  
EDITION



### ***Abraham Silberschatz***

is a Professor of Computer Science at Yale University. Prior to joining Yale, he was the Vice President of the Information Sciences Research Center at Bell Laboratories, Murray Hill, New Jersey. Prior to that, he held a chaired professorship in the Department of Computer Sciences at the University of Texas at Austin. His research interests include operating systems, database systems, real-time systems, storage systems, network management, and distributed systems.

In addition to his academic and industrial positions, Professor Silberschatz served as a member of the Biodiversity and Ecosystems Panel on President Clinton's Committee of Advisors on Science and Technology, as an advisor for the National Science Foundation, and as a consultant for several private industry companies.

Professor Silberschatz is an ACM Fellow and an IEEE Fellow. He received the 2002 IEEE Taylor L. Booth Education Award, the 1998 ACM Karl V. Karlstrom Outstanding Educator Award, the 1997 ACM SIGMOD Contribution Award, and the IEEE Computer Society Outstanding Paper award for the article "Capability Manager," which appeared in the *IEEE Transactions on Software Engineering*. His writings have appeared in numerous ACM and IEEE publications and other professional conferences and journals. He is a coauthor of the textbook *Database System Concepts*.

### ***Greg Gagne***

is chair of the Division of Computer Science and Mathematics at Westminster College in Salt Lake City where he has been teaching since 1990. In addition to teaching operating systems, he also teaches computer networks, distributed systems, object-oriented programming, and data structures. He also provides workshops to computer science educators and industry professionals. Professor Gagne's current research interests include next-generation operating systems and distributed computing.

### ***Peter Baer Galvin***

is the chief technologist for Corporate Technologies ([www.cptech.com](http://www.cptech.com)). Before that, Peter was the systems manager for Brown University's Computer Science Department. He is also contributing editor for *SysAdmin* magazine. Mr. Galvin has written articles for *Byte* and other magazines, and previously wrote the security column and systems administration column for *ITWorld*. As a consultant and trainer, Peter has given talks and taught tutorials on security and system administration worldwide.

# *Operating System Concepts*

Seventh Edition

**ABRAHAM SILBERSCHATZ**  
Yale University

**PETER BAER GALVIN**  
Corporate Technologies, Inc.

**GREG GAGNE**  
Westminster College



JOHN WILEY & SONS, INC

|                          |                |
|--------------------------|----------------|
| EXECUTIVE EDITOR         | Bill Zobrist   |
| SENIOR PRODUCTION EDITOR | Ken Santor     |
| COVER DESIGNER           | Madelyn Lesure |
| COVER ILLUSTRATION       | Susan St. Cyr  |
| TEXT DESIGNER            | Judy Allan     |

This book was set in Palatino by the author using LaTeX and printed and bound by Von Hoffmann, Inc. The cover was printed by Von Hoffmann, Inc.

This book is printed on acid free paper. ∞

Copyright © 2005 John Wiley & Sons, Inc. All rights reserved.

No part of this publication may be reproduced, stored in a retrieval system or transmitted in any form or by any means, electronic, mechanical, photocopying, recording, scanning or otherwise, except as permitted under Sections 107 or 108 of the 1976 United States Copyright Act, without either the prior written permission of the Publisher, or authorization through payment of the appropriate per-copy fee to the Copyright Clearance Center, 222 Rosewood Drive, Danvers, MA 01923, (978) 750-8400, fax (978) 646-8600. Requests to the Publisher for permission should be addressed to the Permissions Department, John Wiley & Sons, Inc., 111 River Street, Hoboken, NJ 07030, (201) 748-6011, fax (201) 748-6008.

To order books or for customer service please, call 1(800)-CALL-WILEY (225-5945).

ISBN 0-471-69466-5

Printed in the United States of America

10 9 8 7 6 5 4 3 2 1

---

# *Contents*

## **PART ONE ■ OVERVIEW**

### **Chapter 1 Introduction**

- |                                  |    |                              |    |
|----------------------------------|----|------------------------------|----|
| 1.1 What Operating Systems Do    | 3  | 1.9 Protection and Security  | 26 |
| 1.2 Computer-System Organization | 6  | 1.10 Distributed Systems     | 28 |
| 1.3 Computer-System Architecture | 12 | 1.11 Special-Purpose Systems | 29 |
| 1.4 Operating-System Structure   | 15 | 1.12 Computing Environments  | 31 |
| 1.5 Operating-System Operations  | 17 | 1.13 Summary                 | 34 |
| 1.6 Process Management           | 20 | Exercises                    | 36 |
| 1.7 Memory Management            | 21 | Bibliographical Notes        | 38 |
| 1.8 Storage Management           | 22 |                              |    |

