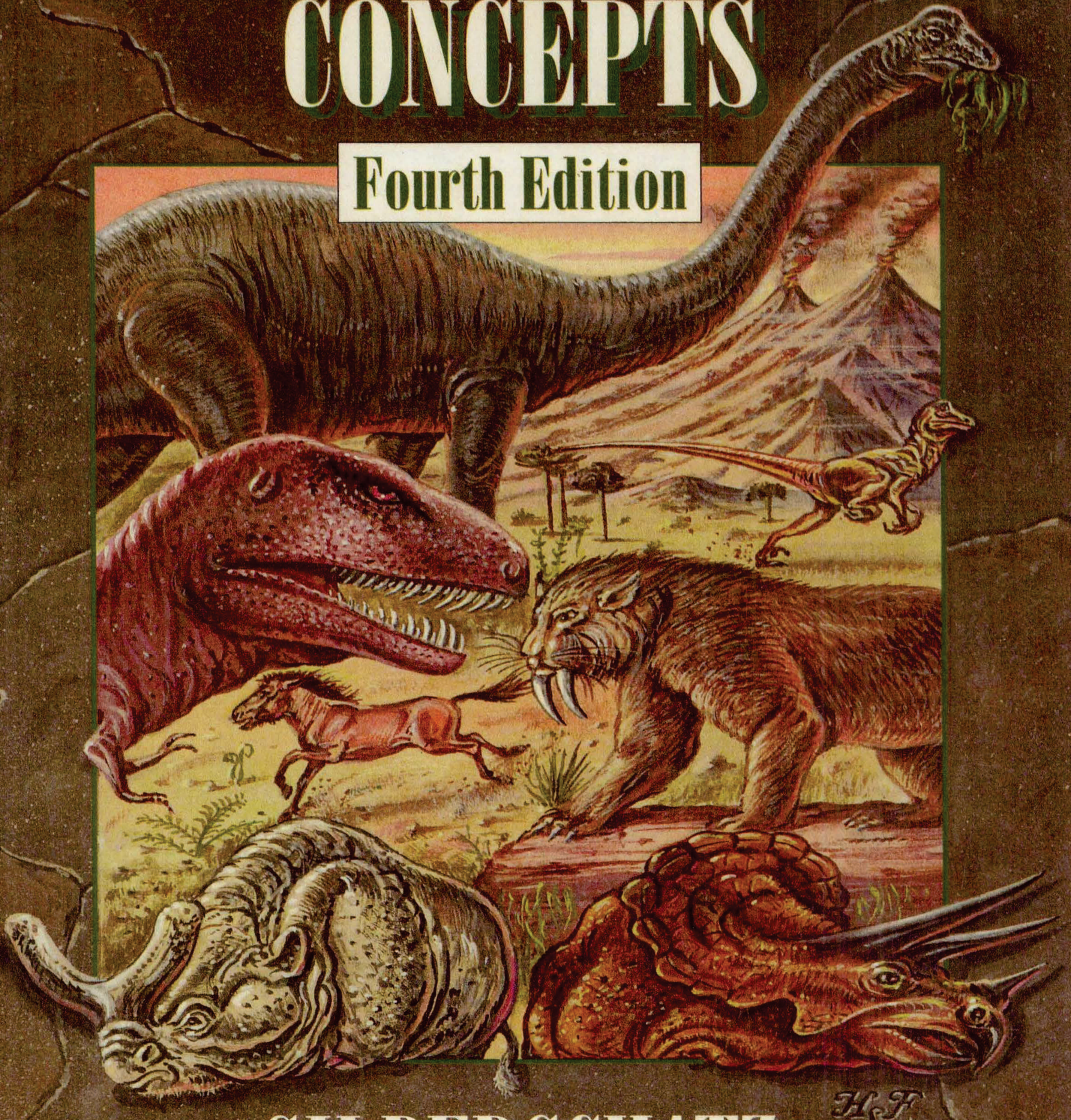
# OPERATING SYSTEM CONCEPTS

Fourth Edition



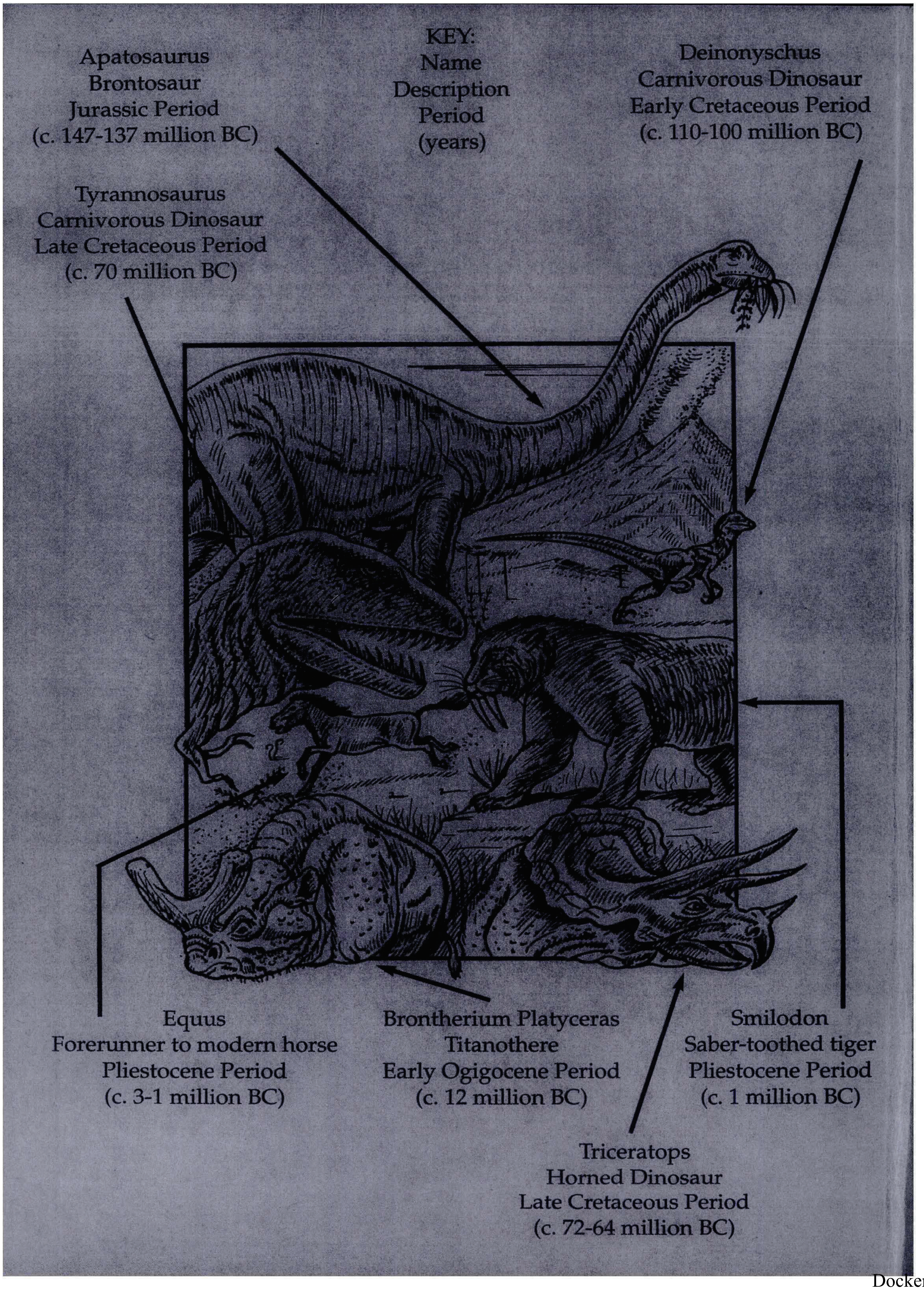
SILBERSCHATZ
GALVIN

SILBERSCHATZ
GALVIN

OPERATING SYSTEM CONCEPT

4-975-321

BookHolders.com


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*

**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

Reprinted with corrections January, 1995

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

ISBN 0-201-50480-4
**8 9 10-MA-99 98 97 96**

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

*Avi Silberschatz*

*To Carla and Gwendolyn.*

*Peter Galvin*

# 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
Exercises 242
Bibliographic Notes 245

## 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

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

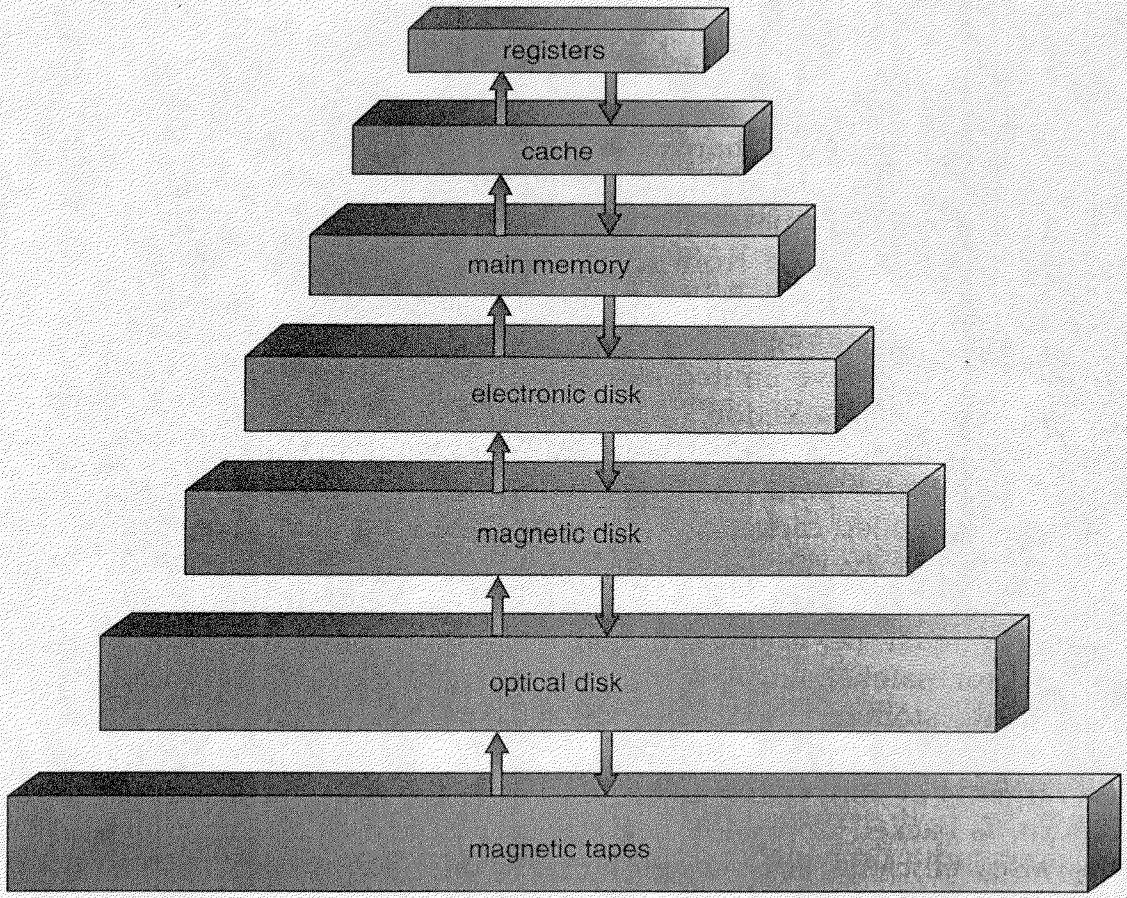


**Figure 2.6** Storage-device hierarchy.

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 pages cannot be modified; thus, they may be discarded when desired. This scheme can reduce significantly the time to service a page fault, since it reduces I/O time by one-half *if* the page is not modified.

Page replacement is basic to demand paging. It completes the separation between logical memory and physical memory. With this mechanism, a very large virtual memory can be provided for programmers on a smaller physical memory. With non-demand paging, user addresses were mapped into physical addresses, allowing the two sets of addresses to be quite different. All of the pages of a process still must be in physical memory, however. With demand paging, the size of the logical address space is no longer constrained by physical memory. If we have a user process of 20 pages, we can execute it in 10 frames simply by using demand paging, and using a replacement algorithm to find a free frame whenever necessary. If a page that has been modified is to be replaced, its contents are copied to the disk. A later reference to that page will cause a page fault. At that time, the page will be brought back into memory, perhaps replacing some other page in the process.

