

Second Edition

2

OPERATING SYSTEMS

A CONCEPT-BASED APPROACH



D M DHAMDHERE

Contents

<i>Preface</i>	xiii
1 Introduction	1
1.1 Abstract Views of an Operating System	1
1.2 Goals of an OS	5
1.3 Operation of an OS	7
1.4 Preview of the Book	16
<i>Exercises</i>	23
<i>Bibliography</i>	25

Part I

Fundamental Concepts

2 Overview of Operating Systems	31
2.1 OS and the Computer System	31
2.2 Efficiency, System Performance and User Convenience	46
2.3 Classes of Operating Systems	49
2.4 Batch Processing Systems	52
2.5 Multiprogramming Systems	56
2.6 Time Sharing Systems	65
2.7 Real Time Operating Systems	70
2.8 Distributed Operating Systems	73
2.9 Modern Operating Systems	75
<i>Exercises</i>	76
<i>Bibliography</i>	80
3 Processes and Threads	84
3.1 Processes and Programs	84
3.2 Programmer View of Processes	87
3.3 OS View of Processes	92
3.4 Threads	106
3.5 Case Studies of Processes and Threads	115
3.6 Interacting Processes—An Advanced Programmer View of Processes	130
<i>Exercises</i>	139
<i>Bibliography</i>	141
4 Scheduling	143
4.1 Preliminaries	143
4.2 Non-preemptive Scheduling Policies	148
4.3 Preemptive Scheduling Policies	151
4.4 Scheduling in Practice	156
4.5 Real Time Scheduling	168
4.6 Scheduling in Unix	175

viii Contents

<u>4.7 Scheduling in Linux</u>	178
<u>4.8 Scheduling in Windows</u>	181
<u>4.9 Performance Analysis of Scheduling Policies</u>	181
<u>Exercises</u>	186
<u>Bibliography</u>	188
5 Memory Management	191
<u>5.1 Managing the Memory Hierarchy</u>	191
<u>5.2 Static and Dynamic Memory Allocation</u>	193
<u>5.3 Memory Allocation to a Process</u>	196
<u>5.4 Reuse of Memory</u>	202
<u>5.5 Contiguous Memory Allocation</u>	211
<u>5.6 Noncontiguous Memory Allocation</u>	213
<u>5.7 Paging</u>	217
<u>5.8 Segmentation</u>	219
<u>5.9 Segmentation with Paging</u>	220
<u>5.10 Kernel Memory Allocation</u>	221
<u>5.11 A Review of Relocation, Linking and Program Forms</u>	226
<u>Exercises</u>	233
<u>Bibliography</u>	235
6 Virtual Memory	237
<u>6.1 Virtual Memory Basics</u>	237
<u>6.2 Demand Paging</u>	240
<u>6.3 Page Replacement Policies</u>	266
<u>6.4 Memory Allocation to a Process</u>	275
<u>6.5 Shared Pages</u>	279
<u>6.6 Memory Mapped Files</u>	282
<u>6.7 Unix Virtual Memory</u>	284
<u>6.8 Linux Virtual Memory</u>	289
<u>6.9 Virtual Memory in Windows</u>	290
<u>6.10 Virtual Memory Using Segmentation</u>	291
<u>Exercises</u>	295
<u>Bibliography</u>	299
7 File Systems	302
<u>7.1 File System and IOCS</u>	302
<u>7.2 Files and File Operations</u>	305
<u>7.3 Fundamental File Organizations</u>	307
<u>7.4 Directory Structures</u>	311
<u>7.5 File Protection</u>	318
<u>7.6 Interface between File System and IOCS</u>	319
<u>7.7 Allocation of Disk Space</u>	322
<u>7.8 Implementing File Access</u>	326
<u>7.9 File Sharing Semantics</u>	331
<u>7.10 File System Reliability</u>	333
<u>7.11 Virtual File System</u>	341
<u>7.12 Unix File System</u>	343

7.13 Linux File System	348
7.14 Windows File System	348
7.15 Performance of File Systems	350
<i>Exercises</i>	352
<i>Bibliography</i>	355
8 Security and Protection	357
8.1 Overview of Security and Protection	357
8.2 Goals of Security and Protection	360
8.3 Security Attacks	362
8.4 Formal and Practical Aspects of Security	364
8.5 Encryption	366
8.6 Authentication and Password Security	373
8.7 Access Descriptors and the Access Control Matrix	375
8.8 Protection Structures	377
8.9 Capabilities	382
8.10 Unix Security	390
8.11 Linux Security	391
8.12 Windows Security	392
<i>Exercises</i>	394
<i>Bibliography</i>	395

Part II Advanced Topics

9 Process Synchronization	403
9.1 Data Access Synchronization and Control Synchronization	403
9.2 Critical Sections	404
9.3 Race Conditions in Control Synchronization	409
9.4 Implementing Critical Sections and Indivisible Operations	411
9.5 Classic Process Synchronization Problems	414
9.6 Structure of Concurrent Systems	421
9.7 Algorithmic Approach Implementing Critical Sections	423
9.8 Semaphores	431
9.9 Conditional Critical Regions	443
9.10 Monitors	447
9.11 Process Synchronization in Unix	456
9.12 Process Synchronization in Linux	457
9.13 Process Synchronization in Windows	458
<i>Exercises</i>	459
<i>Bibliography</i>	463
10 Message Passing	466
10.1 Overview of Message Passing	466
10.2 Implementing Message Passing	470
10.3 Mailboxes	474
10.4 Message Passing in Unix	476

x Contents

10.5	Message Passing in Windows	480
	<i>Exercises</i>	481
	<i>Bibliography</i>	481
11	Deadlocks	483
11.1	Definition of deadlock	483
11.2	Deadlocks in Resource Allocation	484
11.3	Handling Deadlocks	490
11.4	Deadlock Detection and Resolution	491
11.5	Deadlock Prevention	496
11.6	Deadlock Avoidance	500
11.7	Formal Characterization of Resource Deadlocks	506
11.8	Deadlock Handling in Practice	513
	<i>Exercises</i>	515
	<i>Bibliography</i>	518
12	Implementation of File Operations	520
12.1	Layers of the Input Output Control System	520
12.2	Overview of I/O Organization	523
12.3	I/O Devices	525
12.4	Device Level I/O	537
12.5	Disk Scheduling	551
12.6	Buffering of Records	554
12.7	Blocking of Records	559
12.8	Access Methods	562
12.9	Unified Disk Cache	564
12.10	File Processing in Unix	565
12.11	File Processing in Linux	568
12.12	File Processing in Windows	570
	<i>Exercises</i>	571
	<i>Bibliography</i>	573
13	Synchronization and Scheduling in Multiprocessor Operating Systems	576
13.1	Architecture of Multiprocessor Systems	576
13.2	Multiprocessor Operating Systems	583
13.3	Kernel Structure	584
13.4	Process Synchronization	587
13.5	Process Scheduling	591
13.6	Case Studies	593
	<i>Exercises</i>	596
	<i>Bibliography</i>	597
14	Structure of Operating Systems	599
14.1	Operation of an OS	600
14.2	Structure of an Operating System	601
14.3	Operating Systems with Monolithic Structure	603
14.4	Layered Design of Operating Systems	604
14.5	Virtual Machine Operating Systems	607

14.6	Kernel Based Operating Systems	610
14.7	Microkernel Based Operating Systems	612
14.8	Configuring and Installing the Kernel	614
14.9	Architecture of Unix	615
14.10	The Kernel of Linux	617
14.11	Architecture of Windows	618
	<i>Exercises</i>	619
	<i>Bibliography</i>	619

Part III

Distributed Operating Systems

15	Distributed Operating Systems	625
15.1	Features of Distributed Systems	625
15.2	Nodes of a Distributed System	627
15.3	Network Operating Systems	627
15.4	Distributed Operating Systems	628
15.5	Reliable Interprocess Communication	631
15.6	Distributed Computation Paradigms	637
15.7	Networking	644
15.8	Model of a Distributed System	657
15.9	Design Issues in Distributed Operating Systems	659
	<i>Exercises</i>	661
	<i>Bibliography</i>	662
16	Theoretical Issues in Distributed Systems	664
16.1	Notions of Time and State	665
16.2	States and Events in a Distributed System	665
16.3	Time, Clocks and Event Precedences	666
16.4	Recording the State of a Distributed System	672
	<i>Exercises</i>	680
	<i>Bibliography</i>	682
17	Distributed Control Algorithms	684
17.1	Operation of Distributed Control Algorithms	684
17.2	Correctness of Distributed Control Algorithms	686
17.3	Distributed Mutual Exclusion	688
17.4	Distributed Deadlock Handling	693
17.5	Distributed Scheduling Algorithms	698
17.6	Distributed Termination Detection	701
17.7	Election Algorithms	703
17.8	Practical Issues in Using Distributed Control Algorithms	705
	<i>Exercises</i>	707
	<i>Bibliography</i>	708
18	Recovery and Fault Tolerance	710
18.1	Faults, Failures and Recovery	711
18.2	Byzantine Faults and Agreement Protocols	714

xii Contents

<u>18.3 Recovery</u>	715
<u>18.4 Fault Tolerance Techniques</u>	717
<u>18.5 Resiliency</u>	721
<i>Exercises</i>	722
<i>Bibliography</i>	723
19 Distributed File Systems	725
<u>19.1 Design Issues in Distributed File Systems</u>	725
<u>19.2 Transparency</u>	728
<u>19.3 Semantics of File Sharing</u>	729
<u>19.4 Fault Tolerance</u>	730
<u>19.5 DFS Performance</u>	735
<u>19.6 Case Studies</u>	740
<i>Exercises</i>	747
<i>Bibliography</i>	748
20 Distributed System Security	750
<u>20.1 Issues in Distributed System Security</u>	750
<u>20.2 Message Security</u>	753
<u>20.3 Authentication of Data and Messages</u>	760
<u>20.4 Third-Party Authentication</u>	762
<i>Exercises</i>	768
<i>Bibliography</i>	769
Index	770

Preface

The primary objective of a first course in Operating Systems is to develop an understanding of the fundamental concepts and techniques of operating systems. Today, a large section of students is already exposed to diverse information on operating systems due to practical exposure to operating systems and literature on the Internet; such students have a lot of information but few concepts about operating systems. This situation makes teaching of operating systems concepts a challenging task because it is necessary to retrofit some concepts to the information presented by these students without boring them, yet do it in a manner that introduces concepts to first time learners of operating systems without intimidating them. This book presents operating system concepts and techniques in a manner that incorporates these requirements.

General Approach

The book begins by building a core knowledge of what makes an operating system tick in Chapter 2. It presents an operating system as an intermediary between a computer system and user computations whose task is to provide good service to users and achieve efficient use of the computer system. A discussion of interaction of an operating system with the computer system on one hand and with user computations on the other hand consolidates this view and adds practical details to it. This approach has the effect of demystifying an operating system for a new reader, and relating to the background of an experienced reader. It also emphasizes key features of computer architecture which are essential for a study of operating systems. This part of the book provides a basis for discussing details of functions performed by an operating system in more detail.

The rest of the book follows an analogous approach. Each chapter identifies fundamental concepts involved in some functionality of an operating system, describes relevant features in computer architecture, discusses relevant operating system techniques and illustrates their operation through examples. The highlights of this approach are:

- Fundamental concepts are introduced in simple terms.
- Numerous examples are included to illustrate concepts and techniques.
- Implementation details and case studies are organized as small capsules spread throughout the text.
- Optional sections are devoted to advanced topics, e.g. deadlock characterization, kernel memory allocation, synchronization and scheduling in multiprocessor systems, file sharing semantics, file system reliability and capabilities.

The key advantage of this approach is that concepts, techniques and case studies are integrated into cohesive chapters. Hence many design and implementation details look ‘obvious’ when the reader encounters them. This fact helps to emphasize that the study of operating systems must be based on a sound understanding of concepts. This is the most important message a text can give to a student of operating systems who will face both a rich diversity and rapid changes in features of operating systems during his/her career.

Pedagogic Features

Part I

FUNDAMENTAL CONCEPTS

An operating system controls use of a computer system's resources such as CPUs, memory, and I/O devices to meet computational requirements of its users. Users expect convenience, quality of service, and security while executing their programs, whereas system administrators expect efficient use of the computer's resources and good performance in executing user programs. These diverse expectations can be characterized as user convenience, orderly and efficient use of resources; they form the primary goals of an operating system. The extent to which an operating system succeeds in each of these goals depends on its computing environment, i.e., the computer system's hardware, its interfaces with other computers, and the nature of computations performed by its users.

Part Introduction

Each part of the book begins with a description of its contents, and a road map of the chapters in the part.



Fig. 1.7 Fundamental functions in control processes

Sequence of single programs: Each single program in the sequence is initiated by the user through a separate command. However, a sequence of single programs has its own semantics—a program should be executed only if the previous program in the sequence executed successfully. To achieve a common goal, the programs must explicitly interface their inputs and outputs with other programs in the sequence. Example 1.1 discusses the notion of a sequence of single programs in MS-DOS.

Example 1.1: An MS-DOS .bat file can be used to form a sequence of single programs. The file contains a sequence of commands, each command indicating the name of a program. Let a .bat file contain commands to initiate a sequence of programs to compile, link and execute a C program. The C compiler compiles a C program existing in the algor1.c, linker links the output of the compiler to generate an executable program named dmc, and program dmc executes on the data contained in file sample. The command interpreter loads the C compiler into the memory for execution. When the compiler completes its operation, the command interpreter initiates execution of the linker. Eventually, it initiates execution of dmc.

Figures and Boxes

Figures depict practical arrangements used to handle user computations and use of resources, stepwise operation of specific techniques, or comparison of alternative techniques to project their strengths and weaknesses. Boxes are used to enclose key features of operating system functionalities being discussed. They also serve as overviews or summaries of specific topics.

Examples

Examples demonstrate the key issues concerning concepts and techniques being discussed. Examples are typeset in a different style to set them apart from the main body of the text. This feature helps a reader to skip an example if he/she does not want the flow of ideas to be interrupted, especially while reading a chapter for the first time.