### **Chapter 2 Operating-System Structures**

- |                                                |    |                                 |    |
|------------------------------------------------|----|---------------------------------|----|
| 2.1 Operating-System Services                  | 39 | 2.7 Operating-System Structure  | 58 |
| 2.2 User Operating-System Interface            | 41 | 2.8 Virtual Machines            | 64 |
| 2.3 System Calls                               | 43 | 2.9 Operating-System Generation | 70 |
| 2.4 Types of System Calls                      | 47 | 2.10 System Boot                | 71 |
| 2.5 System Programs                            | 55 | 2.11 Summary                    | 72 |
| 2.6 Operating-System Design and Implementation | 56 | Exercises                       | 73 |
|                                                |    | Bibliographical Notes           | 78 |

## **PART TWO ■ PROCESS MANAGEMENT**

### **Chapter 3 Processes**

- |                                |     |                                            |     |
|--------------------------------|-----|--------------------------------------------|-----|
| 3.1 Process Concept            | 81  | 3.6 Communication in Client-Server Systems | 108 |
| 3.2 Process Scheduling         | 85  | 3.7 Summary                                | 115 |
| 3.3 Operations on Processes    | 90  | Exercises                                  | 116 |
| 3.4 Interprocess Communication | 96  | Bibliographical Notes                      | 125 |
| 3.5 Examples of IPC Systems    | 102 |                                            |     |

**Chapter 4 Threads**

- 4.1 Overview 127
- 4.2 Multithreading Models 129
- 4.3 Thread Libraries 131
- 4.4 Threading Issues 138
- 4.5 Operating-System Examples 143
- 4.6 Summary 146
  - Exercises 146
  - Bibliographical Notes 151

**Chapter 5 CPU Scheduling**

- 5.1 Basic Concepts 153
- 5.2 Scheduling Criteria 157
- 5.3 Scheduling Algorithms 158
- 5.4 Multiple-Processor Scheduling 169
- 5.5 Thread Scheduling 172
- 5.6 Operating System Examples 173
- 5.7 Algorithm Evaluation 181
- 5.8 Summary 185
  - Exercises 186
  - Bibliographical Notes 189

**Chapter 6 Process Synchronization**

- 6.1 Background 191
- 6.2 The Critical-Section Problem 193
- 6.3 Peterson's Solution 195
- 6.4 Synchronization Hardware 197
- 6.5 Semaphores 200
- 6.6 Classic Problems of Synchronization 204
- 6.7 Monitors 209
- 6.8 Synchronization Examples 217
- 6.9 Atomic Transactions 222
- 6.10 Summary 230
  - Exercises 231
  - Bibliographical Notes 242

**Chapter 7 Deadlocks**

- 7.1 System Model 245
- 7.2 Deadlock Characterization 247
- 7.3 Methods for Handling Deadlocks 252
- 7.4 Deadlock Prevention 253
- 7.5 Deadlock Avoidance 256
- 7.6 Deadlock Detection 262
- 7.7 Recovery From Deadlock 266
- 7.8 Summary 267
  - Exercises 268
  - Bibliographical Notes 271

**PART THREE ■ MEMORY MANAGEMENT****Chapter 8 Main Memory**

- 8.1 Background 275
- 8.2 Swapping 282
- 8.3 Contiguous Memory Allocation 284
- 8.4 Paging 288
- 8.5 Structure of the Page Table 297
- 8.6 Segmentation 302
- 8.7 Example: The Intel Pentium 305
- 8.8 Summary 309
  - Exercises 310
  - Bibliographical Notes 312

## Chapter 9 Virtual Memory

- 9.1 Background 315
- 9.2 Demand Paging 319
- 9.3 Copy-on-Write 325
- 9.4 Page Replacement 327
- 9.5 Allocation of Frames 340
- 9.6 Thrashing 343
- 9.7 Memory-Mapped Files 348
- 9.8 Allocating Kernel Memory 353
- 9.9 Other Considerations 357
- 9.10 Operating-System Examples 363
- 9.11 Summary 365
  - Exercises 366
  - Bibliographical Notes 370

## PART FOUR ■ STORAGE MANAGEMENT

### Chapter 10 File-System Interface

- 10.1 File Concept 373
- 10.2 Access Methods 382
- 10.3 Directory Structure 385
- 10.4 File-System Mounting 395
- 10.5 File Sharing 397
- 10.6 Protection 402
- 10.7 Summary 407
  - Exercises 408
  - Bibliographical Notes 409

### Chapter 11 File-System Implementation

- 11.1 File-System Structure 411
- 11.2 File-System Implementation 413
- 11.3 Directory Implementation 419
- 11.4 Allocation Methods 421
- 11.5 Free-Space Management 429
- 11.6 Efficiency and Performance 431
- 11.7 Recovery 435
- 11.8 Log-Structured File Systems 437
- 11.9 NFS 438
- 11.10 Example: The WAFL File System 444
- 11.11 Summary 446
  - Exercises 447
  - Bibliographical Notes 449