We must solve two major problems to implement demand paging: We must develop a *frame-allocation algorithm* and a *page-replacement algorithm.* If we have multiple processes in memory, we must decide how many frames to allocate to each process. Further, when page replacement is required, we must select the frames that are to be replaced. Designing appropriate algorithms to solve these problems is an important task, because disk I/O is so expensive. Even slight improvements in demand-paging methods yield large gains in system performance.

## 9.5 ■ Page-Replacement Algorithms

There are many different page-replacement algorithms. Probably every operating system has its own unique replacement scheme. How do we select a particular replacement algorithm? In general, we want the one with the lowest *page-fault rate.*

We evaluate an algorithm by running it on a particular string of memory references and computing the number of page faults. The string of memory references is called a *reference string.* We can generate reference strings artificially (by a random-number generator, for example) or by tracing a given system and recording the address of each memory reference. The latter choice produces a large number of data (on the order of 1 million addresses per second). To reduce the number of data, we note two things.

First, for a given page size (and the page size is generally fixed by the hardware or system), we need to consider only the page number, not the entire address. Second, if we have a reference to a page *p,* then any *immediately* following references to page $p$ will never cause a page fault. Page $p$ will be in memory after the first reference; the immediately following references will not fault.

For example, if we trace a particular process, we might record the following address sequence:

0100, 0432, 0101, 0612, 0102, 0103, 0104, 0101, 0611, 0102, 0103,
0104, 0101, 0610, 0102, 0103, 0104, 0101, 0609, 0102, 0105,

which, at 100 bytes per page, is reduced to the following reference string

1, 4, 1, 6, 1, 6, 1, 6, 1, 6, 1.