Program Code

Program code is presented in an easy to understand pseudo-code form.

```

state : integer value { ... }
buffer : array[1..100] of ...
copy-example : copy_struct; knownothing : string;
useOfExample : integer;
/* Copy processes */
copy-example := CreateProcess(hFile, hFileBuffer, 0, 0, 0, 0,
    &knownothing, &useOfExample, 0, 0);
/* Set more information to copy example and disk write */
useOfExample := 1;
SetCopyExample(hFile, useOfExample);
useOfDiskWrite := 1;
SetDiskWrite(hFile, useOfDiskWrite);
/* Check state of all processes */
useOfDiskWrite := 0;
while (true == true) {
    if (useOfCopyExample == knownUsedAndUseOfDiskWrite) {
        knownUsedAndUseOfDiskWrite := knownUsedAndUseOfDiskWrite + 1;
        break;
    }
}

```

Exercises

Exercises are included at the end of each chapter. These include numerical problems based on material covered in the text, as well as challenging conceptual questions which test understanding of the concepts and techniques covered in a chapter and also provide deeper insights.

14. Operating Systems

EXERCISE 1

- Give examples of two situations in which user concurrency conflicts with efficient use of a computer system.
- Resource preemption may be performed in prioritized fashion in the use of resources. Can more efficient use be achieved in a situation in which resource preemption provides fairness in the use of resources? Read Chapter 2 and describe a situation in which preemption is used to obtain efficiency gain.
- An OS designer makes the following policy statement: "Consider preemption of a resource from a program only if the program can resume its operation after the resource is regranted to it even if the preemption had not occurred." Justify preemption of CPU and memory from a program. Also argue why preemption of a magnetic tape cartridge or printer is undesirable.
- In the Unix operating system, a print server implements printing of files as defined by users. When a user executes a command to print a file, the print server copies the file on the disk and enters it in a print queue. It is printed when an empty reaches the head of the print queue.

Chapter introduction The chapter introduction precedes the first section of each chapter. It describes the objective and importance of the chapter and describes the topics covered in the chapter.

Snapshots of concurrent systems Students have difficulty visualizing concurrent operation of processes in a software system. This difficulty leads to an inadequate understanding of process synchronization. A snapshot depicts the state of different processes and the synchronization data to provide a holistic view of activities in a concurrent system.

Optional sections Optional sections are devoted to advanced topics. These sections discuss theoretical basis for certain topics or advanced implementation issues. An instructor may include these sections in an offering of an operating system course depending on the availability of time, or selectively assign them for self-study.

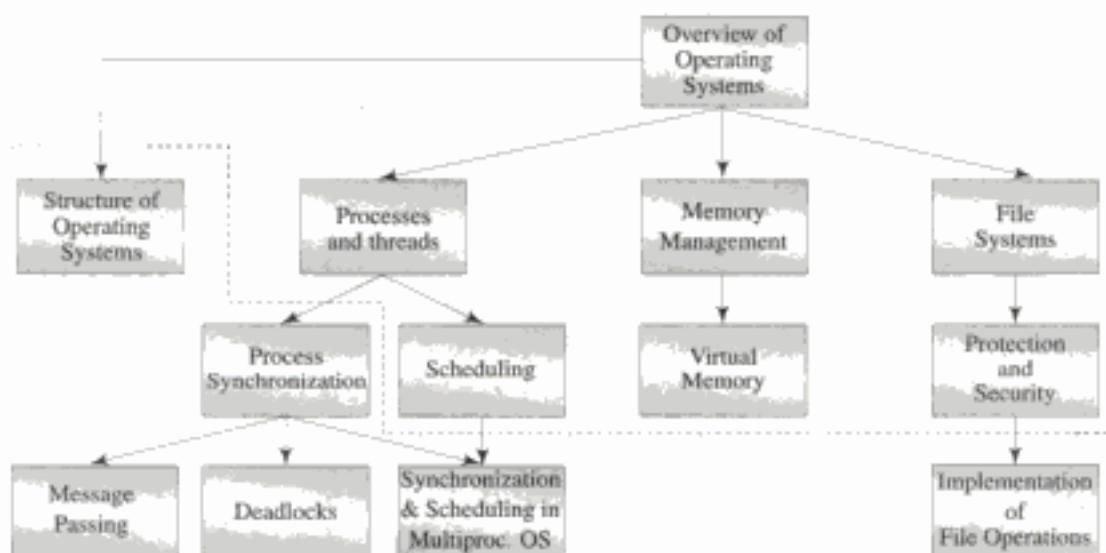
Case studies Case studies are organized as independent sections. They illustrate practical issues, arrangements and tradeoffs in the design and implementation of an operating system. Different versions of the Unix and Windows operating systems, and Linux are the primary operating systems covered by the case studies.

Instructor resources A detailed solutions manual and slides for classroom usage are planned and will be available at <http://www.mhhe.com/dhamdhere/os>

Organization of the book

The book deals with both conventional operating systems for use in independent computer systems, and distributed operating systems for use in computer systems that are formed by networking independent computers. The introduction discusses the fundamental concerns of an operating system and describes different kinds of computations and different methods of ensuring efficient use of resources. The discussion of conventional operating systems is organized into two parts, devoted to fundamental concepts and advanced topics, respectively. The discussion of distributed operating systems forms the third part by itself. The structure of the parts and interdependency between chapters is as follows:

Part I: Fundamental Concepts



Part II: Advanced Topics

Part I: Fundamental Concepts The first part consists of seven chapters. Chapter 2 focuses on interaction of an operating system with a computer system and with user computations. It also describes different classes of operating systems and describes the fundamental concepts and techniques used by them.

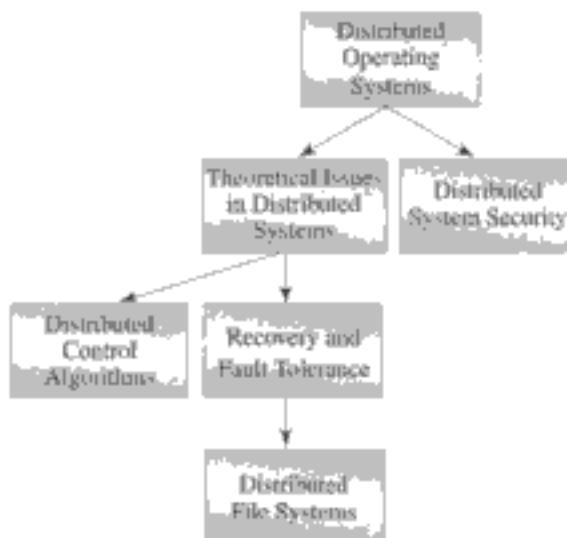
An operating system uses the concepts of *process* and *thread* to manage execution of programs—informally, both a process and a thread represent an execution of a program; we use *process* as a generic term. Chapter 3 describes the user and operating system views of processes, i.e., how processes are created, how they interact with one another, and how they are controlled by the operating system. An operating system overlaps execution of processes to provide good user service and ensure efficient use of resources. Chapter 4 describes the scheduling techniques used for this purpose.

The operating system shares the computer's memory between processes. Chapter 5 deals with the key issue of memory fragmentation, which is a situation in which an area of memory cannot be used because it is too small, and techniques which address memory fragmentation. It also discusses the techniques used by the kernel to manage its own data structures. Chapter 6 discusses implementation of virtual memory, which enables execution of a program whose size exceeds the size of memory.

The next two chapters deal with the File system, and security and protection of data and programs stored in files. Chapter 7 describes facilities for creation, access, sharing and reliable storage of files. Chapter 8 discusses how files are protected against illegal forms of access by users.

Part II: Advanced Concepts This part consists of six chapters devoted to advanced topics dealing with processes and their operation, implementation of file systems, and the structure of operating systems. Chapter 9 deals with process synchronization techniques that enable processes of an application to share common data or coordinate their activities to achieve a common goal, while Chapter 10 discusses how processes exchange messages to pass information to one another. Chapter 11 discusses deadlock, which is a situation in which processes wait for each other indefinitely, thereby, stalling an application. Chapter 12 describes how an OS manages I/O devices and describes techniques it uses to achieve efficient implementation of file operations. Chapter 13 deals with synchronization and scheduling in a multiprocessor operating system. Chapter 14 discusses operating system design techniques that facilitate easy modification of an operating system (1) for use on a computer system with a different architecture, or (2) to meet new requirements of its users.

Part III: Distributed Operating Systems A distributed operating system differs from a conventional one in that the resources, processes and control operations of the OS are distributed between different nodes. This difference gives rise to a host of issues concerning reliability, efficiency, consistency and security of computations and of the OS itself. This Part contains six small chapters that address these issues.



Part III: Distributed Operating Systems

Using this book

This book is intended as a text for a course on operating systems that is modeled after the IEEE and ACM curricula in Computer Science. The first part of the book covers the fundamental OS principles, so it is adequate for a light course on operating system principles by itself. The first two parts of the book and selected sections of Chapters 15–20 of the third part are together adequate for an operating system principles course in a Computer Science or related curriculum. However, all topics discussed here cannot be covered in a semester, so an instructor may wish to omit some of the sections on advanced topics or the chapters on message passing, synchronization and scheduling in multiprocessor operating systems, and structure of operating systems; or assign them for self-study. The three parts of the book together cover the topics needed for a two-course sequence on operating system principles. A course on Distributed operating systems, possibly at the postgraduate level, could use the third part of the book for a substantial portion of the course.

Apart from an introduction to computing, this book does not assume any specific background. Hence instructors and students are likely to find that it contains a lot of introductory material which students already know. I have included this material for one very important reason: As mentioned at the start of the preface, students know many *things* on their own, but often lack *concepts*. So it is useful for students to read even familiar material which is presented in a concept-based manner. For the same reason, I consider it essential for instructors to cover most of Chapter 2, and particularly the following topics, in class:

- Section 2.1: OS and the Computer System, particularly I/O, interrupt, and memory protection hardware; and system calls.
- Section 2.5: Multiprogramming, particularly program mix and priority.

Differences with the first edition

The book has been restructured substantially. Chapters in a part and their sequences are different from those in the first edition. Chapters and their sections have been reorganized, new topics have been added, and tables have been introduced to achieve better focus on concepts. A section containing a preview of the book has been added in the first chapter, and introductions have been added to parts and chapters to better motivate the reader. I hope students and instructors like the new format of the book.

D M DHAMDHHERE

Introduction

An operating system (OS) is different things to different users. Each user's view is called an *abstract view* because it emphasizes features that are important from the viewer's perspective, ignoring all other features. An operating system implements an abstract view by acting as an intermediary between the user and the computer system. This arrangement not only permits an operating system to provide several functionalities at the same time, but also to change and evolve with time. We discuss how abstract views are also useful while designing an operating system, and during its operation.

An operating system has two goals—efficient use of a computer system and user convenience. Unfortunately, user convenience often conflicts with efficient use of a computer system. Consequently, an operating system cannot provide both. It typically strikes a balance between the two that is most effective in the environment in which a computer system is used—efficient use is important when a computer system is shared by several users while user convenience is important in personal computers. We use the term *effective utilization* for the balance between efficiency and user convenience that best suits an environment. We discuss facets of efficient use and user convenience in this chapter.

The primary concern of an OS is to support execution of user programs to ensure user convenience and efficient use of resources. In this chapter, we describe the functions performed by it to address these concerns.

1.1 ABSTRACT VIEWS OF AN OPERATING SYSTEM

A question like ‘What is an OS?’ is likely to evoke different answers. For example,

- For a young school or college student, an OS is the software that permits access to the wealth of knowledge available on the Internet.
- For a programmer, an OS is the software that permits the use of a computer system for program development.

- For a person using an application package, an OS is simply the software that makes it possible for him to use the package.
- For a technician in a computerized chemical plant, the OS is the invisible component of a computer system that controls the plant.

The answers differ because a user's perception about an OS depends on three factors: the purpose for which a computer is being used, the *computing environment*, i.e., the environment in which the computer system is used, and the degree of identity of the computer system with the purpose being served.

For the student, the sole purpose of the computer system might be to use an Internet browser. The OS helps in achieving this, so it is identified with use of the computer system for Internet browsing. A programmer wishes to use a computer system for general purpose program development, so the ability to use a compiler for a specific language is of paramount importance. For a person using an application package, the OS is simply a means to the use of the package. Such a user is oblivious to other capabilities of an OS. A technician in a computerized chemical plant similarly views the OS as everything in the control system of the plant. An OS designer will doubtless have a different perception of what is an OS.

A common thread in all these perceptions is the utilization of a computer system for a specific purpose or a set of purposes. The purposes in the above examples are different, so perceptions of the operating system's role are also different.

From these examples it is clear that a user perceives an OS as the software that helps him in achieving an intended use of a computer system—other capabilities of the computer system and its environment do not seem to matter. Thus, the OS is simply a means of achieving an intended purpose. A user will prefer an OS that permits him to achieve his purpose in the simplest and fastest possible manner. Thus, a user's perspective is always one of convenience and speed in computer utilization.

None of the four views of an OS described at the start of this Section (and illustrated in Figure 1.1) is complete, but each view is correct and useful because it helps a user to understand how to use an OS. Such views are called *abstract views*. An abstract view focuses on essential characteristics of an entity from the perspective of a viewer. Consequently, an abstract view contains some elements of reality but ignores other elements.

1.1.1 Uses of Abstract Views

A user's abstract view contains important features of a system from a user's perspective. It helps an OS designer to understand the requirements of a user, which helps in planning the features of an OS. Abstract views are useful for other purposes summarized in Table 1.1. The key advantage of using an abstract view in design is that it helps to control complexity of the design by separating the concerns of different parts of a system. Abstract views are also useful for understanding the design and implementation of a system.