### Chapter 12 Mass-Storage Structure

- 12.1 Overview of Mass-Storage Structure 451
- 12.2 Disk Structure 454
- 12.3 Disk Attachment 455
- 12.4 Disk Scheduling 456
- 12.5 Disk Management 462
- 12.6 Swap-Space Management 466
- 12.7 RAID Structure 468
- 12.8 Stable-Storage Implementation 477
- 12.9 Tertiary-Storage Structure 478
- 12.10 Summary 488
  - Exercises 489
  - Bibliographical Notes 493

### Chapter 13 I/O Systems

- 13.1 Overview 495
- 13.2 I/O Hardware 496
- 13.3 Application I/O Interface 505
- 13.4 Kernel I/O Subsystem 511
- 13.5 Transforming I/O Requests to Hardware Operations 518
- 13.6 STREAMS 520
- 13.7 Performance 522
- 13.8 Summary 525
  - Exercises 526
  - Bibliographical Notes 527

## PART FIVE ■ PROTECTION AND SECURITY

### Chapter 14 Protection

- |                                      |     |                                  |     |
|--------------------------------------|-----|----------------------------------|-----|
| 14.1 Goals of Protection             | 531 | 14.7 Revocation of Access Rights | 546 |
| 14.2 Principles of Protection        | 532 | 14.8 Capability-Based Systems    | 547 |
| 14.3 Domain of Protection            | 533 | 14.9 Language-Based Protection   | 550 |
| 14.4 Access Matrix                   | 538 | 14.10 Summary                    | 555 |
| 14.5 Implementation of Access Matrix | 542 | Exercises                        | 556 |
| 14.6 Access Control                  | 545 | Bibliographical Notes            | 557 |

### Chapter 15 Security

- |                                         |     |                             |     |
|-----------------------------------------|-----|-----------------------------|-----|
| 15.1 The Security Problem               | 559 | 15.8 Computer-Security      |     |
| 15.2 Program Threats                    | 563 | Classifications             | 600 |
| 15.3 System and Network Threats         | 571 | 15.9 An Example: Windows XP | 602 |
| 15.4 Cryptography as a Security Tool    | 576 | 15.10 Summary               | 604 |
| 15.5 User Authentication                | 587 | Exercises                   | 604 |
| 15.6 Implementing Security Defenses     | 592 | Bibliographical Notes       | 606 |
| 15.7 Firewalling to Protect Systems and |     |                             |     |
| Networks                                | 599 |                             |     |

## PART SIX ■ DISTRIBUTED SYSTEMS

### Chapter 16 Distributed System Structures

- |                                     |     |                             |     |
|-------------------------------------|-----|-----------------------------|-----|
| 16.1 Motivation                     | 611 | 16.7 Robustness             | 631 |
| 16.2 Types of Distributed Operating |     | 16.8 Design Issues          | 633 |
| Systems                             | 613 | 16.9 An Example: Networking | 636 |
| 16.3 Network Structure              | 617 | 16.10 Summary               | 637 |
| 16.4 Network Topology               | 620 | Exercises                   | 638 |
| 16.5 Communication Structure        | 622 | Bibliographical Notes       | 640 |
| 16.6 Communication Protocols        | 628 |                             |     |

### Chapter 17 Distributed File Systems

- |                                        |     |                       |     |
|----------------------------------------|-----|-----------------------|-----|
| 17.1 Background                        | 641 | 17.6 An Example: AFS  | 654 |
| 17.2 Naming and Transparency           | 643 | 17.7 Summary          | 659 |
| 17.3 Remote File Access                | 646 | Exercises             | 660 |
| 17.4 Stateful Versus Stateless Service | 651 | Bibliographical Notes | 661 |
| 17.5 File Replication                  | 652 |                       |     |

**Chapter 18 Distributed Coordination**

- |                              |                              |
|------------------------------|------------------------------|
| 18.1 Event Ordering 663      | 18.6 Election Algorithms 683 |
| 18.2 Mutual Exclusion 666    | 18.7 Reaching Agreement 686  |
| 18.3 Atomicity 669           | 18.8 Summary 688             |
| 18.4 Concurrency Control 672 | Exercises 689                |
| 18.5 Deadlock Handling 676   | Bibliographical Notes 690    |

**PART SEVEN ■ SPECIAL-PURPOSE SYSTEMS****Chapter 19 Real-Time Systems**

- |                                                   |                                   |
|---------------------------------------------------|-----------------------------------|
| 19.1 Overview 695                                 | 19.5 Real-Time CPU Scheduling 704 |
| 19.2 System Characteristics 696                   | 19.6 VxWorks 5.x 710              |
| 19.3 Features of Real-Time Kernels 698            | 19.7 Summary 712                  |
| 19.4 Implementing Real-Time Operating Systems 700 | Exercises 713                     |
|                                                   | Bibliographical Notes 713         |