To determine the number of page faults for a particular reference string and page-replacement algorithm, we also need to know the number of page frames available. Obviously, as the number of frames available increases, the number of page faults will decrease. For the reference string considered previously, for example, if we had three or more frames, we would have only three faults, one fault for the first reference to each page. On the other hand, with only one frame available, we would have a replacement with every reference, resulting in 11 faults. In general, we expect a curve such as that in Figure 9.7. As the number of frames increases, the number of page faults drops to some minimal level. Of course, adding physical memory increases the number of frames.

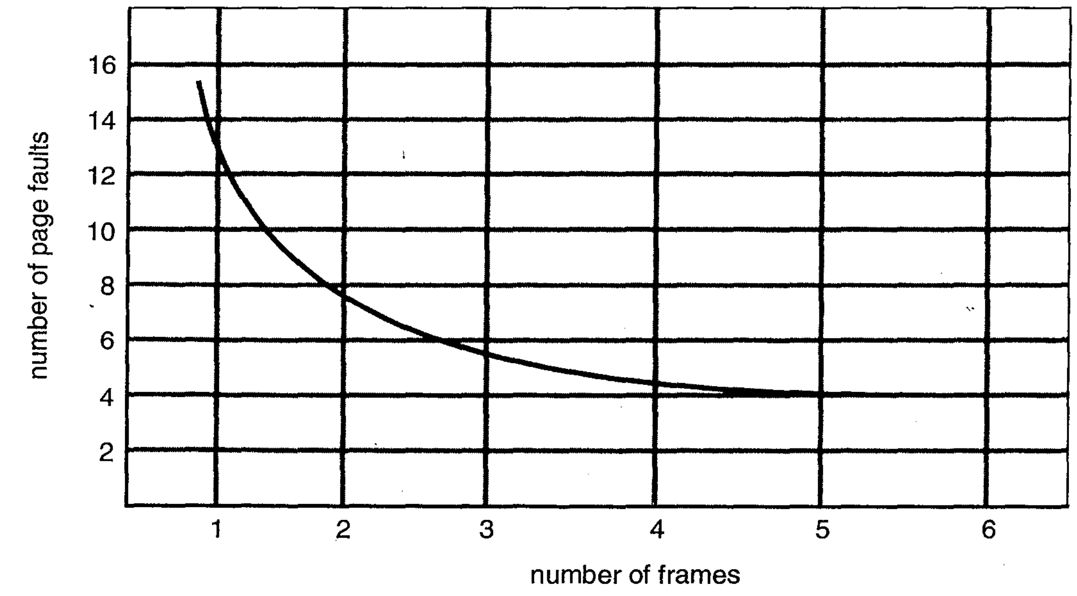


**Figure 9.7** Graph of page faults versus the number of frames.

To illustrate the page-replacement algorithms, we shall use the reference string

7, 0, 1, 2, 0, 3, 0, 4, 2, 3, 0, 3, 2, 1, 2, 0, 1, 7, 0, 1

for a memory with three frames.

### 9.5.1 FIFO Algorithm

The simplest page-replacement algorithm is a FIFO algorithm. A FIFO replacement algorithm associates with each page the time when that page was brought into memory. When a page must be replaced, the oldest page is chosen. Notice that it is not strictly necessary to record the time when a page is brought in. We can create a FIFO queue to hold all pages in memory. We replace the page at the head of the queue. When a page is brought into memory, we insert it at the tail of the queue.

For our example reference string, our three frames are initially empty. The first three references (7, 0, 1) cause page faults, and are brought into these empty frames. The next reference (2) replaces page 7, because page 7 was brought in first. Since 0 is the next reference and 0 is already in memory, we have no fault for this reference. The first reference to 3 results in page 0 being replaced, since it was the first of the three pages in memory (0, 1, and 2) to be brought in. This replacement means that the next reference, to 0, will fault. Page 1 is then replaced by page 0. This process continues as shown in Figure 9.8. Every time a fault occurs, we show which pages are in our three frames. There are 15 faults altogether.

The FIFO page-replacement algorithm is easy to understand and program. However, its performance is not always good. The page replaced may be an initialization module that was used a long time ago and is no longer needed. On the other hand, it could contain a heavily used variable that was initialized early and is in constant use.

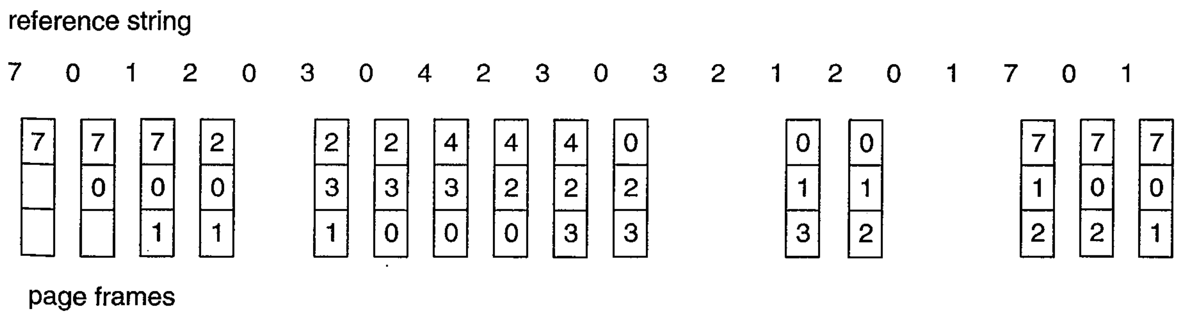


**Figure 9.8** FIFO page-replacement algorithm.

Notice that, even if we select for replacement a page that is in active use, everything still works correctly. After we page out an active page to bring in a new one, a fault occurs almost immediately to retrieve the active page. Some other page will need to be replaced to bring the active page back into memory. Thus, a bad replacement choice increases the page-fault rate and slows process execution, but does not cause incorrect execution.

To illustrate the problems that are possible with a FIFO page-replacement algorithm, we consider the reference string

1, 2, 3, 4, 1, 2, 5, 1, 2, 3, 4, 5.

Figure 9.9 shows the curve of page faults versus the number of available frames. We notice that the number of faults for four frames (10) is *greater* than the number of faults for three frames (nine)! This result is most unexpected and is known as *Belady's anomaly.* Belady's anomaly reflects the fact that, for some page-replacement algorithms, the page-fault rate may *increase* as the number of allocated frames increases. We would expect that giving more memory to a process would improve its performance. In some early research, investigators noticed that this assumption was not always true. Belady's anomaly was discovered as a result.