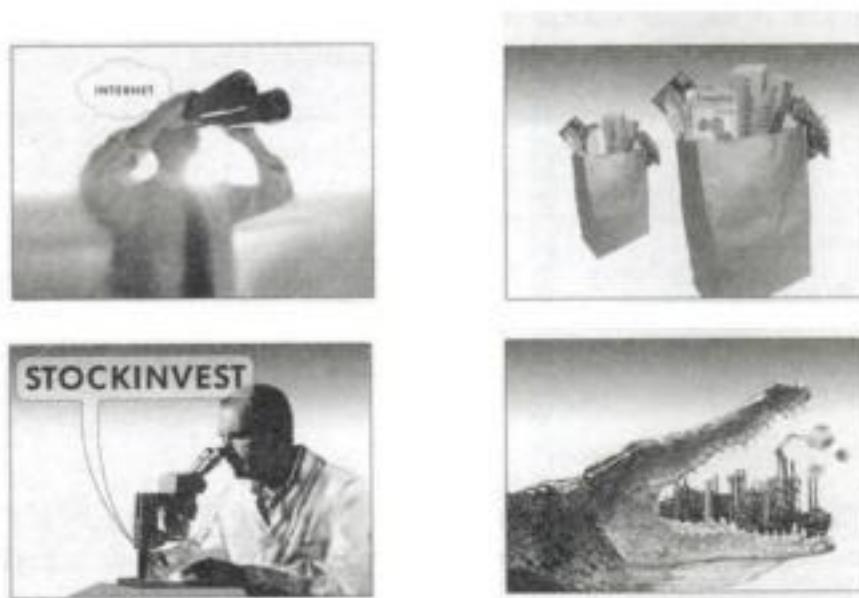


Fig. 1.1 Abstract views of an OS by a student, programmer, user and technician

Table 1.1 Uses of abstract views

Use	Description
Gathering system requirements	A user's abstract view indicates important services which a system should provide. A collection of abstract views can be used to compose a specification of system requirements.
Design of a system	Use of abstract views permits a designer to focus on a specific part of a system. Details of other parts are 'hidden'; these parts can be assumed to be available. This approach helps to control complexity of the design process.
Implementation of a system	A part whose details were hidden in a design view becomes a module, which can be called from other modules. This fact leads to a modular implementation.

Figure 1.2 contains an abstract view of the structure of an OS, which shows three main parts of an OS. Each part consists of a number of programs. The kernel is the core of the OS. It controls operation of the computer and provides a set of functions and services to use the CPU and resources of the computer. Non-kernel programs implement user commands. These programs do not interact with the hardware, they use facilities provided by kernel programs. Programs in the user interface part either provide a command line interface or a graphical user interface (GUI) to the user. These programs use facilities provided by non-kernel programs. A user interacts with

programs in the user interface—typically with the command interpreter—to request use of resources and services provided by the system.

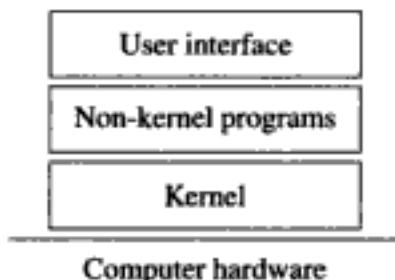


Fig. 1.2 A designer's abstract view of an OS

The abstract view of Figure 1.2 has interesting properties. It contains a hierarchical arrangement of program layers in which programs in a higher layer use the facilities provided by programs in the layer below it. In fact, each layer takes an abstract view of the layer below it. In this view the lower layer appears as a system that is capable of executing certain commands. It could even be a machine that is capable of performing certain operations. The fact that the lower layer consists of a set of programs rather than a computer system makes no difference to the higher layer. Each layer extends capabilities of the machine provided by the lower layer. The machine provided by the user interface layer understands the commands in the command language of the OS. This abstract view helps to understand the design of an OS.

Logical and physical organization A programmer's view of an entity is typically an abstract view. It contains features and useful properties of an entity from a programmer's perspective. Three important entities of this kind are a program, a file and an I/O device. The abstract view of an entity is called the *logical view* and the arrangement and relationship between components of the entity is called the *logical organization*. The real view of an entity, which often coincides with the operating system's view of the entity, is called the *physical view* and the arrangement depicted in it is called the *physical organization*.

Consider the execution of a program P in a computer system. The logical view of the program P is shown in Figure 1.3(a). It shows the code of P written in a higher level language, and the data that it reads during its execution. The physical view of P's execution is shown in Figure 1.3(b). Program P has been compiled into the machine language of the computer system. In this form, the program consists of instructions that the computer can understand and execute, and data held in the computer's memory. The rectangle represents the memory of the computer system. The code of P occupies only one part of the computer's memory. Several other programs also exist in the computer's memory. The data of P is recorded in a file named *info*. This file is stored on a disk. During its execution, P reads some data

from `info`, manipulates it, and prints its results. This view includes the disk on which `info` is recorded and the printer on which P's results are to be printed, the memory of the computer system and the CPU to execute P's instructions.

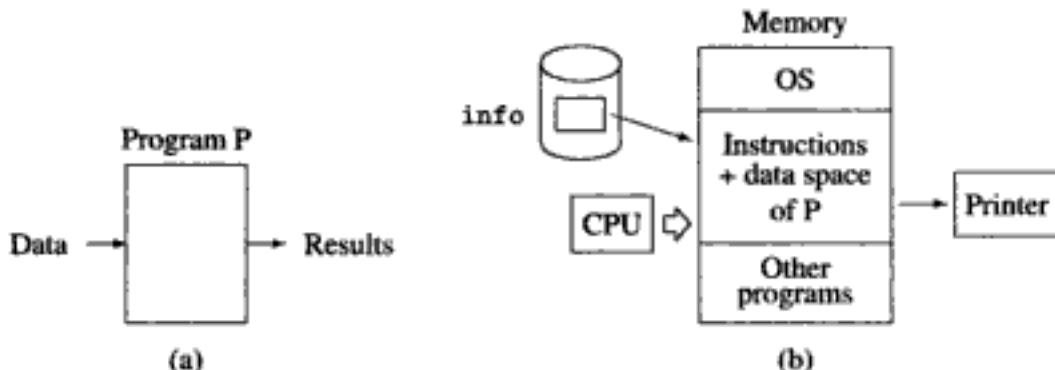


Fig. 1.3 Logical and physical views of execution of a program

In Section 1.3.2.1, we discuss the logical and physical views of I/O devices. We will discuss logical and physical organizations of files in Chapter 7.

1.2 GOALS OF AN OS

An operating system must not only ensure efficient use of a computer system, but also provide user convenience. However, these considerations often conflict. For example, providing fast service to one user's request could mean that other users of the system have to be neglected. It could also mean that resources should remain allocated to a user's program even when the program is not using them, which would lead to under-utilization of resources. In such cases, the OS designer must make a conscious decision to trade off a part of convenience and speed for one user with that of other users or with efficient use of the computer system. Accordingly, the key goal of an operating system is to provide a combination of efficient use and user convenience that best suits the environment in which it is used. This is the notion of *effective utilization* of a computer system.

We find a large number of operating systems in use because each one of them provides a different flavor of effective utilization. At one extreme we have OSs that provide fast service required by command and control applications, while at the other we have OSs that make efficient use of computer resources to provide low-cost computing. In the middle, we have OSs that provide different combinations fast service and efficient use. In the following, we discuss several aspects of efficient use and user convenience to understand the notion of effective utilization.

OS and effective utilization of computer systems The notion of effective utilization spans a broad spectrum of considerations. At one end of the spectrum are user-centric considerations, such as user convenience and fast service to user requests. They are important in operating systems that support interactive computing or time critical

applications. System-centric considerations exist at the other end of the spectrum. Efficient use of the system is the paramount concern in a system-centric computing environment such as non-interactive data processing. It is achieved through use of good resource allocation policies.

Efficiency of use has two aspects. An OS consumes some resources of a computer system during its own operation, for example, it occupies memory and uses the CPU. This consumption of resources constitutes an *overhead* that reduces the resources available to user programs.

The other aspect of efficient use concerns use of resources by user programs. Poor efficiency can result from two causes—if an OS over-allocates resources to programs, or if it is unable to allocate free resources to programs that need them. The former leads to wastage of resources, while the latter leads to idling of resources and affects progress of programs. To achieve good efficiency, an OS must counter both these effects and also minimize its overhead.

Efficient use Efficient use of resources can be obtained by monitoring the use of resources and performing corrective actions when necessary. However, a computer system contains several resources like memory, I/O devices and the CPU, so a comprehensive method of monitoring efficiency may lead to high overhead. Consequently, operating systems employ simple and easy to apply but known-to-be-suboptimal strategies for ensuring good efficiency, e.g., they either focus on efficiency of a few important resources like the CPU and memory, or handle user programs in a manner that guarantees high efficiency. The latter alternative avoids computation of efficiency, which is a recurrent overhead. Batch processing and multiprogramming, two early OSs using efficiency of use as a design aim, used this approach (see Chapter 2).

User convenience User convenience has several facets as summarized in Table 1.2. In the early days of computing, computer systems were expensive, so operating systems emphasized efficient use. Consequently, user convenience was synonymous with bare necessity—the mere ability to execute a program written in a higher level language was considered adequate. Experience with early OSs led to demands for better service, where the notion of service was limited to fast response to a user request.

Other facets of user convenience can be linked with the use of computers and computerized applications in new fields. Early operating systems had complex user interfaces, whose use required substantial user training. This was acceptable because most users were scientists and engineers. Easy-to-use interfaces had to be developed to facilitate computer usage by new classes of users. User friendly operating systems achieved this effect. In many ways, this move can be compared to the spread of car driving skills in the first half of the twentieth century. Over a period of time, driving became less of an expertise and more of a skill that could be acquired with limited training and experience.

Table 1.2 Facets of User Convenience

Facet	Examples
Necessity	Ability to execute programs, use the file system
Good Service	Speedy response to computational requests
User friendly OS	Easy-to-use commands, Graphical user interface (GUI)
New programming model	Concurrent programming
Features for experts	Means to set up complex computational structures
Web-oriented features	Means to set up Web-enabled servers
Evolution	Ability to add new features, use new computers

Computer users attacked new problems as computing power increased. New models were proposed for developing cost-effective solutions to new types of problems. Some of these models could be supported by the compiler technology and required little specific support from the OS. Modular and object oriented program design are two such models. Other models like the concurrent programming model required specific support features in the OS. User friendly user interfaces were considered boring by professional programmers, hence specialized interfaces were developed for expert users.

Yet another class of OS features was motivated by the advent of the Internet. Effective use of the web requires the ability to set-up servers that are usable over the web. This feature requires extensive networking support and an ability to scale up the performance of a server or shut it down depending on the load directed at it.

Finally, users expect their operating system to evolve with time to support new features as new application areas and new computer technologies develop.

1.3 OPERATION OF AN OS

An operating system implements computational requirements of its users with the help of resources of the computer system. Its key concerns are described in Table 1.3.

Table 1.3 Key concerns of an operating system

Concern	OS responsibility
Programs	Initiation and termination of programs. Providing convenient methods so that several programs can work towards a common goal.
Resources	Ensuring availability of resources in the system and allocating them to programs.
Scheduling	Deciding when, and for how long, to devote the CPU to a program.
Protection	Protect data and programs against interference from other users and their programs.

To realize execution of a program, an operating system must ensure that resources

like memory and I/O devices are available to it. It must also provide sufficient CPU attention to a program's execution. This function is called *scheduling*. A user may employ several programs to fulfill a computational requirement, so an OS must provide means to ensure harmonious working of such programs. To win the trust of users, an OS must provide a guarantee that their data would not be illegally used or lost, and that execution of their programs would not face interference from other programs in the system.

An operating system performs several housekeeping tasks to support the functions of creation and termination of programs, resource allocation, scheduling and protection. An operating system also performs other tasks to implement its notion of effective utilization. Table 1.4 describes commonly performed OS tasks.

Table 1.4 Common tasks performed by operating systems

Task	When/who performs
1. Maintain a list of authorized users	System administrator
2. Construct a list of all resources in the system	When OS is started
3. Initiate execution of programs	At user commands
4. Maintain resource usage information by programs and current status of all programs	Continuously during OS operation
5. Maintain current status of all resources and allocate resources to programs when requested	At resource request or release
6. Perform scheduling	During OS operation
7. Maintain information for protection	During OS operation
8. Handle requests made by users and their programs	At user requests

The list of persons who are authorized to use the computer system is maintained by the system administrator of the OS. The administrator adds and deletes names to this list in accordance with the computer usage policy. When the computer system is switched on, it invokes a procedure called *booting*, which performs a number of preparatory actions and starts operation of an OS. One of these actions is to make a list of all resources in the system. The OS initiates a new program when a user issues a command to execute it. The OS keeps track of resource usage by programs as this information might be needed to implement its notion of effective utilization, e.g., to ensure fairness in the use of resources. The OS also maintains information about current status of resources and programs for use while performing resource allocation and scheduling. The following sections discuss preliminaries of how an OS manages programs, resources and scheduling.

1.3.1 Programs

A *computational structure* is a configuration of one or more programs that work towards a common goal. It is created by issuing one or more commands specifying relationships between programs and to initiate their execution. Some typical compu-

tational structures are:

- A single program
- A sequence of single programs
- Co-executing programs.

These computational structures will be defined and described in later chapters; only their salient features are described here to illustrate user convenience provided by them. Table 1.5 summarizes OS responsibilities for these computational structures.

Table 1.5 Computational structures and OS responsibilities

Computational structure	OS responsibilities
Single program	Perform program initiation/termination, resource management
Sequence of single programs	Implement program dependence—terminate the sequence if a program faces abnormal termination
Co-executing programs	Provide appropriate interfaces between programs, perform termination of the co-executing programs

Single program A single program computation consists of the execution of a program on a given set of data. The program can be either sequential or concurrent. A single program is the simplest computational structure; it matches with the conventional notion of a program. In a concurrent program, different parts of the program can execute concurrently. The OS needs to know the identities of these parts to organize their concurrent execution. This function is *not* served by the user interface of the OS; Chapter 9 describes how it is implemented. In this section each program is assumed to be sequential in nature.

Sequence of single programs Each single program in the sequence is initiated by the user through a separate command. However, a sequence of single programs has its own semantics—a program should be executed only if the previous programs in the sequence executed successfully. To achieve a common goal, the programs must explicitly interface their inputs and outputs with other programs in the sequence. Example 1.1 discusses the notion of a sequence of single programs in MS DOS.

Example 1.1 An MS DOS .bat file can be used to form a sequence of single programs. The file contains a sequence of commands, each command indicating execution of a program. Let a .bat file contain commands to initiate a sequence of programs to compile, link and execute a C program: The C compiler compiles a C program existing in file `alpha.c`, linker links the output of the compiler to generate an executable program named `demo`, and program `demo` executes on the data contained in file `sample`. The command interpreter loads the C compiler into the memory for execution. When the compiler completes its operation, the command interpreter initiates execution of the linker. Eventually, it initiates execution of `demo`.