**Chapter 20 Multimedia Systems**

- |                                             |                                |
|---------------------------------------------|--------------------------------|
| 20.1 What Is Multimedia? 715                | 20.6 Network Management 725    |
| 20.2 Compression 718                        | 20.7 An Example: CineBlitz 728 |
| 20.3 Requirements of Multimedia Kernels 720 | 20.8 Summary 730               |
| 20.4 CPU Scheduling 722                     | Exercises 731                  |
| 20.5 Disk Scheduling 723                    | Bibliographical Notes 733      |

**PART EIGHT ■ CASE STUDIES****Chapter 21 The Linux System**

- |                             |                                     |
|-----------------------------|-------------------------------------|
| 21.1 Linux History 737      | 21.8 Input and Output 770           |
| 21.2 Design Principles 742  | 21.9 Interprocess Communication 773 |
| 21.3 Kernel Modules 745     | 21.10 Network Structure 774         |
| 21.4 Process Management 748 | 21.11 Security 777                  |
| 21.5 Scheduling 751         | 21.12 Summary 779                   |
| 21.6 Memory Management 756  | Exercises 780                       |
| 21.7 File Systems 764       | Bibliographical Notes 781           |

**Chapter 22 Windows XP**

- |                                   |                               |
|-----------------------------------|-------------------------------|
| 22.1 History 783                  | 22.6 Networking 822           |
| 22.2 Design Principles 785        | 22.7 Programmer Interface 829 |
| 22.3 System Components 787        | 22.8 Summary 836              |
| 22.4 Environmental Subsystems 811 | Exercises 836                 |
| 22.5 File System 814              | Bibliographical Notes 837     |

## Chapter 23 Influential Operating Systems

- |                        |                         |
|------------------------|-------------------------|
| 23.1 Early Systems 839 | 23.7 MULTICS 849        |
| 23.2 Atlas 845         | 23.8 IBM OS/360 850     |
| 23.3 XDS-940 846       | 23.9 Mach 851           |
| 23.4 THE 847           | 23.10 Other Systems 853 |
| 23.5 RC 4000 848       | Exercises 853           |
| 23.6 CTSS 849          |                         |

## Appendix A UNIX BSD (contents online)

- |                               |                                     |
|-------------------------------|-------------------------------------|
| A.1 UNIX History A855         | A.7 File System A878                |
| A.2 Design Principles A860    | A.8 I/O System A886                 |
| A.3 Programmer Interface A862 | A.9 Interprocess Communication A889 |
| A.4 User Interface A869       | A.10 Summary A894                   |
| A.5 Process Management A872   | Exercises A895                      |
| A.6 Memory Management A876    | Bibliographical Notes A896          |

## Appendix B The Mach System (contents online)

- |                                     |                               |
|-------------------------------------|-------------------------------|
| B.1 History of the Mach System A897 | B.7 Programmer Interface A919 |
| B.2 Design Principles A899          | B.8 Summary A920              |
| B.3 System Components A900          | Exercises A921                |
| B.4 Process Management A903         | Bibliographical Notes A922    |
| B.5 Interprocess Communication A909 | Credits A923                  |
| B.6 Memory Management A914          |                               |

## Appendix C Windows 2000 (contents online)

- |                                   |                               |
|-----------------------------------|-------------------------------|
| C.1 History A925                  | C.6 Networking A952           |
| C.2 Design Principles A926        | C.7 Programmer Interface A957 |
| C.3 System Components A927        | C.8 Summary A964              |
| C.4 Environmental Subsystems A943 | Exercises A964                |
| C.5 File System A945              | Bibliographical Notes A965    |

## Bibliography 855

## Credits 885

## Index 887

## Another defining moment in the evolution of operating systems

Small footprint operating systems, such as those driving the handheld devices that the baby dinosaurs are using on the cover, are just one of the cutting-edge applications you'll find in Silberschatz, Galvin, and Gagne's *Operating System Concepts*, Seventh Edition.

By staying current, remaining relevant, and adapting to emerging course needs, this market-leading text has continued to define the operating systems course. This Seventh Edition not only presents the latest and most relevant systems, it also digs deeper to uncover those fundamental concepts that have remained constant throughout the evolution of today's operating systems. With this strong conceptual foundation in place, students can more easily understand the details related to specific systems.

### New Adaptations

- Increased coverage of user perspective in Chapter 1.
- Increased coverage of OS design throughout.
- A new chapter on real-time and embedded systems (Chapter 19).
- A new chapter on multimedia (Chapter 20).
- Additional coverage of security and protection.
- Additional coverage of distributed programming.
- New exercises at the end of each chapter.
- New programming assignments and projects at the end of each chapter.
- New student-focused pedagogy and a new two-color design to enhance the learning process.