## 9.5.2 Optimal Algorithm

One result of the discovery of Belady's anomaly was the search for an *optimal* page-replacement algorithm. An optimal page-replacement algorithm has the lowest page-fault rate of all algorithms. An optimal

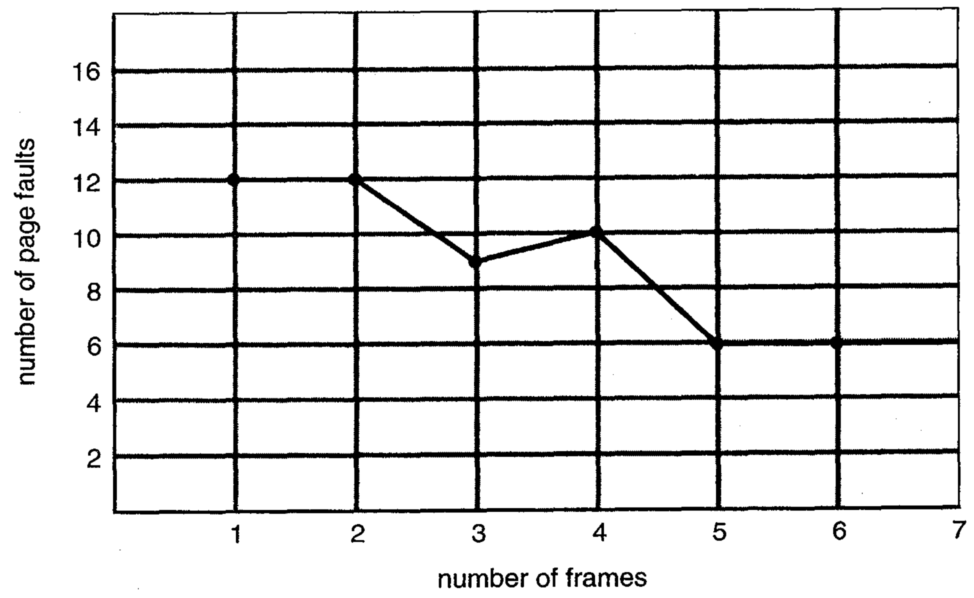


**Figure 9.9** Page-fault curve for FIFO replacement on a reference string.

algorithm will never suffer from Belady's anomaly. An optimal page-replacement algorithm exists, and has been called OPT or MIN. It is simply

> Replace the page that will not be used
> for the longest period of time.

Use of this page-replacement algorithm guarantees the lowest possible page-fault rate for a fixed number of frames.

For example, on our sample reference string, the optimal page-replacement algorithm would yield nine page faults, as shown in Figure 9.10. The first three references cause faults that fill the three empty frames. The reference to page 2 replaces page 7, because 7 will not be used until reference 18, whereas page 0 will be used at 5, and page 1 at 14. The reference to page 3 replaces page 1, as page 1 will be the last of the three pages in memory to be referenced again. With only nine page faults, optimal replacement is much better than a FIFO algorithm, which had 15 faults. (If we ignore the first three, which all algorithms must suffer, then optimal replacement is twice as good as FIFO replacement.) In fact, no replacement algorithm can process this reference string in three frames with less than nine faults.

Unfortunately, the optimal page-replacement algorithm is difficult to implement, because it requires future knowledge of the reference string. (We encountered a similar situation with the SJF CPU-scheduling algorithm in Section 5.3.2.) As a result, the optimal algorithm is used mainly for comparison studies. For instance, it may be quite useful to know that, although a new algorithm is not optimal, it is within 12.3 percent of optimal at worst and within 4.7 percent on average.

### 9.5.3 LRU Algorithm

If the optimal algorithm is not feasible, perhaps an approximation to the optimal algorithm is possible. The key distinction between the FIFO and OPT algorithms (other than looking backward or forward in time) is that the

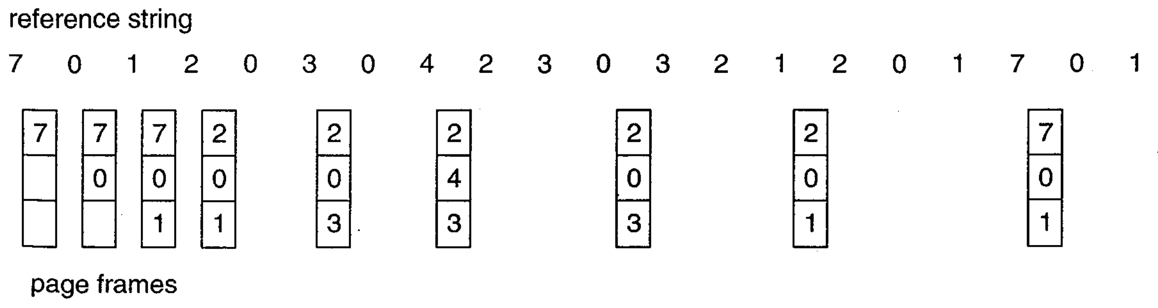


**Figure 9.10** Optimal page-replacement algorithm.

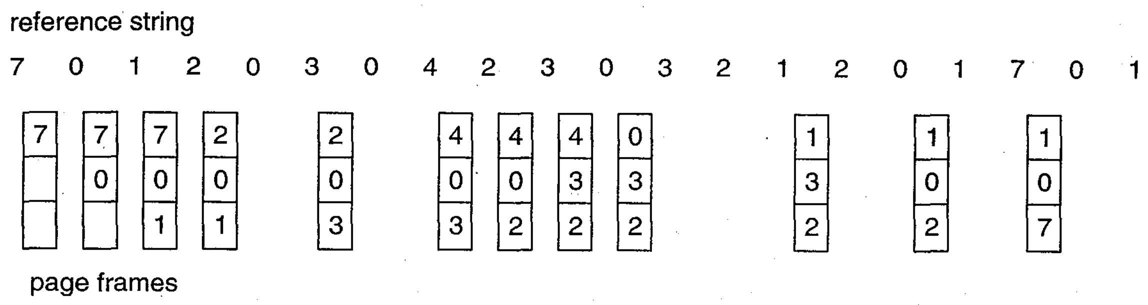


**Figure 9.11** LRU page-replacement algorithm.

FIFO algorithm uses the time when a page was brought into memory; the OPT algorithm uses the time when a page is to be *used.* If we use the recent past as an approximation of the near future, then we will replace the page that *has not been used* for the longest period of time (Figure 9.11). This approach is the *least recently used (LRU)* algorithm.

LRU replacement associates with each page the time of that page's last use. When a page must be replaced, LRU chooses that page that has not been used for the longest period of time. This strategy is the optimal page-replacement algorithm looking backward in time, rather than forward. (Strangely, if we let $S^R$ be the reverse of a reference string $S$, then the page-fault rate for the OPT algorithm on $S$ is the same as the page-fault rate for the OPT algorithm on $S^R$. Similarly, the page-fault rate for the LRU algorithm on $S$ is the same as the page-fault rate for the LRU algorithm on $S^R$.)