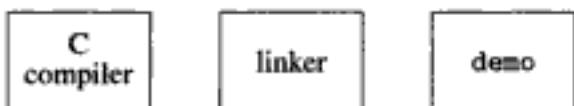


Fig. 1.4 Steps in execution of a sequence of programs

Figure 1.4 illustrates the three computations. Note that each program in the sequence must make its own arrangements to communicate its results to other programs. This is typically achieved by using some file naming conventions, e.g., the output of the C compiler may be put into a file named `alpha.obj` so that `linker` can pick it up from there. The command interpreter is unaware of these arrangements; it simply implements the commands issued to it.

Semantics of a sequence of programs dictate that `linker` would be executed only if the compilation of the C program was successful, and that `demo` would be executed only if linking was successful. This would not have been the case if the same three programs were initiated as three single programs.

Co-executing programs A user initiates co-executing programs by indicating their names in a single command. Co-execution semantics require that the programs should execute at the same time, rather than one after another as in a sequence of programs, and interact with one another during their execution. The nature of interaction is specific to a co-execution command. The OS provides appropriate interfaces between the co-executing programs. Example 1.2 discusses co-execution of programs in Unix.

Example 1.2 The Unix command

```
cat names | sort | uniq | wc -l
```

initiates execution of four co-executing programs to count the number of unique names in file `names`. These programs are—`cat`, `sort`, `uniq` and `wc`. '`|`' is the symbol for a Unix pipe, which sends the output of one program as input to another program. Thus, the output of `cat` is given to `sort` as its input, the output of `sort` is given to `uniq` and the output of `uniq` is given to `wc`. `cat` reads contents of file `names` and writes each name it finds there into its standard output file. The output of `cat` is the input to `sort`, so `sort` reads these names, sorts them in alphabetical order, and writes them into its output file. `uniq` removes duplicate names appearing in its input and outputs the unique names. `wc` counts and reports the number of names in its input. (`-l` is an option to the `wc` command asking it to count the number of lines in its input since `uniq` puts each name in a line by itself.)

Note that the four programs do not form a sequence of programs. The programs co-exist in the memory and co-operate with one another during their execution (see Figure 1.5). Unlike in a sequence of programs, passing of one program's results to the next program is the responsibility of the OS. The command processor also has to provide appropriate synchronization between the programs to ensure that a program consumes some data only after it has been produced. All co-executing programs terminate together. Control then returns to the command processor.

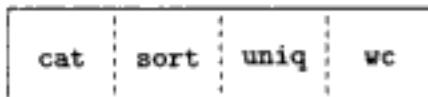


Fig. 1.5 Co-executing programs

1.3.2 Resource Allocation and Scheduling

The resource allocation function performs *binding* of one or more resources with a requesting program—that is, it associates resources with a program. It also deallocates resources from a program and allocates them to other programs. Allocation and deallocation of resources in this manner helps to implement resource sharing by users. Resources can be divided into system resources and user-created resources.

The resource protection function prevents mutual interference between users sharing a set of resources. Protection is implemented by performing an allocation only if a request satisfies a set of constraints. Some of these constraints relate to the nature of a resource, e.g., whether a resource can be used concurrently by several users. Other constraints are specified by the owner of a resource. These constraints typically embody the notion of access privileges and specify whether a specific user should or should not be allowed to access a resource.

Two popular strategies for resource allocation are:

- Partitioning of resources
- Allocation from a pool.

In the resource partitioning approach, the OS decides *a priori* what resources should be allocated to a user program. This approach is called *static allocation* because the allocation is made *before* the execution of a program begins. Static resource allocation is simple to implement. However, it lacks flexibility that leads to problems like wastage of resources that are allocated to a program but remain unused, and inability of the OS to grant additional resources to a program during its execution. These difficulties arise because allocation is made on the basis of perceived needs of a program, rather than its actual needs.

In the pool-based approach the OS maintains a common pool of resources and allocates from this pool whenever a program requests a resource. This approach is called *dynamic allocation* because allocation takes place *during* execution of a program. It avoids wastage of allocated resources, hence it can provide better resource utilization.

A simple resource allocation scheme uses a resource table as the central data structure (see Table 1.6). Each entry in the table contains the name and address of a resource unit and its present status, i.e., whether it is free or allocated to some program. This table is built by the boot procedure by sensing the presence of I/O devices in the system. In the partitioned resource allocation approach, the OS considers the

Table 1.6 Resource allocation table

Resource name	Class	Address	Allocation status
printer1	Printer	101	Allocated to P1
printer2	Printer	102	Free
printer3	Printer	103	Free
disk1	Disk	201	Allocated to P1
disk2	Disk	202	Allocated to P2
cdw1	CD writer	301	Free

number of resources and programs in the system and decides how many resources of each kind would be allocated to a program. For example, an OS may decide that a program can be allocated 1 MB of memory, 2000 disk blocks and a monitor. Such a collection of resources is called a *partition*.

In the pool-based allocation approach, OS consults the resource table when a program makes a request for a resource, and allocates accordingly. When many units of a resource class exist in the system, a resource request only indicates the resource class and the OS checks if any unit of that class is available for allocation. Pool based allocation incurs overhead of allocating and deallocating resources individually; however, it avoids both problems faced by the resource partitioning approach by adapting the allocation to resource requirements of programs.

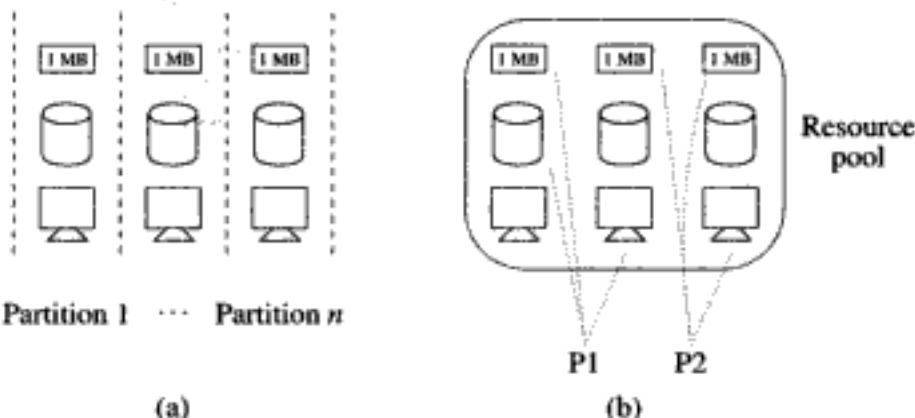


Fig. 1.6 Resource partitioning and pool based allocation

Figure 1.6(a) shows a set of partitions that are defined during boot time. The resource table contains entries for resource partitions rather than for individual resources. A free partition is allocated to each program before its execution is initiated. Figure 1.6(b) illustrates pool based allocation. Program P1 has been allocated a monitor, a disk area of 2000 blocks and 1 MB of memory. Program P2 has been allocated a monitor and 2 MB of memory; disk area is not allocated because P2 did not request it. Thus pool based allocation avoids allocation of resources that are not

needed by a program. Programs with large or unusual resource requirements can be handled by the OS so long as the required resources exist in the system.

Sharing of resources Programs can share a resource in two ways:

- Sequential sharing
- Concurrent sharing.

In sequential sharing, a resource is allocated for exclusive use by a program. When the resource is deallocated, it is marked *free* in the resource table. Now it can be allocated to another program. In concurrent sharing, two or more programs can concurrently use the same resource. Examples of concurrent sharing are data files like bus time tables. Most other resources cannot be shared concurrently. Unless otherwise mentioned, all through this text resources are assumed to be only sequentially shareable.

The OS deallocates a resource when the program to which it is allocated either terminates or makes an explicit request for deallocation. Sometimes it deallocates a resource by force to ensure fairness in its utilization by programs, or to realize certain system-level goals. This action is called *resource preemption*. We use the shorter term *preemption* for the forced deallocation of the CPU. A program that loses the CPU in this manner is called a preempted program.

CPU sharing The CPU can be shared only in a sequential manner, so it can be assigned to only one program at a time. Other programs in the system have to wait their turn on the CPU. The OS must share the CPU among programs in a fair manner. Therefore, after a program has executed for a reasonable amount of time, it preempts the program and gives the CPU to another program. The function of deciding which program should be given the CPU, and for how long, is called *scheduling*.

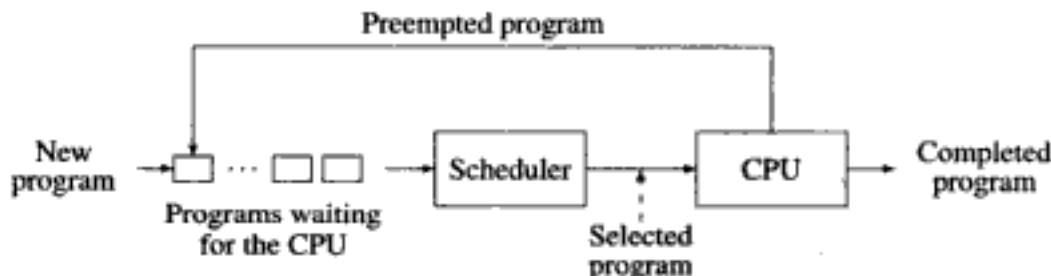


Fig. 1.7 A schematic of scheduling

Figure 1.7 shows a schematic for CPU scheduling. Several programs await allocation of the CPU. The scheduler selects one of these programs for execution on the CPU. A preempted program is added to the set of programs waiting for the CPU.

Memory sharing Parts of memory can be treated as independent resources. Both partitioning and pool-based allocation can be used to manage memory. Partitioning

is simple to implement. It also simplifies protection of memory areas allocated to different programs. However, pool-based allocation achieves better use of memory. Memory can be preempted from inactive programs and used to accommodate active programs. The special term *swapping* is used for memory preemption; the term ‘memory preemption’ is rarely used in OS literature.

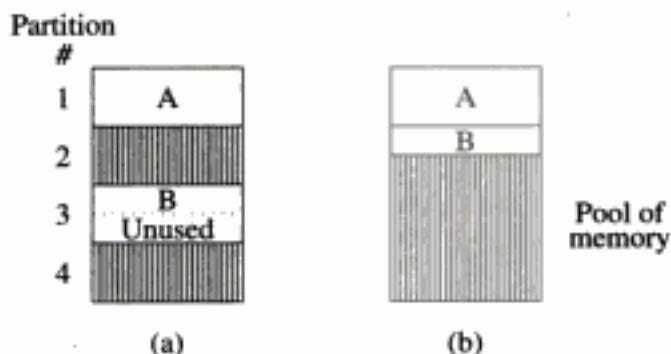


Fig. 1.8 Memory allocation schematics: (a) partitioned allocation, (b) pool-based allocation

Figure 1.8 illustrates the approaches to memory allocation. In the fixed partitioned approach the memory is divided *a priori* into many areas. A partition is allocated to a program at its initiation. Figure 1.8(a) illustrates the situation when memory has been divided into equal sized partitions and two of them have been allocated to programs A and B. Two partitions are currently free. Note that the size of B is smaller than the size of the partition, so some memory allocated to it remains unused. Figure 1.8(b) illustrates the pool based approach. Newly initiated programs are allocated memory from the pool. Each program is allocated only as much memory as requested by it, so allocated memory is not wasted.

Disk sharing Different parts of a disk can be treated as independent resources. Both partitioning and pool based approaches are feasible; however, modern OSs show a preference for the pool based approach. Disk preemption is not practiced by an operating system; individual users can preempt their disk areas by copying their files onto tape cartridges.

1.3.2.1 Virtual Resources

A *virtual resource* is a fictitious resource—it is an illusion supported by an OS through use of a real resource. An OS may use the same real resource to support several virtual resources. This way, it can give the impression of having a larger number of resources than it actually does. Each use of a virtual resource results in the use of an appropriate real resource. In that sense, a virtual resource is an abstract view of a resource taken by a computation.

Use of virtual resources started with the use of virtual devices. To prevent mutual interference between programs, it was a good idea to allocate a device exclusively

for use by one program. However, a computer system did not possess many real devices, so virtual devices were used. An OS would create a virtual device when a user needed an I/O device. Thus, the disk areas called disk1 and disk2 in Table 1.6 could be viewed as small virtual disks based on the real disk. Virtual devices are used in contemporary operating systems as well. A print server is a common example of a virtual device. When a program wishes to print a file, the print server simply copies the file into the print queue. The program requesting the print goes on with its operation as if the printing had been performed. The print server continuously examines the print queue and prints any file it finds in the queue.

Most operating systems provide a virtual resource called *virtual memory*, which is an illusion of a memory that is larger in size than the real memory of a computer. Its use enables a programmer to execute a program whose size may exceed the size of real memory.

Some operating systems create *virtual machines* (VMs) so that each machine can be allocated to a user. The virtual machines are created by partitioning the memory and I/O devices of a real machine. The advantage of this approach is twofold. Allocation of a virtual machine to each user eliminates mutual interference between users. It also permits each user to select an OS of his choice to execute on his virtual machine. In effect, this arrangement permits users to use different operating systems on the same computer system simultaneously (see Section 14.5).

1.3.3 Security and Protection

An unauthorized person may try to use or modify a file. He may also try to interfere with use of the file by authorized users and their programs. An operating system must thwart such attempts. Protection and security are two aspects of this issue. The protection function counters threats of unauthorized use or interference that are posed by users of a computer system, whereas the security function counters similar threats that are posed by persons outside the control of an operating system.

When a computer system functions in complete isolation, the protection and security issues can be separated easily. The operating system verifies the identity of a person through a password check when the person logs in. This procedure is called *authentication*. Therefore, security threats do not arise in the system if the authentication procedure is foolproof. A file open statement executed by a program poses a protection threat if the user who initiated the program is not authorized to access the file. The file system counters this threat by aborting such a program. Figure 1.9 illustrates this arrangement.