### The next evolutionary leap in the classroom—eGrade Plus!

Instructors, looking for a better way to create and assign homework, prepare class presentations, and manage your course? *eGrade Plus* provides an integrated suite of teaching and learning resources, along with a complete online version of the text, in one easy-to-use web site. Go to [www.wiley.com/college/egradeplus](http://www.wiley.com/college/egradeplus) for more information.

See why the most widely used operating systems text stays that way.

SEVENTH  
EDITION

# SILBERSCHATZ • GALVIN • GAGNE *Operating System Concepts*



[www.wiley.com/college/silberschatz](http://www.wiley.com/college/silberschatz)



## Part One

# Overview

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

An operating system is software that manages the computer hardware. The hardware must provide appropriate mechanisms to ensure the correct operation of the computer system and to prevent user programs from interfering with the proper operation of the system.

Internally, operating systems vary greatly in their makeup, since they are 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. These goals form the basis for choices among various algorithms and strategies.

Because an operating system is large and complex, it must be created piece by piece. Each of these pieces should be a well delineated portion of the system, with carefully defined inputs, outputs, and functions.

# Introduction



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

Before we can explore the details of computer system operation, we need to know something about system structure. We begin by discussing the basic functions of system startup, I/O, and storage. We also describe the basic computer architecture that makes it possible to write a functional operating system.

Because an operating system is large and complex, it must be created piece by piece. Each of these pieces should be a well-delineated portion of the system, with carefully defined inputs, outputs, and functions. In this chapter we provide a general overview of the major components of an operating system.

## CHAPTER OBJECTIVES

- To provide a grand tour of the major operating systems components.
- To provide coverage of basic computer system organization.

### 1.1 What Operating Systems Do

We begin our discussion by looking at the operating system's role in the overall computer system. A computer system can be divided roughly into four components: the *hardware*, the *operating system*, the *application programs*, and the *users* (Figure 1.1).

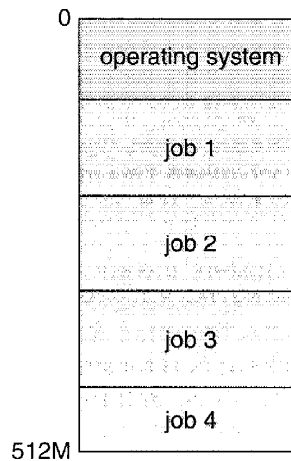


**Figure 1.7** Memory layout for a multiprogramming system.

non-multiprogrammed system, the CPU would sit idle. In a multiprogrammed system, the operating system simply switches to, and executes, another job. When *that* job needs to wait, the CPU is switched to *another* job, and so on. Eventually, the first job finishes waiting and gets the CPU back. As long as at least one job needs to execute, the CPU is never idle.

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

Multiprogrammed systems provide an environment in which the various system resources (for example, CPU, memory, and peripheral devices) are utilized effectively, but they do not provide for user interaction with the computer system. **Time sharing** (or **multitasking**) is a logical extension of multiprogramming. In time-sharing systems, the CPU executes multiple jobs by switching among them, but the switches occur so frequently that the users can interact with each program while it is running.

Time sharing requires an **interactive** (or **hands-on**) **computer system**, which provides direct communication between the user and the system. The user gives instructions to the operating system or to a program directly, using a input device such as a keyboard or a mouse, and waits for immediate results on an output device. Accordingly, the **response time** should be short—typically less than one second.

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

A time-shared operating system uses CPU scheduling and multiprogramming to provide each user with a small portion of a time-shared computer. Each user has at least one separate program in memory. A program loaded into

memory and executing is called 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 goes to a display for the user, and input comes from a user keyboard, mouse, or other device. Since interactive I/O typically runs at “people speeds,” it may take a long time to complete. Input, for example, may be bounded by the user’s typing speed; seven characters per second is fast for people but incredibly slow for computers. Rather than let the CPU sit idle as this interactive input takes place, the operating system will rapidly switch the CPU to the program of some other user.

Time-sharing and multiprogramming require several jobs to be kept simultaneously in memory. Since in general main memory is too small to accommodate all jobs, the jobs are kept initially on the disk in the **job pool**. This pool consists of all processes residing on disk awaiting allocation of main memory. If several jobs are ready to be brought into memory, and if there is not enough room for all of them, then the system must choose among them. Making this decision is **job scheduling**, which is discussed in Chapter 5. When the operating system selects a job from the job pool, it loads that job into memory for execution. Having several programs in memory at the same time requires some form of memory management, which is covered in Chapters 8 and 9. In addition, if several jobs are ready to run at the same time, the system must choose among them. Making this decision is **CPU scheduling**, which is discussed in Chapter 5. Finally, running multiple jobs concurrently requires 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.