The result of applying LRU replacement to our example reference string is shown in Figure 9.11. The LRU algorithm produces 12 faults. Notice that the first five faults are the same as the optimal replacement. When the reference to page 4 occurs, however, LRU replacement sees that, of the three frames in memory, page 2 was used least recently. The most recently used page is page 0, and just before that page 3 was used. Thus, the LRU algorithm replaces page 2, not knowing that page 2 is about to be used. When it then faults for page 2, the LRU algorithm replaces page 3 since, of the three pages in memory {0, 3, 4}, page 3 is the least recently used. Despite these problems, LRU replacement with 12 faults is still much better than FIFO replacement with 15.

The LRU policy is often used as a page-replacement algorithm and is considered to be quite good. The major problem is *how* to implement LRU replacement. An LRU page-replacement algorithm may require substantial hardware assistance. The problem is to determine an order for the frames defined by the time of last use. Two implementations are feasible:

- **Counters:** In the simplest case, we associate with each page-table entry a time-of-use field, and add to the CPU a logical clock or counter. The clock is incremented for every memory reference. Whenever a reference to a page is made, the contents of the clock register are copied to the time-of-use field in the page table for that page. In this way, we always have the "time" of the last reference to each page. We replace the page with the smallest time value. This scheme requires a search of the page table to find the LRU page, and a write to memory (to the time-of-use field in the page table) for each memory access. The times must also be maintained when page tables are changed (due to CPU scheduling). Overflow of the clock must be considered.

- **Stack:** Another approach to implementing LRU replacement is to keep a *stack* of page numbers. Whenever a page is referenced, it is removed from the stack and put on the top. In this way, the top of the stack is always the most recently used page and the bottom is the LRU page (Figure 9.12). Because entries must be removed from the middle of the stack, it is best implemented by a doubly linked list, with a head and tail pointer. Removing a page and putting it on the top of the stack then requires changing six pointers at worst. Each update is a little more expensive, but there is no search for a replacement; the tail pointer points to the bottom of the stack, which is the LRU page. This approach is particularly appropriate for software or microcode implementations of LRU replacement.

Neither optimal replacement nor LRU replacement suffers from Belady's anomaly. There is a class of page-replacement algorithms, called *stack algorithms,* that can never exhibit Belady's anomaly. A stack algorithm is an algorithm for which it can be shown that the set of pages in memory for *n*

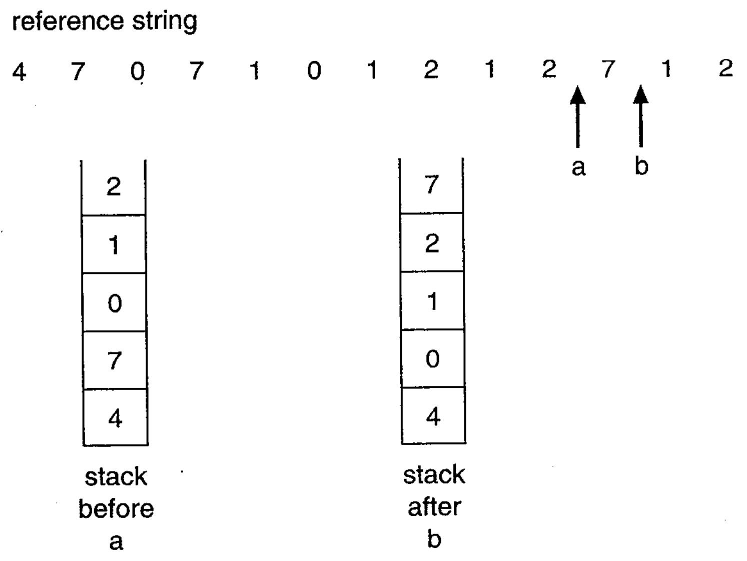


**Figure 9.12** Use of a stack to record the most recent page references.

frames is always a *subset* of the set of pages that would be in memory with $n + 1$ frames. For LRU replacement, the set of pages in memory would be the $n$ most recently referenced pages. If the number of frames is increased, these $n$ pages will still be the most recently referenced and so will still be in memory.

Note that neither implementation of LRU would be conceivable without hardware assistance beyond the standard TLB registers. The updating of the clock fields or stack must be done for *every* memory reference. If we were to use an interrupt for every reference, to allow software to update such data structures, it would slow every memory reference by a factor of at least 10, hence slowing every user process by a factor of 10. Few systems could tolerate that level of overhead for memory management.

### 9.5.4 LRU Approximation Algorithms

Few systems provide sufficient hardware support for true LRU page replacement. Some systems provide no hardware support, and other page-replacement algorithms (such as a FIFO algorithm) must be used. Many systems provide some help, however, in the form of a *reference bit*. The reference bit for a page is set, by the hardware, whenever that page is referenced (either a read or a write to any byte in the page). Reference bits are associated with each entry in the page table.

Initially, all bits are cleared (to 0) by the operating system. As a user process executes, the bit associated with each page referenced is set (to 1) by the hardware. After some time, we can determine which pages have been used and which have not been used by examining the reference bits. We do not know the *order* of use, but we know which pages were used and which were not used. This partial ordering information leads to many page-replacement algorithms that approximate LRU replacement.

#### 9.5.4.1 Additional-Reference-Bits Algorithm

We can gain additional ordering information by recording the reference bits at regular intervals. We can keep an 8-bit byte for each page in a table in memory. At regular intervals (say every 100 milliseconds), a timer interrupt transfers control to the operating system. The operating system shifts the reference bit for each page into the high-order bit of its 8-bit byte, shifting the other bits right 1 bit, discarding the low-order bit. These 8-bit shift registers contain the history of page use for the last eight time periods. If the shift register contains 00000000, then the page has not been used for eight time periods; a page that is used at least once each period would have a shift register value of 11111111.

A page with a history register value of 11000100 has been used more recently than has one with 01110111. If we interpret these 8-bit bytes as unsigned integers, the page with the lowest number is the LRU page, and it can be replaced. Notice that the numbers are not guaranteed to be unique, however. We can either replace (swap out) all pages with the smallest value, or use a FIFO selection among them.

The number of bits of history can be varied, of course, and would be selected (depending on the hardware available) to make the updating as fast as possible. In the extreme case, the number can be reduced to zero, leaving only the reference bit itself. This algorithm is called the *second-chance* page-replacement algorithm.

### 9.5.4.2 Second-Chance Algorithm

The basic algorithm of second-chance replacement is a FIFO replacement algorithm. When a page has been selected, however, we inspect its reference bit. If the value is 0, we proceed to replace this page. If the reference bit is 1, however, we give that page a second chance and move on to select the next FIFO page. When a page gets a second chance, its reference bit is cleared and its arrival time is reset to the current time. Thus, a page that is given a second chance will not be replaced until all other pages are replaced (or given second chances). In addition, if a page is used often enough to keep its reference bit set, it will never be replaced.

One way to implement the second-chance (sometimes referred to as the clock) algorithm is as a circular queue. A pointer indicates which page is to be replaced next. When a frame is needed, the pointer advances until it finds a page with a 0 reference bit. As it advances, it clears the reference bits (Figure 9.13). Once a victim page is found, the page is replaced and the new page is inserted in the circular queue in that position. Notice that, in the worst case, when all bits are set, the pointer cycles through the whole queue, giving each page a second chance. It clears all the reference bits before selecting the next page for replacement. Second-chance replacement degenerates to FIFO replacement if all bits are set.