When a computer system is connected to the Internet, and a user downloads a program from the Internet, there is a danger that the downloaded program may interfere with other programs or with resources in the system. This is a security threat because the interference is caused by some person outside the system, called an *intruder*, who either wrote the downloaded program, or modified it, so as to interfere with other programs. Such security threats are posed either through a *Trojan horse*,

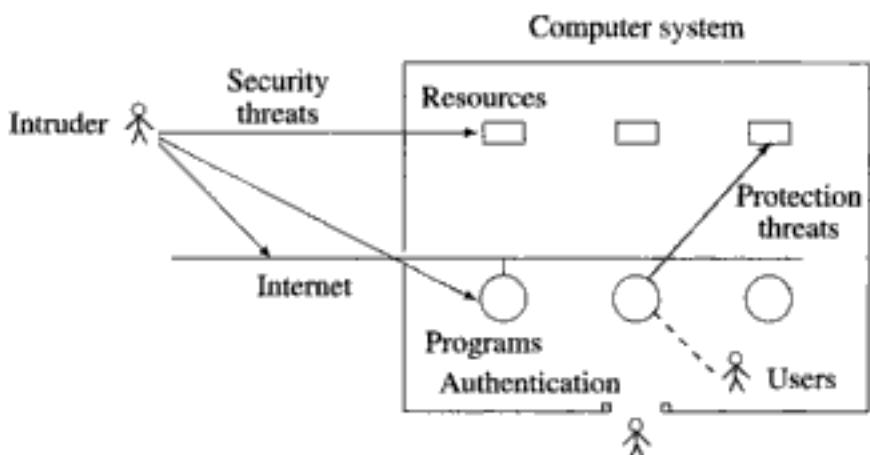


Fig. 1.9 Overview of protection and security concerns

which is a program with a known legitimate function, and a well-disguised malicious function; or through a *virus*, which is a piece of code with a malicious function that attaches itself to other programs and spreads to other systems when such programs are copied. Another class of security threats is posed by *worms*, which are programs that replicate by themselves through holes in the security set-ups of operating systems.

Operating systems address security threats by ensuring that a program cannot be modified while it is being copied over the Internet, and by plugging security holes when they are discovered. Users are expected to contribute to security by exercising caution while downloading programs from the Internet.

1.4 PREVIEW OF THE BOOK

Abstract views help us to limit the scope of study so that we may focus on a selected feature, and to present generic concepts or ideas. We will use abstract views to present the design and implementation of features of an operating system all through this book.

In Part I of the book, we focus on fundamental concepts, i.e., how an operating system organizes its own functioning, and manages user programs, and the fundamental resources in the system, namely, the CPU, memory and files. Part II of the book is devoted to advanced topics in the management of user programs and resources, and the design techniques used to ensure that operating systems can adapt to evolution in computer technology and expectations of computer users. Parts I and II primarily discuss operating systems for conventional computing environments characterized by use of a single computer system having a single CPU; only Chapter 13 discusses operating systems for the multiprocessor computing environment. Part III of the book is devoted to a study of distributed operating systems, which gained in importance in the 1990's due to advances in networking and availability of cheap

computer hardware.

1.4.1 Part I: Fundamental Concepts

An operating system is said to be *event driven*. An *event* is any situation that requires attention of the operating system, e.g., a resource request by a user program, or the end of an I/O operation. When an event occurs, control of the CPU is passed to the operating system. The operating system analyzes the event and performs the appropriate actions. For example, when a program requests a resource, the OS allocates the resource if it is available and when an I/O operation ends, it informs the program that requested the I/O operation and starts another I/O operation on the device, if one is pending. After attending to an event, the OS schedules a user program for execution on the CPU.

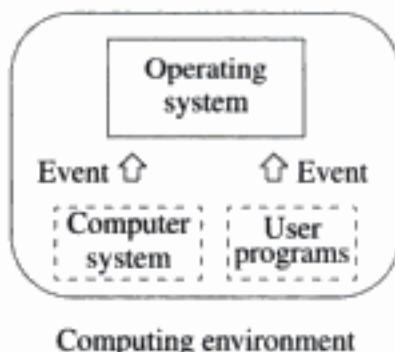


Fig. 1.10 An operating system in its computing environment

The event driven functioning of an operating system shown in Figure 1.10 is a logical view of its functioning. In the corresponding physical view, the end of an I/O operation or a resource request by a program causes an *interrupt* in the computer system. The CPU is designed to recognize an interrupt and divert itself to execution of an interrupt processing routine, which activates an appropriate event handler routine. This physical view, which is the foundation for a study of operating systems, is developed in Section 2.1.

As mentioned earlier in Section 1.2, an operating system has to achieve effective utilization of a computer system, which is the combination of efficient use and user convenience that best suits its computing environment. Different computing environments, such as interactive and real time environments, were evolved in response to advances in computer architecture and new requirements of computer users. In each computing environment, an operating system uses appropriate techniques to handle user programs and system resources. We study these techniques in Chapter 2 of the book. A modern computing environment contains features of several computing environments, such as non-interactive, interactive, i.e., time sharing, real time, and distributed computing environments; so these techniques are used in a modern

operating system as well.

Managing user computations The operating system uses the abstraction called a *process* to help it organize execution of programs. A process is simply an execution of a program. Computational structures give rise to processes with different characteristics, e.g., sequential execution of single programs gives rise to processes that operate one after another, whereas co-executing programs give rise to processes that operate concurrently. Use of the process abstraction enables the operating system to handle all such program executions in a uniform manner. The notion of a *thread* is introduced to reduce the OS overhead involved in managing execution of programs. An OS needs to keep track of much less information concerning a thread than concerning a process. Except for this difference, processes and threads are similar in other respects; so we use the term process as a generic term for both a process and a thread. We discuss processes and threads in Chapter 3.

An operating system aims at achieving effective utilization of a computer system by sharing the CPU among several processes. It uses a *scheduling policy* to decide which process should be executed at any time. In Chapter 4, we discuss the classical scheduling policies, which aimed either at efficient use of a computer system, or at high user convenience; and scheduling policies used in modern operating systems, which aim at suitable combinations of efficient use and user convenience.

Managing memory An OS allocates a memory area when a process requests memory, keeps track of memory areas released by processes, and tries to reuse released memory areas efficiently while satisfying new memory requests. The key issue in memory management is *memory fragmentation*, which is the presence of unusable free memory areas. In Figure 1.11(a), some memory has been left over after allocating memory to processes P_i and P_j . This memory area remains unused because it is too small to accommodate a process. The situation gets worse when process P_i completes its operation, and the OS starts a new process P_k in some part of the memory released by P_i . Now, the memory contains two free but unused areas of memory, even though together they are large enough to accommodate some other process P_l . We describe memory reuse techniques and the problem of memory fragmentation in Chapter 5.

Modern operating systems provide *virtual memory*, which is an illusion of a memory that is larger than the actual memory of a computer system. Use of virtual memory also overcomes the problem of memory fragmentation. Virtual memory is implemented as follows: The code and the data of a process are stored on a disk. At any instant during the operation of the process, only some parts of its code and data are kept in the memory; new parts are loaded in memory only when they are needed during its operation. The OS removes a part of a process's code or data from the memory if it feels that the part may not be needed for some time during the operation of the process.

Virtual memory is implemented using the *noncontiguous memory allocation*

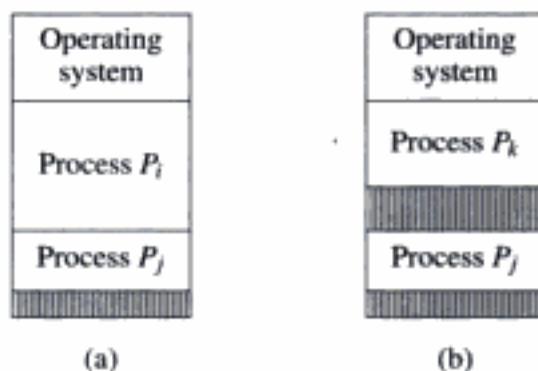


Fig. 1.11 Memory fragmentation: Unusable free areas in memory increase when P_i terminates

model. The CPU passes each instruction address or data address used during operation of the process to a special hardware unit called the *memory management unit*, which consults the memory allocation information for the process and computes the address in the memory where the instruction or data actually resides. If the part of a process that contains a required address does not exist in memory, a ‘missing from memory’ interrupt occurs and the OS loads the missing part in memory and resumes operation of the process (see Figure 1.12). To achieve efficient operation, the OS has to ensure a low rate of ‘missing from memory’ interrupts. The techniques used for this purpose are described in Chapter 6.

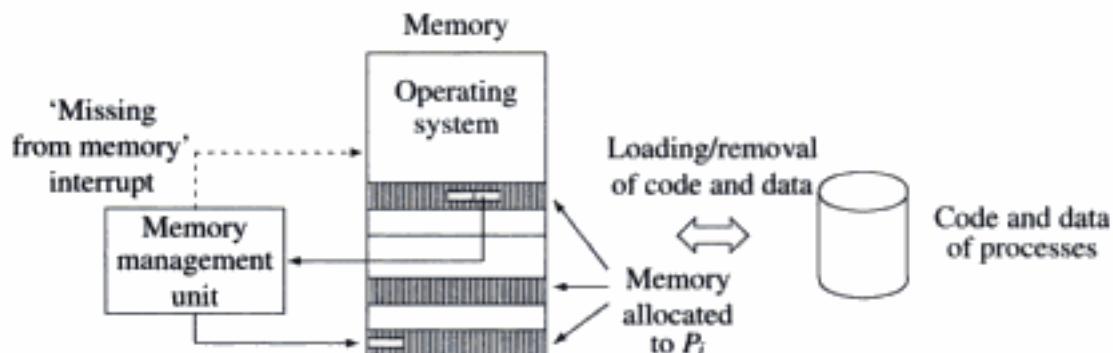


Fig. 1.12 Virtual memory operation

Managing files Computer users expect fast access to files, reliability in the face of faults, ability to share files with co-workers, and a guarantee that no unauthorized persons can use, or tamper with, their files. The file system provides a logical view to a user that consists of a directory hierarchy of the file system, in which each user has a home directory. The user can create directories, or folders, as they are called in the Windows operating system, in his home directory, and other directories or folders in these directories, and so on. A user can authorize some co-workers to use some of

his files, by informing the file system about the names or ids of co-workers and the files or directories that they are authorized to use. The file system also performs a few other tasks. It allocates space on a disk to record a file. To ensure reliability, it protects the data in a file, and also protects its own data such as directories, against damage due to faults such as faulty I/O media or power outages. All these features of file systems are discussed in Chapter 7.

The arrangement used to implement protection and security of files was described earlier in Section 1.3.3. To strengthen the security arrangement, the passwords data stored in the operating system is encrypted with a secret key known only to the operating system. Hence an intruder cannot obtain passwords of users except through trial and error, which is prohibitively expensive and time consuming. Protection and security threats, the technique of encryption, and various methods used to implement protection are described in Chapter 8.

1.4.2 Part II: Advanced Topics

Process synchronization and deadlocks Processes of an application coordinate their activities to perform their actions in a desired order; this is called *process synchronization*. Figure 1.13 illustrates two kinds of process synchronization. Figure 1.13(a) shows processes named *credit* and *debit* that access the balance in a bank account. The results may be incorrect if both processes update the balance at the same time, so they must perform their updates strictly one after another. Figure 1.13(b) shows a process named *generate* that produces some data and puts it into a variable named *sample*, and the process named *analyze* that performs analysis on the data contained in variable *sample*. Here, process *analyze* should not perform analysis until process *generate* has deposited the next lot of data in *sample*, and process *generate* should not produce the next lot of data until process *analyze* has analyzed the previous data. Programming languages and operating systems provide several facilities that processes may use to perform synchronization, while computer architecture provides some special instructions to support the implementation of these facilities. All these features and process synchronization techniques using them are described in Chapter 9.

Processes may also interact through *message passing*. When a process sends some information in a message to another process, the operating system stores the message in its own data structures until the destination process makes a request to receive a message. It also synchronizes operation of the sender and destination processes. Details of message passing are described in Chapter 10.

Processes share the resources of a computer system. If a resource requested by some process P_i is currently allocated to process P_j , P_i has to wait until P_j releases the resource. A *deadlock* is a situation in which some processes wait for each other's actions indefinitely. It adversely affects performance of a system because processes involved in it cannot make any progress and resources allocated to them are wasted. An operating system uses several techniques to deal with deadlocks. They range from

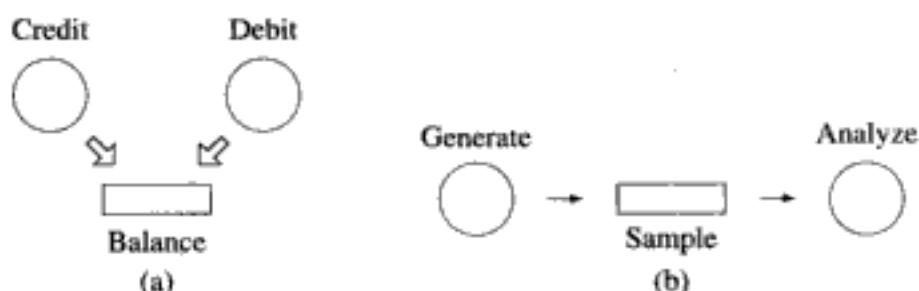


Fig. 1.13 Two kinds of process synchronization

drastic measures such as aborting a process involved in a deadlock and distributing its resources among other processes, to use of resource allocation policies which ensure that deadlocks do not arise in a system. We discuss deadlock handling techniques used in operating systems in Chapter 11.

Implementation of file operations While implementing operations on a file, it is important to ensure good performance of a process engaged in a file processing activity, and good performance of I/O devices. To achieve this, file operations are actually implemented by an *Input output control system* (IOCS). When a process issues a read operation on a file, the file system invokes the IOCS to implement the read operation by performing appropriate I/O operations on a device. The IOCS employs several techniques to implement I/O operations efficiently. The techniques of *buffering* and *blocking* speed up access to data stored on an I/O device, while the techniques of *caching* and *disk scheduling* are used to ensure good performance of a disk. Different kinds of I/O devices, and the techniques of buffering, blocking, and disk scheduling used in an IOCS are discussed in Chapter 12.