In a time-sharing system, the operating system must ensure reasonable response time, which is sometimes accomplished through **swapping**, where processes are swapped in and out of main memory to the disk. A more common method for achieving this goal is **virtual memory**, a technique that allows the execution of a process that is not completely in memory (Chapter 9). The main advantage of the virtual-memory scheme is that it enables users to run programs that are larger than actual **physical memory**. Further, it abstracts main memory into a large, uniform array of storage, separating **logical memory** as viewed by the user from physical memory. This arrangement frees programmers from concern over memory-storage limitations.

Time-sharing systems must also provide a file system (Chapters 10 and 11). The file system resides on a collection of disks; hence, disk management must be provided (Chapter 12). Also, time-sharing systems provide a mechanism for protecting resources from inappropriate use (Chapter 14). To ensure orderly execution, the system must provide mechanisms for job synchronization and communication (Chapter 6), and it may ensure that jobs do not get stuck in a deadlock, forever waiting for one another (Chapter 7).

## 1.5 Operating-System Operations

As mentioned earlier, 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

with possibly disastrous results. Recent versions of the Intel CPU, such as the Pentium, do provide dual-mode operation. Accordingly, most contemporary operating systems, such as Microsoft Windows 2000 and Windows XP, and Linux and Solaris for x86 systems, take advantage of this feature and provide greater protection for the operating system.

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

### 1.5.2 Timer

We must ensure that the operating system maintains control over the CPU. We must prevent a user program from getting stuck in an infinite loop or not calling system services and never returning control to the operating system. To accomplish this goal, we can use a **timer**. A timer can be set to interrupt the computer after a specified period. The period may be fixed (for example, 1/60 second) or variable (for example, from 1 millisecond to 1 second). A **variable timer** is generally implemented by a fixed-rate clock and a counter. The operating system sets the counter. Every time the clock ticks, the counter is decremented. When the counter reaches 0, an interrupt occurs. For instance, a 10-bit counter with a 1-millisecond clock allows interrupts at intervals from 1 millisecond to 1,024 milliseconds, in steps of 1 millisecond.

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

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

## 1.6 Process Management

A program does nothing unless its instructions are executed by a CPU. A program in execution, as mentioned, is a process. A time-shared user program such as a compiler is a process. A word-processing program being run by an

individual user on a PC is a process. A system task, such as sending output to a printer, can also be a process (or at least part of one). For now, you can consider a process to be a job or a time-shared program, but later you will learn that the concept is more general. As we shall see in Chapter 3, 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, various 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 and display on the terminal the desired information. When the process terminates, the operating system will reclaim any reusable resources.

We emphasize that a program by itself is not a process; a program is a *passive* entity, such as the contents of a file stored on disk, whereas a process is an *active* entity. A single-threaded process has one **program counter** specifying the next instruction to execute. (Threads will be covered in Chapter 4.) The execution of such a process must be sequential. The CPU executes one instruction of the process after another, until the process completes. Further, at any time, one instruction at most 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. A multithreaded process has multiple program counters, each pointing to the next instruction to execute for a given thread.

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 on a single CPU, for example.

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

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

We discuss process-management techniques in Chapters 3 through 6.

## 1.7 Memory Management

As we discussed in Section 1.2.2, the main memory is central to the operation of a modern computer system. Main memory is a large array of words or bytes,

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 system calls are used to access this information. Generally, calls are also used to reset the process information (get process attributes and set process attributes). In Section 3.1.3, we discuss what information is normally kept.

### 2.4.5 Communication

There are two common models of interprocess communication: the message-passing model and the shared-memory model. In the **message-passing model**, the communicating processes exchange messages with one another to transfer information. Messages can be exchanged between the processes either directly or indirectly through a common mailbox. 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 system 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. A host also has a network identifier, such as an IP address. Similarly, each process has a *process name*, and this name is translated into an identifier by which the operating system can refer to the process. The `get hostid` and `get processid` system calls do this translation. The 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 communication. 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 systems programs provided for that purpose. They execute a `wait for connection` call and are awakened when a connection is made. The source of the communication, known as the *client*, and the receiving daemon, known as a *server*, then exchange messages by using `read message` and `write message` system calls. The `close connection` call terminates the communication.

In the **shared-memory model**, processes use shared memory `create` and `shared memory attach` system calls to create and 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's memory. Shared memory requires that two or more processes agree to remove this restriction. They can then exchange information by reading and writing data in the shared areas. The form of the data and the location are determined by the 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. In Chapter 4, we look at a variation of the process scheme—threads—in which memory is shared by default.