### 9.5.4.3 Enhanced Second-Chance Algorithm

The second-chance algorithm described above can be enhanced by considering both the reference bit and the modify bit (Section 9.4) as an ordered pair. With these 2 bits, we have the following four possible classes:

1. (0,0) neither recently used nor modified - best page to replace
2. (0,1) not recently used but modified - not quite as good, because the page will need to be written out before replacement
3. (1,0) recently used but clean - probably will be used again soon
4. (1,1) recently used and modified - probably will be used again, and write out will be needed before replacing it

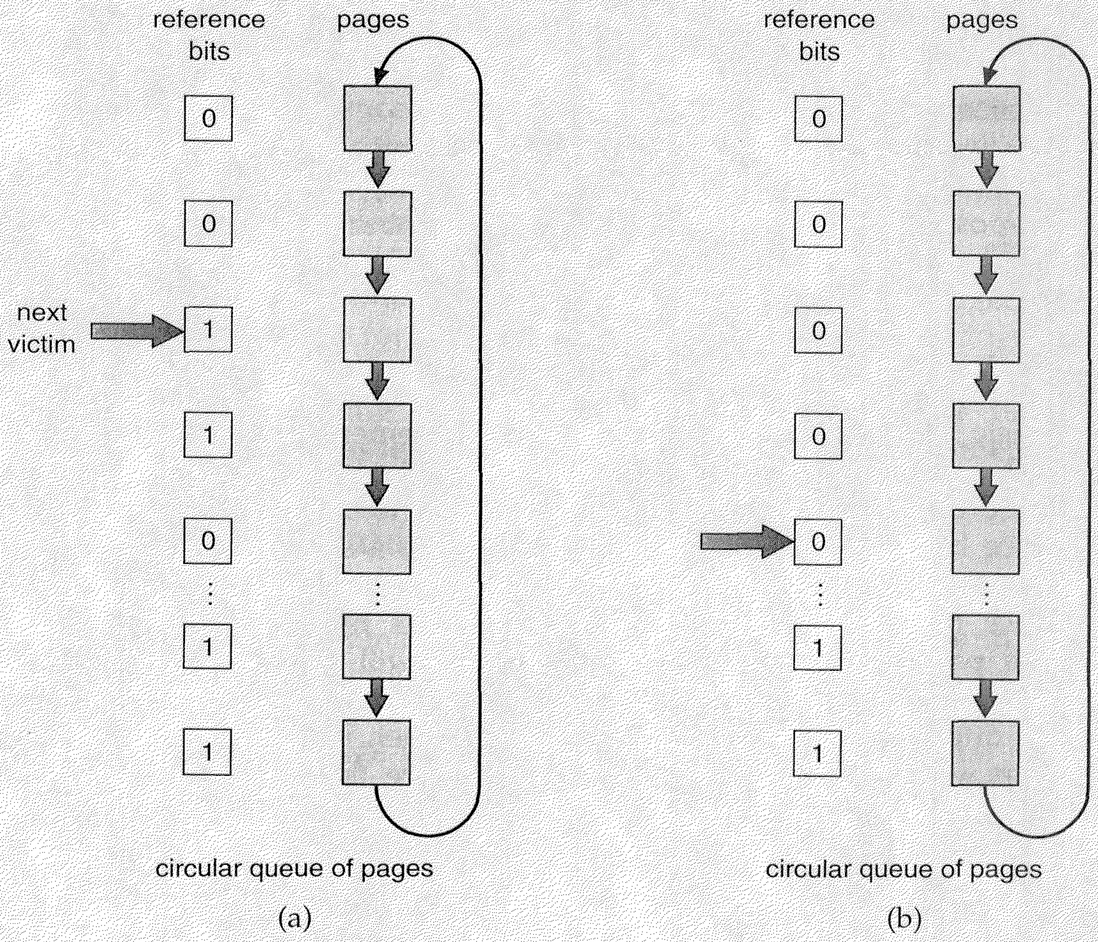


**Figure 9.13** Second-chance (clock) page-replacement algorithm.

When page replacement is called for, each page is in one of these four classes. We use the same scheme as the clock algorithm, but instead of examining whether the page to which we are pointing has the reference bit set to 1, we examine the class to which that page belongs. We replace the first page encountered in the lowest nonempty class. Notice that we may have to scan the circular queue several times before we find a page to be replaced.

This algorithm is used in the Macintosh virtual-memory-management scheme. The major difference between this algorithm and the simpler clock algorithm is that here we give preference to those pages that have been modified to reduce the number of I/Os required.

## 9.5.5 Counting Algorithms

There are many other algorithms that can be used for page replacement. For example, we could keep a counter of the number of references that have been made to each page, and develop the following two schemes.

- LFU **Algorithm:** The *least frequently used* (LFU) page-replacement algorithm requires that the page with the smallest count be replaced. The reason for this selection is that an actively used page should have a large reference count. This algorithm suffers from the situation in which a page is used heavily during the initial phase of a process, but then is never used again. Since it was used heavily, it has a large count and remains in memory even though it is no longer needed. One solution is to shift the counts right by 1 bit at regular intervals, forming an exponentially decaying average usage count.
- MFU **Algorithm:** The *most frequently used* (MFU) page-replacement algorithm is based on the argument that the page with the smallest count was probably just brought in and has yet to be used.

As you might expect, neither MFU nor LFU replacement is common. The implementation of these algorithms is fairly expensive, and they do not approximate OPT replacement very well.

### 9.5.6 Page Buffering Algorithm

Other procedures are often used in addition to a specific page-replacement algorithm. For example, systems commonly keep a *pool* of free frames. When a page fault occurs, a victim frame is chosen as before. However, the desired page is read into a free frame from the pool before the victim is written out. This procedure allows the process to restart as soon as possible, without waiting for the victim page to be written out. When the victim is later written out, its frame is added to the free-frame pool.

An expansion of this idea is to maintain a list of modified pages. Whenever the paging device is idle, a modified page is selected and is written to the disk. Its modify bit is then reset. This scheme increases the probability that a page will be clean when it is selected for replacement, and will not need to be written out.

Another modification is to keep a pool of free frames, but to remember which page was in each frame. Since the frame contents are not modified when a frame is written to the disk, the old page can be reused directly from the free-frame pool if it is needed before that frame is reused. No I/O is needed in this case. When a page fault occurs, we first check whether the desired page is in the free-frame pool. If it is not, we must select a free frame and read into it.

This technique is used in the VAX/VMS system, with a FIFO replacement algorithm. When the FIFO replacement algorithm mistakenly replaces a page that is still in active use, that page is quickly retrieved from the free-frame buffer, and no I/O is necessary. The free-frame buffer provides protection against the relatively poor, but simple, FIFO replacement algorithm. This method is necessary because the early versions of the VAX did not correctly implement the reference bit.

operating systems, such as from MIT's CTSS and the XDS-940 system, were also used.