Multiprocessor operating systems A *multiprocessor computer system* can provide high performance by servicing several processes simultaneously. It can also speed-up user computations by scheduling its processes simultaneously. The operating system has to use special scheduling and synchronization techniques to realize these advantages. We discuss these techniques in Chapter 13.

Structure of operating systems Modern operating systems such as Unix and Windows have very long lifetimes during which time several changes take place in computer architecture and computing environments. Hence such operating systems have to be *portable* so that they can be implemented on many computer architectures, and *extendible* so that they can meet new requirements due to changes in the nature of its computing environment. We discuss operating system design techniques for portability and extendibility in Chapter 14.

1.4.3 Part III: Distributed Operating Systems

Architecture of distributed systems A distributed computer system consists of several computer systems, called *nodes*, connected through a network. Use of a distributed computer system provides three key advantages: speeding up of a computation by scheduling its processes in different nodes of the system simultaneously, high reliability through redundancy of resources, and sharing of resources in different nodes. New computation paradigms such as remote procedure calls are employed to exploit the features of a distributed system. The model of a distributed computer system, its networking hardware and software, distributed computation paradigms, and kinds of operating systems used for distributed systems are discussed in Chapter 15.

Theoretical issues in distributed systems Networking delays in a distributed system may lead to an inconsistent view of data located in different nodes of the distributed system. Figure 1.14 shows bank accounts A and B located in nodes X and Y of the distributed system that contain 2000 dollars and 500 dollars, respectively. Let a banking application transfer 1000 dollars from account A to account B. The observer located in node Z would obtain an inconsistent view of the balances in accounts A and B if he records the balance in account A before the transfer, and the balance in account B after the transfer—he would find that accounts A and B contain 2000 and 1500 dollars, respectively. Algorithms to record a consistent view of data in a distributed system must avoid such problems. Similar problems are faced in determining the chronological order of events occurring in different nodes of the system. These problems, and the methods for tackling them, are discussed in Chapter 16.

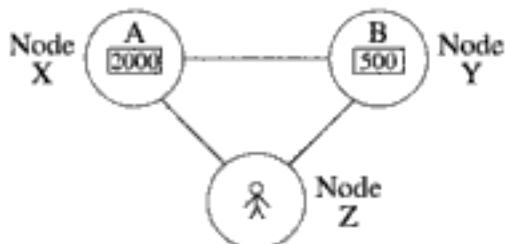


Fig. 1.14 An observer records inconsistent balances in bank accounts A and B

Distributed control functions Control functions of a conventional operating system, such as scheduling, resource allocation and deadlock detection, require a consistent view of (1) data concerning resources, and (2) activities in processes. In a distributed system, however, use of algorithms for recording consistent views involves delays and overhead, so distributed operating systems use special algorithms for performing control functions without requiring consistent information about resources and processes located in different nodes of the system. These algorithms are discussed in Chapter 17.

Recovery The *state* of a process is the collection of information describing the current activity in the process. A fault like a crash of a node or a communication link, may affect or destroy states of some of the processes that were in operation when the fault occurred. Operating systems use two approaches to restore processes affected by a crash. In the *fault tolerance* approach, sufficient information concerning the state of a process is stored during operation of a process. It is used to repair the state of a process when a fault occurs. In the *recovery* approach, the state of a process is recorded periodically in the form of a *back-up*, and the process is restored to this state when a fault occurs. To avoid inconsistencies when a process is restored to a previous state, some other processes might also have to be restored to their earlier states, and so on. For example, consider failure of node Y in the banking application of Figure 1.14 while 1000 dollars are being added to the balance in account B. A fault tolerance technique would complete the update operation after the failure and get the correct balance, i.e., 1500 dollars, in account B. A recovery technique might restore the balance in account B to an old value, i.e., 500 dollars. Now, it would also have to restore the balance in account A to a mutually consistent old value, i.e., 2000 dollars. Fault tolerance and recovery techniques used in distributed systems are discussed in Chapter 18.

Distributed file system A distributed file system stores files in several nodes of a distributed system, so a file and a process that accesses it may exist in different nodes. This situation has several consequences: Performance of the file processing activity is poor because data in the file is accessed over the network. Also, failure of either node, or failure of the network connecting the two, can disrupt the file processing activity. A distributed file system improves performance by caching a file in the node where the process exists, and improves reliability through special techniques that ensure that a transient failure of a node containing the process or the file does not disrupt a file processing activity. These and other techniques used by distributed file systems are discussed in Chapter 19.

Distributed system security When processes in a distributed system exchange messages, the messages may travel over public communication channels and may pass through computer systems that are not under the control of the distributed operating system. As discussed in Section 1.3.3, an intruder may gain control of these communication channels or computer systems and tamper with messages, copy or destroy them, or create fake messages. Using modified, copied or fake messages, the intruder may even be able to fool the authentication procedure and impersonate a user of the system. These security threats are countered by using encryption to prevent tampering of messages, and using special authentication procedures to prevent impersonation of users through copied messages. These techniques are discussed in Chapter 20.

EXERCISE 1

1. Give examples of two situations in which user convenience conflicts with efficient use of a computer system.
2. Resource preemption may be performed to provide (a) fairness in the use of resources, (b) more efficient use. Describe a situation in which resource preemption provides fairness in the use of resources. Read Chapter 2 and describe a situation in which preemption is used to obtain efficient use.
3. An OS designer makes the following policy statement: "Consider preemption of a resource from a program only if the program can resume its operation after the resource is re-granted to it *as if the preemption had not occurred*." Justify preemption of CPU and memory from a program. Also argue why preemption of a magnetic tape cartridge or printer is inadvisable.
4. In the Unix operating system, a print server implements printing of files as desired by users. When a user executes a command to print a file, the print server copies the file on the disk and enters it in a print queue. It is printed when its entry reaches the head of the print queue.
Write a description of the print service as a use of virtual devices and spooling.
5. Comment on validity of the following statement: Partitioned resource allocation provides more user convenience than pool-based allocation but may provide poor efficiency."
6. Write a short note on the costs and benefits of pool based resource allocation. Describe situations in which it enhances (a) efficiency, or (b) user convenience.
7. Write a short note on swapping describing (a) OS overhead, and (b) impact on efficiency.
8. Comment on implications of the following features for efficient use and user convenience
 - (a) Sequence of programs
 - (b) Virtual devices.
9. A program is in a dormant state if it is not engaged in any activity (it may be waiting for an action by a user). Comment on validity of the following statement: "A dormant program also consumes resources."
10. If a 'component of an OS' is defined as a program that helps in handling either user computations or resources, would the following programs qualify to be called components of an OS? Give a yes/no answer supported by arguments. If you have any reservations about a yes/no answer, describe them but give a yes/no answer anyway.
 - (a) Compiler for a concurrent programming language
 - (b) Command processor
 - (c) ssh program (this program is used to use a computer system remotely)
 - (d) File system.
11. A logical view of execution of a program written in a higher level language L consists of the data of the program being input to its code in language L to produce results (see Figure 1.3(a)). Develop a physical view of the compilation and execution of this program.

12. A logical view of file processing performed by a user consists of directories, files and statements in a program to open a file and read/write its data. Write a note on the physical view of file processing performed by a user.
13. A computing environment is simply the environment in which a computer system is used. Make a list of computing environments that have distinct features concerning effective utilization.

BIBLIOGRAPHY

The view of an OS as a collection of programs is usually propounded in most operating systems texts. Tanenbaum (2001), Stallings (2001), Nutt (2004) and Silberschatz *et al* (2005) are some of the recent texts on operating systems.

Berzins *et al* (1986) discuss how the complexity of designing a software system can be reduced by constructing a set of abstractions that hide the internal working of a subsystem. Most books on software engineering discuss the role of abstraction in software design. The paper by Parnas and Siewiorek (1975) on the concept of transparency in software design is considered a classic of software engineering. The book by Booch (1994) discusses abstractions in object oriented software development.

The concept of virtual devices was first used in the spooling system of the Atlas computer system developed at Manchester University. It is described in Kilburn *et al* (1961).

Ludwig (1998) and Ludwig (2002) describe different kinds of viruses, while Berghel (2001) describes the Code Red worm that caused havoc in 2001. Pfleger and Pfleger (2003) is a text on computer security. Garfink el *et al* (2003) discusses security in Solaris, Mac OS, Linux, and FreeBSD operating systems. Russinovich and Solomon (2005) discuss security features in Windows.

1. Berghel, H. (2001): "The Code Red worm," *Communications of the ACM*, **44** (12), 15–19.
2. Berzins, V., M. Gray, and D. Naumann (1986): "Abstraction-based software development," *Communications of the ACM*, **29** (5), 403–415.
3. Booch, G. (1994): *Object-Oriented Analysis and Design*, Benjamin-Cummings, Santa Clara.
4. Garfink el, S., G. Spafford, and A. Schwartz (2003): *Practical UNIX and Internet Security, Third edition*, O'Reilly, Sebastopol.
5. Kilburn, T., D. J. Howarth, R. B. Payne, and F. H. Sumner (1961): "The Manchester University Atlas Operating System, Part I : Internal Organization," *Computer Journal*, **4** (3), 222–225.
6. Ludwig, M. A. (1998): *The Giant Black Book of Computer Viruses, Second edition*, American Eagle, Show Low.
7. Ludwig, M. A. (2002): *The Little Black Book of Email Viruses*, American Eagle, Show Low.
8. Nutt, G. (2004): *Operating Systems—A Modern Perspective, Third edition*, Addison-Wesley, Reading.
9. Parnas, D. L., and D. P. Siewiorek (1975): "Use of the concept of transparency in the design of hierarchically structured systems," *Communications of the ACM*, **18** (7), 401–408.
10. Pfleger, C. P., and S. Pfleger (2003): *Security in Computing*, Prentice Hall, N.J.

26 Operating Systems

11. Russinovich, M. E., and D. A. Solomon (2005): *Microsoft Windows Internals, Fourth edition*, Microsoft Press, Redmond.
12. Silberschatz, A., P. B. Galvin, and G. Gagne (2005): *Operating System Principles, Seventh edition*, John Wiley, New York.
13. Stallings, W. (2001): *Operating Systems—Internals and Design Principles, Fourth edition*, Pearson Education, New York.
14. Tanenbaum, A. S. (2001): *Modern Operating Systems, Second edition*, Prentice-Hall, Englewood Cliffs.

Part I

FUNDAMENTAL CONCEPTS

An operating system controls use of a computer system's resources such as CPUs, memory, and I/O devices to meet computational requirements of its users. Users expect convenience, quality of service, and security while executing their programs, whereas system administrators expect efficient use of the computer's resources and good performance in executing user programs. These diverse expectations can be characterized as *user convenience, security* and *efficient use* of resources; they form the primary goals of an operating system. The extent to which an operating system meets each of these goals depends on its *computing environment*, i.e., the computer system's hardware, its interfaces with other computers, and the nature of computations performed by its users.

A *process* is an execution of a program. Programmers and operating systems view and use processes differently. To a programmer, the most important feature of processes is their ability to execute concurrently and to interact with one another to fulfill a common goal. To an operating system, a process is a means for achieving execution of programs. Accordingly, we discuss issues concerning creation of, and interaction between, processes; and their *scheduling* for use of a CPU.

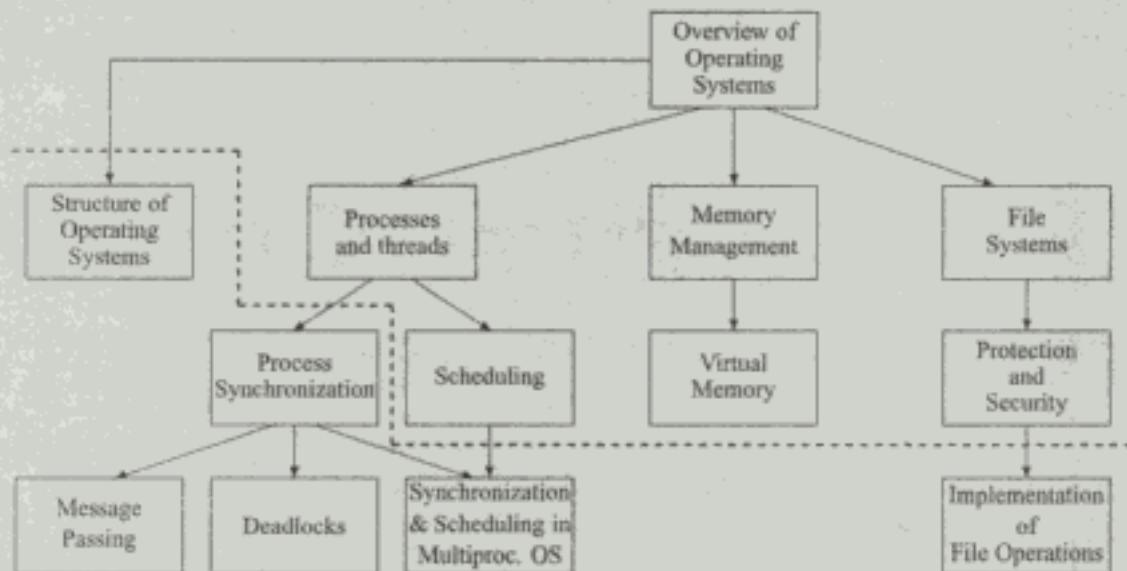
The memory of a computer system is shared by a large number of processes. The number of processes that can be accommodated in memory at any time influences performance of the system because presence of too few processes in memory could lead to CPU idling. System performance deteriorates when some memory areas remain unused because they are too small to accommodate a process. This situation is called *memory fragmentation*; the OS must prevent it from arising. The OS should also enable execution of large programs, whose size might exceed size of memory.

Computer users expect convenience and efficiency while creating, manipulating and sharing files with other users of the system. They also expect a file system to provide protection, security and reliability features so that their files are not subjected to illegal accesses or tampering by other persons, or damage due to faults.

In this Part, we discuss the fundamental concepts and techniques used in operating systems to govern use of CPUs, memory and files.