Both of the models just discussed are common in operating systems, and most systems implement both. Message passing is useful for exchanging smaller amounts of data, because no conflicts need be avoided. It is also easier to implement than is shared memory for intercomputer communication. Shared

memory allows maximum speed and convenience of communication, since it can be done at memory speeds when it takes place within a computer. Problems exist, however, in the areas of protection and synchronization between the processes sharing memory.

## 2.5 System Programs

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

- **File management.** These programs create, delete, copy, rename, print, dump, list, and generally manipulate files and directories.
- **Status information.** Some programs simply ask the system for the date, time, amount of available memory or disk space, number of users, or similar status information. Others are more complex, providing detailed performance, logging, and debugging information. Typically, these programs format and print the output to the terminal or other output devices or files or display it in a window of the GUI. Some systems also support a **registry**, which is used to store and retrieve configuration information.
- **File modification.** Several text editors may be available to create and modify the content of files stored on disk or other storage devices. There may also be special commands to search contents of files or perform transformations of the text.
- **Programming-language support.** Compilers, assemblers, debuggers and interpreters for common programming languages (such as C, C++, Java, Visual Basic, and PERL) are often provided to the user with the operating system.
- **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 as well.
- **Communications.** These programs provide the mechanism for creating virtual connections among processes, users, and computer systems. They allow users to send messages to one another's screens, to browse web pages, to send electronic-mail messages, to log in remotely, or to transfer files from one machine to another.

In addition to systems programs, most operating systems are supplied with programs that are useful in solving common problems or performing common operations. Such programs include web browsers, word processors and text formatters, spreadsheets, database systems, compilers, plotting and

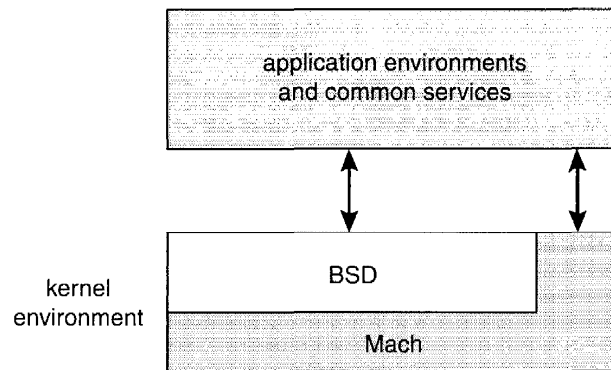


Figure 2.14 The Mac OS X structure.

## 2.8 Virtual Machines

The layered approach described in Section 2.7.2 is taken to its logical conclusion in the concept of a **virtual machine**. The fundamental idea behind a virtual machine is to abstract the hardware of a single computer (the CPU, memory, disk drives, network interface cards, and so forth) into several different execution environments, thereby creating the illusion that each separate execution environment is running its own private computer.

By using CPU scheduling (Chapter 5) and virtual-memory techniques (Chapter 9), an operating system can create the illusion that a process has its own processor with its own (virtual) memory. Normally, a process has additional features, such as system calls and a file system, that are not provided by the bare hardware. The virtual-machine approach does not provide any such additional functionality but rather provides an interface that is *identical* to the underlying bare hardware. Each process is provided with a (virtual) copy of the underlying computer (Figure 2.15).

There are several reasons for creating a virtual machine, all of which are fundamentally related to being able to share the same hardware yet run several different execution environments (that is, different operating systems) concurrently. We will explore the advantages of virtual machines in more detail in Section 2.8.2. Throughout much of this section, we discuss the VM operating system for IBM systems, as it provides a useful working example; furthermore IBM pioneered the work in this area.

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

Users thus are given their own virtual machines. They can then run any of the operating systems or software packages that are available on the underlying

## 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. Systems consist of a collection of processes: Operating-system processes execute system code, and user processes execute user code. All these processes may execute concurrently.

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

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

# Processes



Early computer systems allowed only one program to be executed at a time. This program had complete control of the system and had access to all the system's resources. In contrast, current-day computer systems allow multiple programs to be loaded into memory and executed concurrently. This evolution required firmer control and more compartmentalization of the various programs; and 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 is, 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. Potentially, all these processes can 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.

## CHAPTER OBJECTIVES

- To introduce the notion of a process — a program in execution, which forms the basis of all computation.
- To describe the various features of processes, including scheduling, creation and termination, and communication.
- To describe communication in client–server systems.

### 3.1 Process Concept

A question that arises in discussing operating systems involves what to call all the CPU activities. A batch system executes *jobs*, whereas a time-shared system has *user programs*, or *tasks*. Even on a single-user system such as Microsoft Windows, a user may be able to run several programs at one time: a word processor, a web browser, and an e-mail package. Even if the user can execute

### 3.4 Interprocess Communication

Processes executing concurrently 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. Any process that does not share data with any other process is independent. 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 such information.
- **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 or threads, as we discussed in Chapter 2.
- **Convenience.** Even an individual user may work on many tasks at the same time. For instance, a user may be editing, printing, and compiling in parallel.