Ritchie and Thompson worked quietly on UNIX for many years. Their work on the first version allowed them to move it to a PDP-11/20, for a second version. A third version resulted from their rewriting most of the operating system in the systems-programming language C, instead of the previously used assembly language. C was developed at Bell Laboratories to support UNIX. UNIX was also moved to larger PDP-11 models, such as the 11/45 and 11/70. Multiprogramming and other enhancements were added when it was rewritten in C and moved to systems (such as the 11/45) that had hardware support for multiprogramming.

As UNIX developed, it became widely used within Bell Laboratories and gradually spread to a few universities. The first version widely available outside Bell Laboratories was Version 6, released in 1976. (The version number for early UNIX systems corresponds to the edition number of the *UNIX Programmer's Manual* that was current when the distribution was made; the code and the manuals were revised independently.)

In 1978, Version 7 was distributed. This UNIX system ran on the PDP-11/70 and the Interdata 8/32, and is the ancestor of most modern UNIX systems. In particular, it was soon ported to other PDP-11 models and to the VAX computer line. The version available on the VAX was known as 32V. Research has continued since then.

After the distribution of Version 7 in 1978, the UNIX Support Group (USG) assumed administrative control and responsibility from the Research Group for distributions of UNIX within AT&T, the parent organization for Bell Laboratories. UNIX was becoming a product, rather than simply a research tool. The Research Group has continued to develop their own version of UNIX, however, to support their own internal computing. Next came Version 8, which included a facility called the *stream I/O system* that allows flexible configuration of kernel IPC modules. It also contained RFS, a remote file system similar to Sun's NFS. Next came Versions 9 and 10 (the latter version, released in 1989, is available only within Bell Laboratories).

USG mainly provided support for UNIX within AT&T. The first external distribution from USG was System III, in 1982. System III incorporated features of Version 7, and 32V, and also of several UNIX systems developed by groups other than Research. Features of UNIX/RT, a real-time UNIX system, as well as numerous portions of the Programmer's Work Bench (PWB) software tools package were included in System III.

USG released System V in 1983; it is largely derived from System III. The divestiture of the various Bell operating companies from AT&T has left AT&T in a position to market System V aggressively. USG was restructured as the UNIX System Development Laboratory (USDL), which released UNIX System V Release 2 (V.2) in 1984. UNIX System V Release 2, Version 4 (V.2.4) added a new implementation of virtual memory with copy-on-write paging and shared memory. USDL was in turn replaced by AT&T

wait for its completion. The parent process uses **vfork** to produce the child process. Because the child process wishes to use an **execve** immediately to change its virtual address space completely, there is no need for a complete copy of the parent process. Such data structures as are necessary for manipulating pipes may be kept in registers between the **vfork** and the **execve**. Files may be closed in one process without affecting the other process, since the kernel data structures involved depend on the user structure, which is not shared. The parent is suspended when it calls **vfork** until the child either calls **execve** or terminates, so that the parent will not change memory that the child needs.

When the parent process is large, **vfork** can produce substantial savings in system CPU time. However, it is a fairly dangerous system call, since any memory change occurs in both processes until the **execve** occurs. An alternative is to share all pages by duplicating the page table, but to mark the entries of both page tables as *copy-on-write*. The hardware protection bits are set to trap any attempt to write in these shared pages. If such a trap occurs, a new frame is allocated and the shared page is copied to the new frame. The page tables are adjusted to show that this page is no longer shared (and therefore no longer needs to be write-protected), and execution can resume.

An **execve** system call creates no new process or user structure; rather, the text and data of the process are replaced. Open files are preserved (although there is a way to specify that certain file descriptors are to be closed on an **execve**). Most signal-handling properties are preserved, but arrangements to call a specific user routine on a signal are canceled, for obvious reasons. The process identifier and most other properties of the process are unchanged.

### 19.5.2 CPU Scheduling

*CPU scheduling* in UNIX is designed to benefit interactive processes. Processes are given small CPU time slices by a priority algorithm that reduces to round-robin scheduling for CPU-bound jobs.

Every process has a *scheduling priority* associated with it; larger numbers indicate lower priority. Processes doing disk I/O or other important tasks have priorities less than "pzero" and cannot be killed by signals. Ordinary user processes have positive priorities and thus are all less likely to be run than are any system process, although user processes can set precedence over one another through the *nice* command.

The more CPU time a process accumulates, the lower (more positive) its priority becomes, and vice versa, so there is negative feedback in CPU scheduling and it is difficult for a single process to take all the CPU time. Process aging is employed to prevent starvation.

Older UNIX systems used a 1-second quantum for the round-robin scheduling. 4.3BSD reschedules processes every 0.1 second and recomputes by copying the data from the sender to the receiver. Because this technique can be inefficient, especially in the case of large messages, Mach optimizes this procedure. The data referenced by a pointer in a message being sent to a port on the same system are not copied between the sender and the receiver. Instead, the address map of the receiving task is modified to include a copy-on-write copy of the pages of the message. This operation is *much* faster than a data copy, and makes message passing efficient. In essence, message passing is implemented via virtual-memory management.

In Version 2.5, this operation was implemented in two phases. A pointer to a region of memory caused the kernel to map that region of memory into its own space temporarily, setting the sender's memory map to copy-on-write mode to ensure that any modifications did not affect the original version of the data. When a message was received at its destination, the kernel moved its mapping to the receiver's address space, using a newly allocated region of virtual memory within that task.

In Version 3, this process was simplified. The kernel creates a data structure that would be a copy of the region if it were part of an address map. On receipt, this data structure is added to the receiver's map and becomes a copy accessible to the receiver.

The newly allocated regions in a task do not need to be contiguous with previous allocations, so Mach virtual memory is said to be *sparse*, consisting of regions of data separated by unallocated addresses. A full message transfer is shown in Figure 20.6.