Road Map for Part I

Part I: Fundamental Concepts



Part II: Advanced Topics

Chapter 2: Overview of Operating Systems

This key chapter introduces the fundamental principles of an operating system (OS). Initial sections discuss how an OS uses features in the hardware of the computer system to control computational and I/O activities in user programs, and how user programs use features in the hardware to interact with the OS and obtain the services they need. Later sections discuss five classes of operating systems—*batch processing, multiprogramming, time sharing, real time* and *distributed operating systems*—and describe the principal concepts and techniques they use to meet their goals. The last section discusses how a modern OS uses most of these concepts and techniques.

Chapter 3: Processes and Threads

This chapter discusses both the programmer view and the OS view of processes. Discussion of the programmer view concerns how processes are created and terminated, and how they interact with one another. Discussion of the OS view focuses on the way the operating system manages a process—how it uses the notion of *process state* to keep track of what a process is doing and how it reflects the effect of an event on states of affected processes. The chapter also introduces the notion of *threads* and illustrates their features.

Chapter 4: Scheduling

Scheduling is the act of selecting the next process to be serviced by a CPU.

This chapter discusses how a scheduler uses the fundamental techniques of *process priorities*, *reordering* of requests and variation of *time slice* to achieve a suitable combination of user service, efficient use of resources, and system performance. It describes different scheduling policies and their properties.

Chapter 5: Memory Management

This chapter is devoted to the fundamentals of memory management. It begins by discussing the memory allocation model used for a process and the hardware support for memory protection. This is followed by a discussion of memory allocation techniques, and the causes of *memory fragmentation*. It then discusses elements of the *noncontiguous memory allocation* model, which is used to reduce memory fragmentation. The hardware support for the noncontiguous memory allocation model and the two approaches to noncontiguous memory allocation called *paging* and *segmentation*, are discussed.

Chapter 6: Virtual Memory

A virtual memory system uses the noncontiguous memory allocation model. It keeps the code and data of a process on a disk and loads portions of it into memory when required. The principle of *locality of reference* helps to predict which portions of a process are likely to be referenced in immediate future. The performance of a process is determined by the rate at which its portions have to be loaded into memory, which depends on the amount of memory allocated to it.

This chapter deals with virtual memory implementation using *paging* in detail. It discusses the notion of *working set* used to determine the amount of memory that should be allocated to each process, and *page replacement algorithms* used to decide which pages of a process should be in memory at any time. Aspects of code and data sharing are discussed. Virtual memory implementation using *segmentation* is also described.

Chapter 7: File systems

This chapter discusses a programmer's view of files and the file system. It describes fundamental *file organizations*, *directory structures*, operations on files and directories, and *file sharing semantics*, which determine the manner in which results of file manipulations performed by concurrent processes are visible to one another. Issues which compromise reliability of a file system are discussed. Fault tolerance using *atomic actions* and recovery using *back-ups* are described.

Chapter 8: Protection and Security

Protection and security measures together ensure that only authorized users can access a file. This chapter discusses different kinds of protection and security threats in an operating system, protection and security measures used to thwart these threats, and the role played by the *encryption* technique in implementing these measures.

Overview of Operating Systems

Interaction of an operating system with the computer system and user programs is a key aspect of its operation. Features of computer system architecture are used to realize this interaction. We discuss two arrangements used to implement this interaction—interrupt processing and system calls.

A computing environment is characterized by a computer system, its interfaces with other systems, its users and the nature of their computational requirements. Goals of an OS are determined by the notion of *effective utilization* of a computer system in the computing environment where it is used, so operating systems are classified on the basis of computing environments where they are used.

We discuss different classes of operating systems. For each class, we characterize the computing environment in which an OS of that class functions and describe typical techniques used to achieve effective utilization of a computer system. We also introduce relevant terminology and definitions of concepts.

A modern OS contains features of several classes of operating systems, so the overview of concepts and techniques provided in this chapter is important for appreciating the need and significance of various features. The overview also provides a useful background for the study of operating system components in later chapters.

2.1 OS AND THE COMPUTER SYSTEM

Figure 2.1 illustrates use of memory during operation of an operating system. The memory is divided into a system area and a user area. Some OS programs exist permanently in the system area of the memory to monitor and control activities in the computer system. Other programs exist on a disk and are loaded in the transient area when needed. Remainder of the memory is occupied by user programs. A variety of terms have been used to describe the collection of OS programs that exist

in the memory, e.g., monitor, supervisor, executive and kernel. In fact, different terms were used in the context of different classes of operating systems. In this book, we uniformly use the term *kernel* for the collection of OS programs in memory.

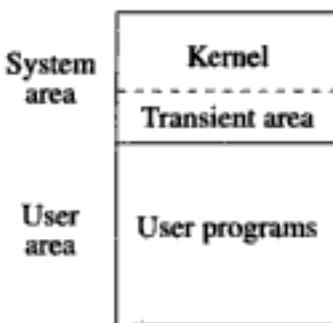


Fig. 2.1 Memory utilization during operation of an OS

The kernel fulfills the goals of an OS by performing a set of control functions. It is a collection of programs, and not a hardware unit, so it performs control functions by executing instructions on the CPU. Thus the CPU is used by both, user programs and the OS. The former constitutes productive use of the CPU, and contributes to CPU efficiency. The latter constitutes overhead of the operating system.

We use the term *switching of the CPU* for an action that forces the CPU to stop executing one program and start executing another program. When the kernel needs to perform a control functions, the CPU must be switched to execution of the kernel. After completing the control functions, the CPU is switched back to execution of a user program. Switching of the CPU between user programs and the kernel is the most crucial aspect of the operation of an OS. Section 2.1.2 describes how it is realized.

We begin this Section by describing important features of a computer system.

2.1.1 The Computer System

Figure 2.2 shows a model of a computer system that shows only those functional units that are relevant from the viewpoint of an operating system. We describe important details of these units in the following.

The Central Processing Unit (CPU)

The CPU contains two kinds of registers. Program accessible registers, sometimes called *general purpose registers* or simply *CPU registers*, are used to store addresses or data during execution of a program. *Control registers* contain information that controls or influences the operation of the CPU itself. The *program counter* (PC) register contains address of the next instruction to be executed by the CPU. The *condition code* (CC) register contains a code describing some properties of the last arith-

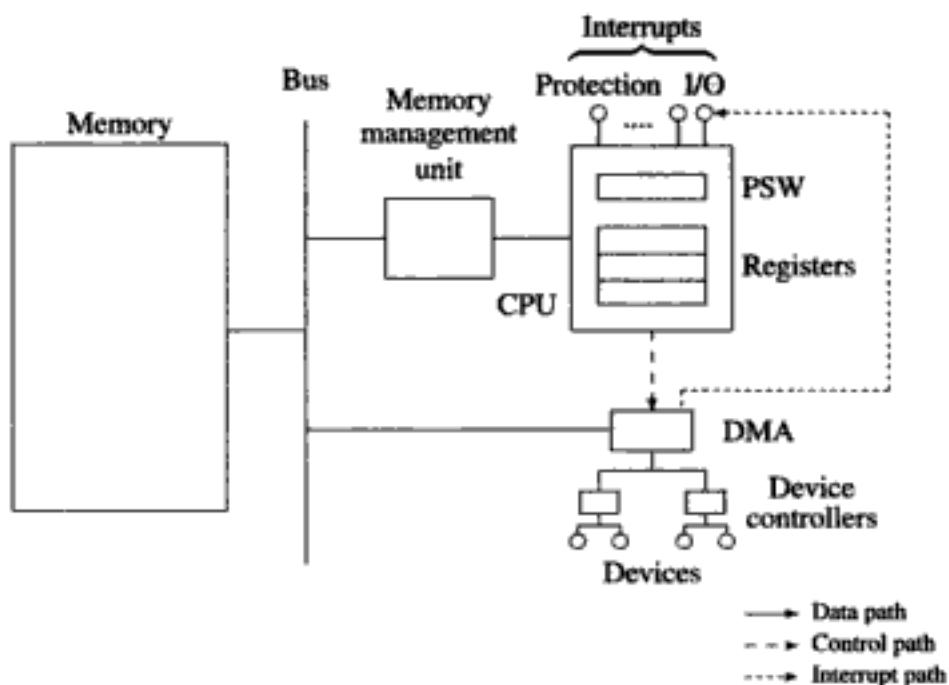


Fig. 2.2 Model of a computer system

metric or logical result computed by the CPU (e.g., whether it is less than zero). Contents of other control registers are described later in this Section. The *program status word* (PSW) is a collection of control registers in the system. Figure 2.3 illustrates the PSW. We refer to each individual control register as a field of the PSW.

IM	P	IC	MPI	CC	PC
IM : Interrupt Mask					
P : Privileged mode					
IC : Interrupt Code					
MPI : Memory Protection Information					
CC : Condition Code					
PC : Program Counter					

Fig. 2.3 Fields of the Program Status Word (PSW)

Privileged mode of CPU operation The CPU can operate in two modes called *user mode* and *privileged mode*. The *privileged mode* (P) field of the PSW is a single bit field. We assume that this field contains a 1 if the CPU is in privileged mode and a 0 if it is in the user mode. Certain instructions can be executed only when the CPU is in the privileged mode. Such instructions are called *privileged instructions*.

Privileged instructions typically implement control operations required by an OS to perform a control function, e.g., initiating an I/O operation or setting protection information for programs. Instructions that change contents of PSW fields are them-

selves privileged instructions. The CPU is put in the privileged mode when the kernel executes so that the kernel can perform control functions. The CPU is put in the user mode when a user program is executing. This action ensures that the user program cannot execute any instruction that may interfere with operation of other user programs or the OS.

State of CPU The set of program accessible registers and the PSW together contain all information necessary to execute a program on the CPU. The program accessible registers contain all data and address values put there during execution of the program. The program counter (PC) field contains address of the next instruction to be executed. The condition code (CC) field contains a code describing the last arithmetic or logical result computed by the CPU. This code can be used by a conditional branch instruction. The privileged mode (P) field indicates whether the CPU is in the privileged mode. The memory protection info (MPI) field contains information indicating the memory entities that can be accessed by the program. The interrupt mask (IM) and interrupt code (IC) fields contain information concerning handling of interrupts. Use of this information is described later in this Section.

Contents of CPU registers and the PSW together indicate what the CPU is doing at any moment, so their contents are said to constitute the *state* of the CPU. If the kernel needs to perform a control function while a user program is executing on the CPU, it saves the state of the CPU and uses the CPU to execute its own code. When execution of the user program is to be resumed, the kernel simply reloads the saved CPU state into the registers and the PSW. Execution of the program now resumes from the point at which the CPU was taken away from it.

Address		Instruction			PSW	P
		MOVE	A, ALPHA	00 0150		0
0142		COMPARE	A, 1	CC	PC	
0146		BEQ	NEXT			
0150		...				
0192	NEXT	...			CPU registers	1 A
0210	ALPHA	DS	1			B
		...				X

(a)
(b)

Fig. 2.4 (a) A program, (b) State of the CPU after executing the COMPARE instruction

Example 2.1 Figure 2.4 shows a program for a hypothetical computer. The CPU has two data registers A and B and an index register X. The program moves the value of ALPHA to register A and compares it with 1. Let the value of ALPHA be 1, hence the comparison operation sets the condition code to 00 to indicate that the two operands are equal. An interrupt occurs at the end of the COMPARE instruction. The state of the CPU is as shown in Figure 2.4(b). It contains the PSW, and contents of registers A, B and X. The condition code register contains 00 and the PC contains 150, which is the address of the BEQ (branch if equal) instruction. If the CPU state is loaded back

into the CPU, the program would resume its execution at the BEQ instruction. The condition code register contains 00, so the instruction would execute correctly.

Memory Hierarchy

Ideally, a computer system should contain a large and fast memory so that the CPU is not slowed down due to accesses to memory. However, fast memory is expensive, so a memory hierarchy is used to create the illusion of a large and fast memory at a low cost. A memory hierarchy contains a number of memory units with differing speeds. The fastest memory in the hierarchy is the smallest in size; slower memories are larger in size. The CPU accesses only the fastest memory. If the data or instruction needed by the CPU does not exist in this memory, it is transferred there from a slower memory. The effective memory access time depends on how frequently such transfers are needed during execution of a program.

Figure 2.5 shows a schematic of a memory hierarchy. The hierarchy contains three memory units. The cache memory is fast and small. The main memory (we will simply call it memory) is slow and large. The disk is the slowest and largest unit in the hierarchy. The cache memory speeds up access to the main memory, and the main memory speeds up access to the disk. The *memory management unit* (MMU) helps in effective memory management by an OS. The CPU passes the address of the byte that it wishes to access to the MMU. The MMU passes a translated address to the cache memory. The cache memory reads or writes the required byte, making accesses to the main memory only if the required byte is not present in the cache memory. Accesses are made to the disk only if the required byte is not present in the main memory.

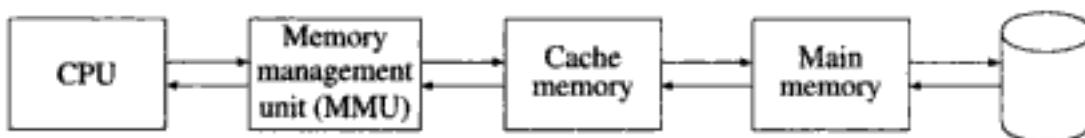


Fig. 2.5 Memory hierarchy containing cache, main memory and disk

Memory management unit (MMU) The MMU ‘translates’ each memory address generated by the CPU into another address called the physical memory address. The address translation feature is used in noncontiguous memory allocation. It is also used in implementing virtual memory systems that use the disk as a part of the memory hierarchy (see Chapter 6). In the interest of simplicity, we do not describe this feature here.

Cache memory The cache memory holds some instructions and data values that were recently accessed by the CPU. When the CPU wishes to read a byte of memory, the cache is searched to check whether the byte exists in the cache. If so, it is accessed from the cache. If the byte is not present in the cache, it is loaded from the memory

and the access is completed. To enhance cache performance, the memory hardware does not load a single byte from memory into the cache. It always loads a fixed sized block of memory, called a *cache block* or *cache line*, into the cache. This way, access to a byte that is in proximity of a recently accessed byte can be implemented without having to access the memory. When the CPU writes a new value into a byte, the value is written into the cache. Sooner or later it should also be written into the memory. Different schemes exist for updating the byte in memory. A simple scheme is to write the byte into the cache and the main memory at the same time.