Cooperating processes require an **interprocess communication (IPC)** mechanism that will allow them to exchange data and information. There are two fundamental models of interprocess communication: (1) **shared memory** and (2) **message passing**. In the shared-memory model, a region of memory that is shared by cooperating processes is established. Processes can then exchange information by reading and writing data to the shared region. In the message-passing model, communication takes place by means of messages exchanged between the cooperating processes. The two communications models are contrasted in Figure 3.13.

Both of the models just discussed are common in operating systems, and many systems implement both. Message passing is useful for exchanging smaller amounts of data, because no conflicts need be avoided. Message passing 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. Shared memory is faster than message passing, as message-passing systems are typically implemented using system calls and thus require the more time-consuming task of kernel intervention. In contrast, in shared-memory systems, system calls are required only to establish shared-memory regions. Once shared memory is established, all accesses are treated as routine memory accesses, and no assistance from the kernel is required. In the remainder of this section, we explore each of these IPC models in more detail.

# 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 basic CPU-scheduling concepts and present several CPU-scheduling algorithms. We also consider the problem of selecting an algorithm for a particular system.

In Chapter 4, we introduced threads to the process model. On operating systems that support them, it is kernel-level threads—not processes—that are in fact being scheduled by the operating system. However, the terms **process scheduling** and **thread scheduling** are often used interchangeably. In this chapter, we use *process scheduling* when discussing general scheduling concepts and *thread scheduling* to refer to thread-specific ideas.

## CHAPTER OBJECTIVES

- To introduce CPU scheduling, which is the basis for multiprogrammed operating systems.
- To describe various CPU-scheduling algorithms.
- To discuss evaluation criteria for selecting a CPU-scheduling algorithm for a particular system.

### 5.1 Basic Concepts

In a single-processor system, only one process can run at a time; any others must wait until the CPU is free and can be rescheduled. The objective of multiprogramming is to have some process running at all times, to maximize CPU utilization. The idea 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 then just sits 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

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, at completion of I/O)
4. When a process terminates

For situations 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 situations 2 and 3.

When scheduling takes place only under circumstances 1 and 4, we say that the scheduling scheme is **nonpreemptive** or **cooperative**; otherwise, it 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 was used by Microsoft Windows 3.x; Windows 95 introduced preemptive scheduling, and all subsequent versions of Windows operating systems have used preemptive scheduling. The Mac OS X operating system for the Macintosh uses preemptive scheduling; previous versions of the Macintosh operating system relied on cooperative scheduling. Cooperative scheduling 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 associated with access to shared data. Consider the case of two processes that share data. While one is updating the data, it is preempted so that the second process can run. The second process then tries to read the data, which are in an inconsistent state. In such situations, we need new mechanisms to coordinate access to shared data; we discuss this topic in Chapter 6.

Preemption also affects 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. Certain 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 19.5.

Because interrupts can, by definition, occur at any time, and because they cannot always be ignored by the kernel, the sections of code affected by interrupts must be guarded from simultaneous use. The operating system needs to accept interrupts at almost all times; otherwise, input might be lost or output overwritten. So that these sections of code are not accessed concurrently by several processes, they disable interrupts at entry and reenables interrupts at exit. It is important to note that sections of code that disable interrupts do not occur very often and typically contain few instructions.

### 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 the following:

- 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, since 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 the choice of a particular algorithm 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 which algorithm is judged to be best. The criteria include the following:

- **CPU utilization.** We want to keep the CPU as busy as possible. Conceptually, CPU utilization can range from 0 to 100 percent. In a real system, it should range from 40 percent (for a lightly loaded system) to 90 percent (for a heavily used system).
- **Throughput.** If the CPU is busy executing processes, then work is being done. One measure of work is the number of processes that are completed per time unit, called *throughput*. For long processes, this rate may be one process per hour; for short transactions, it may be 10 processes per second.
- **Turnaround time.** From the point of view of a particular process, the important criterion is how long it takes to execute that process. The interval from the time of submission of a process to the time of completion is the *turnaround time*. Turnaround time is the sum of the periods spent waiting to get into memory, waiting in the ready queue, executing on the CPU, and doing I/O.
- **Waiting time.** The CPU scheduling algorithm does not affect the amount of time during which a process executes or does I/O; it affects only the amount of time that a process spends waiting in the ready queue. *Waiting time* is the sum of the periods spent waiting in the ready queue.
- **Response time.** In an interactive system, turnaround time may not be the best criterion. Often, a process can produce some output fairly early and can continue computing new results while previous results are being