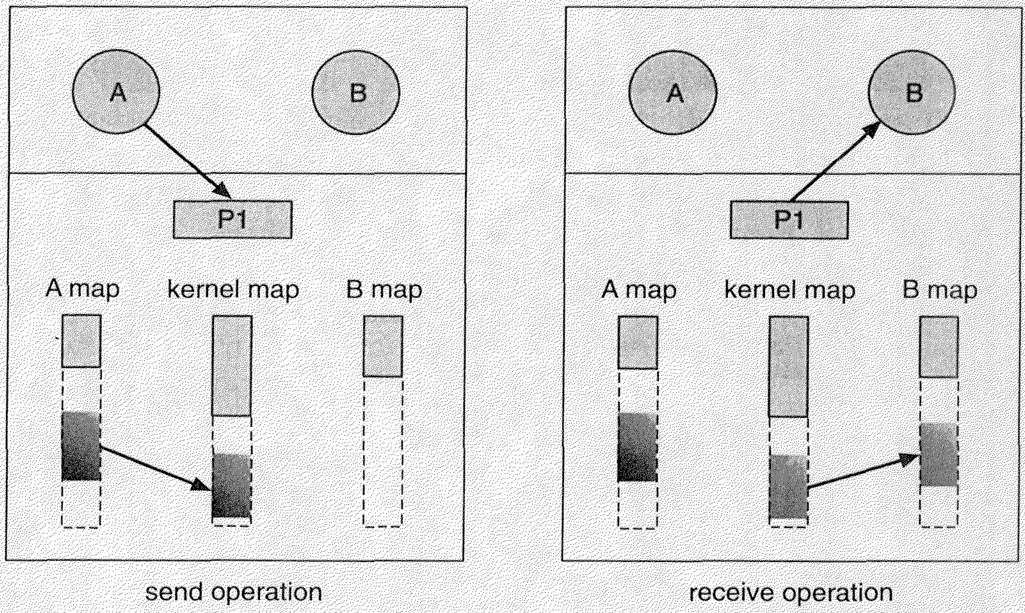


**Figure 20.6** Mach message transfer.

Mach makes no attempt to compress the address space, although a task may fail (crash) if it has no room for a requested region in its address space. Given that address spaces are 4 gigabytes or more, this limitation is not currently a problem. However, maintaining a regular page table for a 4 gigabyte address space for each task, especially one with holes in it, would use excessive amounts of memory (1 megabyte or more). The key to sparse address spaces is that page-table space is used for only currently allocated regions. When a page fault occurs, the kernel must check to see whether the page is in a valid region, rather than simply indexing into the page table and checking the entry. Although the resulting lookup is more complex, the benefits of reduced memory-storage requirements and simpler address-space maintenance make the approach worthwhile.

Mach also has system calls to support standard virtual-memory functionality, including the allocation, deallocation, and copying of virtual memory. When allocating a new virtual-memory object, the thread may provide an address for the object or may let the kernel choose the address. Physical memory is not allocated until pages in this object are accessed. The object's backing store is managed by the default pager (discussed in Section 20.6.2). Virtual-memory objects are also allocated automatically when a task receives a message containing out-of-line data.

Associated system calls return information about a memory object in a task's address space, change the access protection of the object, and specify how an object is to be passed to child tasks at the time of their creation (shared, copy-on-write, or not present).

### 20.6.2 User-Level Memory Managers

A secondary-storage object is usually mapped into the virtual address space of a task. Mach maintains a cache of memory-resident pages of all mapped objects, as in other virtual-memory implementations. However, a page fault occurring when a thread accesses a nonresident page is executed as a message to the object's port. The concept of a memory object being created and serviced by nonkernel tasks (unlike threads, for instance, which are created and maintained by only the kernel) is important. The end result is that, in the traditional sense, memory can be paged by user-written memory managers. When the object is destroyed, it is up to the memory manager to write back any changed pages to secondary storage. No assumptions are made by Mach about the content or importance of memory objects, so the memory objects are independent of the kernel.

There are several circumstances in which user-level memory managers are insufficient. For instance, a task allocating a new region of virtual memory might not have a memory manager assigned to that region, since it does not represent a secondary-storage object (but must be paged), or a memory manager could fail to perform pageout. Mach itself also needs a

In the current version, Mach does not allow external memory managers to affect the page-replacement algorithm directly. Mach does not export the memory-access information that would be needed for an external task to select the least recently used page, for instance. Methods of providing such information are currently under investigation. An external memory manager is still useful for a variety of reasons, however:

- It may reject the kernel's replacement victim if it knows of a better candidate (for instance, MRU page replacement).
- It may monitor the memory object it is backing, and request pages to be paged out before the memory usage invokes Mach's pageout daemon.
- It is especially important in maintaining consistency of secondary storage for threads on multiple processors, as we shall show in Section 20.6.3.
- It can control the order of operations on secondary storage, to enforce consistency constraints demanded by database management systems. For example, in transaction logging, transactions must be written to a log file on disk before they modify the database data.
- It can control mapped file access.

## 20.6.3 Shared Memory

Mach uses shared memory to reduce the complexity of various system facilities, as well as to provide these features in an efficient manner. Shared memory generally provides extremely fast interprocess communication, reduces overhead in file management, and helps to support multiprocessing and database management. Mach does not use shared memory for all these traditional shared-memory roles, however. For instance, all threads in a task share that task's memory, so no formal shared-memory facility is needed within a task. However, Mach must still provide traditional shared memory to support other operating-system constructs, such as the UNIX **fork** system call.

It is obviously difficult for tasks on multiple machines to share memory, and to maintain data consistency. Mach does not try to solve this problem directly; rather, it provides facilities to allow the problem to be solved. Mach supports consistent shared memory only when the memory is shared by tasks running on processors that share memory. A parent task is able to declare which regions of memory are to be *inherited* by its children, and which are to be readable–writable. This scheme is different from copy-on-write inheritance, in which each task maintains its own copy of any changed pages. A writable object is addressed from each task's address map, and all changes are made to the same copy. The threads within the tasks are responsible for coordinating changes to memory so that they do not interfere with one another (by writing to the same location concurrently). This coordination may be done through normal synchronization methods: critical sections or mutual-exclusion locks.

For the case of memory shared among separate machines, Mach allows the use of external memory managers. If a set of unrelated tasks wishes to share a section of memory, the tasks may use the same external memory manager and access the same secondary-storage areas through it. The implementor of this system would need to write the tasks and the external pager. This pager could be as simple or as complicated as needed. A simple implementation would allow no readers while a page was being written to. Any write attempt would cause the pager to invalidate the page in all tasks currently accessing it. The pager would then allow the write and would revalidate the readers with the new version of the page. The readers would simply wait on a page fault until the page again became available. Mach provides such a memory manager: the Network Memory Server (NetMemServer). For multicomputers, the NORMA configuration of Mach 3.0 provides similar support as a standard part of the kernel. This XMM subsystem allows multicomputer systems to use external memory managers that do not incorporate logic for dealing with multiple kernels; the XMM subsystem is responsible for maintaining data consistency among multiple kernels that share memory, and makes these kernels appear to be a single kernel to the memory manager. The XMM subsystem also implements virtual copy logic for the mapped objects that it manages. This virtual copy logic includes both copy-on-reference among multicomputer kernels, and sophisticated copy-on-write optimizations.

## 20.7 ■ Programmer Interface

There are several levels at which a programmer may work within Mach. There is, of course, the system-call level, which, in Mach 2.5, is equivalent to the 4.3BSD system-call interface. Version 2.5 includes most of 4.3BSD as one thread in the kernel. A BSD system call traps to the kernel and is serviced by this thread on behalf of caller, much as standard BSD would handle it. The emulation is not multithreaded, so it has limited efficiency.

Mach 3.0 has moved from the single-server model to support of multiple servers. It has therefore become a true microkernel without the full features normally found in a kernel. Rather, full functionality can be provided via emulation libraries, servers, or a combination of the two. In keeping with the definition of a microkernel, the emulation libraries and servers run outside the kernel at user level. In this way, multiple operating systems can run concurrently on one Mach 3.0 kernel.

An emulation library is a set of routines that lives in a read-only part of a program's address space. Any operating-system calls the program makes