The *hit ratio* (h) of the cache indicates what fraction of bytes accessed by the CPU were found in the cache. High hit ratio values are observed in practice because of two reasons—programs tend to make memory accesses that are in proximity of previous accesses, which is called *spatial locality*, and some data and instructions are accessed repeatedly, which is called *temporal locality*. Effective memory access time is given by the formula

Effective memory access time

$$\begin{aligned} &= h \times \text{access time of cache memory} \\ &\quad + (1 - h) \times (t_{tra} + \text{access time of cache memory}) \end{aligned} \quad (2.1)$$

where t_{tra} is the time taken to transfer a cache block from the memory to the cache. If we consider a cache block that is 1 byte in size, and a cache memory that is ten times faster than the memory, a hit ratio of 0.8 provides an effective memory access time that is only 28 percent of the access time of memory, i.e., it speeds up memory accesses by over three times! However, cache block size of 1 byte does not exploit spatial locality, hence a hit ratio of 0.8 is unrealistic. Therefore larger cache blocks are used in practice, and advanced memory organizations are used to reduce t_{tra} . The Intel Pentium processor uses a cache block size of 128 bytes and a memory organization that makes t_{tra} about 10 times the memory access time. With a cache hit ratio of 0.97, this organization provides an effective memory access time that is 40 percent of the access time of memory.

Processor architectures provide several levels of cache memories to enhance the access speed-up. Note that the cache hit ratio is poor at the start of execution of a program because none of its instructions or data exist in the cache. The hit ratio is higher if the program has been in execution for some time.

Processor architectures use several levels of cache memories to reduce the effective memory access time. An L1 cache—that is, a level 1 cache—is incorporated into the CPU chip itself. An L2 cache may be external to the CPU but still be directly connected to it. An L3 cache may be part of a memory chip.

Main memory Memory units are directly connected to the bus. The CPU is connected to the bus through the memory management unit and the cache, and I/O devices are connected to the bus through device controllers (see Figure 2.2). If the CPU and I/O devices try to access the memory at the same time, the bus permits only

one of them to proceed. The other accesses are delayed until this access completes.

Memory protection Many programs coexist in the memory of the computer system, so it is necessary to prevent instructions of one program from destroying contents of memory being used by another program. This function is called memory protection. Memory protection is implemented by checking whether a memory word accessed by a program lies outside the memory area allocated to it.

A popular memory protection technique uses the notion of *memory bound registers* (also called *fence registers*). The start and end addresses of the memory area allocated to a program are loaded in the *lower bound register* (LBR) and *upper bound register* (UBR) of the CPU, respectively. Before making any memory reference, say reference to a memory location with address *aaa*, the memory protection hardware checks whether *aaa* lies outside the range of addresses defined by contents of LBR and UBR. If so, it generates an interrupt to signal a memory protection violation. As described in a later Section, the kernel now terminates the erring program. The MPI field of the PSW (see Figure 2.3) contains LBR and UBR. This way the memory protection information also becomes a part of the CPU state. Example 2.2 illustrates how memory protection is implemented.

Example 2.2 A program is allocated the memory area 200K–250K by the kernel. Figure 2.6 illustrates memory protection for this program using memory bound registers. LBR is loaded with the start address of the allocated area (i.e., 200K) while UBR is loaded with the end address (i.e., 250K). A memory protection violation interrupt would be generated if an address used during execution of P_1 lies outside the range 200K–250K.

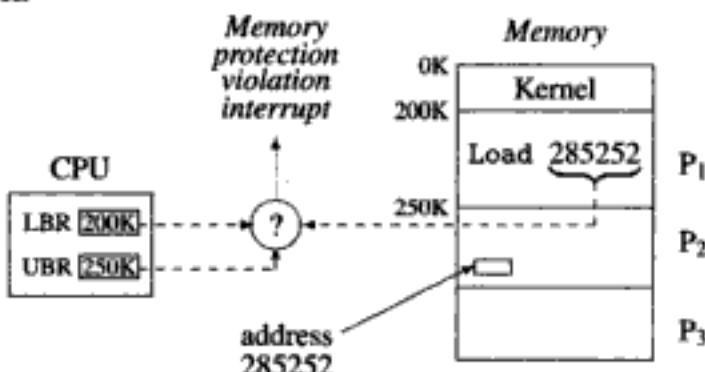


Fig. 2.6 Memory protection using bound registers

The I/O Subsystem

I/O operations can be performed in one of the three modes described in Table 2.1. The programmed I/O mode is slow and involves the CPU in an I/O operation. Therefore only one I/O operation can be performed at a time. The interrupt mode is also slow a byte-by-byte transfer of data is performed, however it frees the CPU between byte transfers. The direct memory access (DMA) mode can transfer a block of data

between the memory and an I/O device without involving the CPU. The interrupt and DMA modes permit several I/O operations to be performed simultaneously.

Table 2.1 Input-Output modes

I/O mode	Description
Programmed I/O	Operands of an I/O instruction indicate details of an I/O operation. CPU decodes the instruction and sends a signal to an I/O device. Data transfer between the I/O device and the memory takes place through the CPU. CPU cannot perform any other operation while I/O is in progress.
Interrupt I/O	CPU executes an I/O instruction. It starts an I/O operation and frees the CPU to execute other instructions. An interrupt is raised when a data byte is to be transferred between the I/O device and the memory. The CPU executes an interrupt processing routine that transfers the byte.
Direct Memory Access (DMA)	An I/O instruction indicates the I/O operation to be performed and also the number of bytes to be transferred. I/O operation starts when the instruction is executed. Data transfer between the device and memory takes place over the system bus. The CPU is not involved in data transfer. An interrupt is raised when the transfer of all bytes is complete.

The DMA operations are performed by a special purpose processor that is dedicated to performing I/O operations. Many variants of the DMA controller are used in different architectures, hence different terms are used for it. We will use the generic term DMA. Figure 2.2 illustrates the I/O subsystem using a DMA. One or more device controllers are connected to the DMA. I/O devices are connected to device controllers. The DMA transfers data between an I/O device and the memory without involving the CPU. When an I/O instruction is executed, say a *read* instruction on device *d*, CPU transfers details of the I/O operation to the DMA. The CPU is not involved in the I/O operation beyond this point; it is free to execute instructions while the I/O operation is in progress. The DMA initiates the *read* operation on device *d* and the data transfer between device *d* and the memory takes place over the bus. Thus the CPU and the I/O subsystem can operate concurrently. At the end of the I/O operation the DMA generates an *I/O interrupt*. The interrupt hardware switches the CPU to execution of the kernel, which processes the interrupt and realizes that the I/O operation is complete.

Interrupts

The function of an interrupt is to draw the kernel's attention to a condition or an event that has occurred in the system. Computer hardware associates a numeric priority with each interrupt. If conditions or events corresponding to several interrupts arise

at the same time, the interrupt mechanism allows the highest priority interrupt to occur. A unique *interrupt code* is associated with each interrupt. This code provides sufficient information concerning the condition or the event so that the kernel can take appropriate actions. The term 'interrupt processing' is used to describe execution of such actions.

Interrupt processing requires the CPU to be diverted from whatever computation it is engaged in executing to execution of an interrupt processing routine in the kernel. The interrupt mechanism achieves this effect by saving contents of the PSW in memory, and loading new contents in the PSW that direct the CPU to execution of the interrupt processing routine. The kernel determines the cause of the interrupt and takes appropriate actions. At the end of interrupt processing it either resumes execution of the interrupted program by loading back the saved PSW contents, or diverts the CPU to execution of another program.

Table 2.2 Classes of interrupts

Class	Description
Program interrupt	Caused by conditions within the CPU that require supervisor attention, for example, arithmetic exceptions like overflow and loss of precision, addressing exceptions and memory protection violations. An interrupt called a <i>software interrupt</i> is caused by execution of a special instruction called the software interrupt instruction.
I/O interrupt	Caused by conditions like I/O completion and malfunctioning of I/O devices.
Timer interrupt	Raised by the interval timer of the computer system.

Classes of interrupts Table 2.2 describes three classes of interrupts that are important during normal operation of an OS. A program interrupt arises due to execution of an instruction and typically indicates an exceptional condition like an arithmetic condition or a memory protection violation. Most CPUs provide a special instruction whose sole purpose is to raise a program interrupt. We assume a hypothetical instruction called software interrupt instruction with the operation-code SI, and call the interrupt raised by it a *software interrupt*. As described later, a program uses the software interrupt instruction to make requests to the kernel.

Interrupt masking The *interrupt mask* field (IM) of the PSW indicates which interrupts are permitted to occur at any moment of time. The field may contain an integer m , in which case only interrupts with priority $\geq m$ are permitted to occur. Alternatively, the IM field may contain a bit encoded value, each bit indicating whether a specific kind of interrupt is permitted to occur. Interrupts that are permitted to occur are said to be *enabled*, others are said to be *masked* (or *masked off*). Masked

interrupts remain pending until they are permitted to occur.

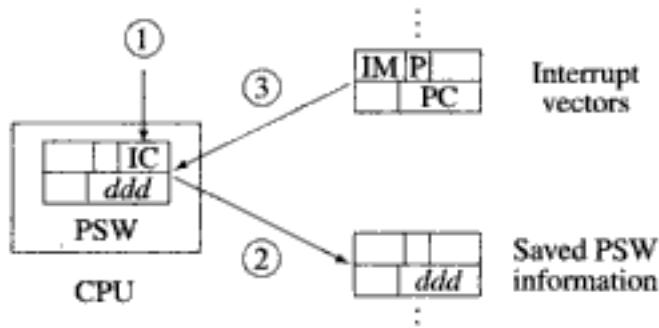


Fig. 2.7 The interrupt action

Interrupt action The purpose of the interrupt action is to transfer control to an appropriate routine in the kernel. Figure 2.7 contains a schematic of the interrupt action. Two areas of memory called the *interrupt vectors* area and the *saved PSW information* area participate in these actions. Each unit in the *interrupt vectors* area is an interrupt vector that controls processing of one class of interrupts. It contains the following information:

1. Address of an interrupt processing routine.
2. An interrupt mask indicating whether other interrupts can be permitted to occur while this interrupt is being processed.
3. Whether the CPU should be in privileged mode while processing the interrupt.

For simplicity it is assumed that an interrupt vector has the same format as a PSW and contains these above items of information in the program counter (PC), interrupt mask (IM) and privileged mode (P) fields, respectively. The interrupt vectors are initialized to appropriate values while booting the system. Each unit in the *saved PSW information* area stores information that is copied from the PSW when an interrupt occurs. In some computer systems, a fixed memory area is associated with each kind of interrupt to store the saved PSW information, while in some other systems the saved PSW information is pushed on a stack (see Problem 7).

The CPU checks for occurrence of an interrupt at the end of each instruction cycle. If an interrupt has occurred, the CPU performs the interrupt action for it. The interrupt action concerns the PSW, a unit from the *saved PSW information* area, and an interrupt vector from the *interrupt vectors* area. The action consists of three steps marked ① ③ in Figure 2.7 and described in Table 2.3. At the end of the interrupt action, the interrupt code that describes the cause of the interrupt has been stored as a part of the saved PSW information and information from the relevant interrupt vector has been loaded into various fields of the PSW.

Subsequent actions of the CPU depend on new contents of the PSW, specifically the fields *privileged mode*, *interrupt mask* and *program counter*. The interrupt vector

Table 2.3 Steps in interrupt action

Step	Description
1. Set Interrupt code	The interrupt hardware for the chosen interrupt forms a code describing the cause of the interrupt. This code is stored in the <i>interrupt code</i> field of the PSW. For example, for an ‘I/O completion’ interrupt, the code could be the address of the I/O device causing the interrupt.
2. Save the PSW	The PSW is copied into the <i>saved PSW information</i> area in the unit corresponding to the interrupt class.
3. Load interrupt vector	The interrupt vector corresponding to the interrupt class is accessed. Information from the interrupt vector is loaded into the corresponding fields of the PSW. This action transfers control to an appropriate interrupt processing routine.

contains the address of the routine that is supposed to handle the interrupt, so the interrupt action effectively transfers control to the appropriate routine in the kernel. Note that the interrupt action must put the CPU in the privileged mode if the processing of an interrupt requires use of privileged instructions. This is achieved by putting 1 in the *privileged mode* field of the interrupt vector. Only interrupts enabled in the new interrupt mask can occur after the interrupt action has been performed, others remain pending until they are enabled by changing the interrupt mask.

2.1.2 OS Interaction with Computer Hardware and User Programs

For good CPU efficiency, the CPU must execute instructions of user programs most of the time. However, the CPU should be switched to execution of the kernel when the kernel needs to perform a control function. This requirement is met by using an arrangement in which any situation that requires the kernel’s attention leads to an interrupt. Interrupt vectors for various interrupts are formed such that their PC fields point to start instructions of appropriate kernel routines. At an interrupt, a kernel routine gains control and performs the required control function. Thus operation of the kernel is *interrupt-driven*.

Figure 2.8 shows actions performed by the kernel when an interrupt occurs. Interrupt processing involves saving the CPU state so that the program that was being executed at the time of the interrupt can be resumed in future, and performing control actions in response to the interrupt. After processing the interrupt, the kernel selects a program for execution and passes control to it. This action is called *scheduling*. Depending on the situation that caused the interrupt, the kernel may select the program that was executing when the interrupt occurred or choose some other program.

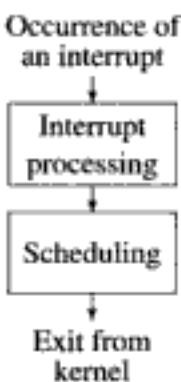


Fig. 2.8 Interrupt-driven operation of a kernel

The interrupt-driven operation of the kernel requires that a program should be able to cause an interrupt to attract the kernel's attention, e.g., when it requires more memory, or when it wishes to start an I/O operation. A *system call* is a generic term used for such an arrangement. Whenever a user program requires some assistance from the OS, it performs a system call by executing a software interrupt instruction. This call leads to an interrupt and the kernel responds to the user's request. Section 2.1.2.2 discusses system calls.

The interrupt processing and scheduling actions of the OS consume resources of the computer system, particularly CPU time. *Scheduling overhead* is what we call the CPU time spent in making a scheduling decision; we denote it by σ (the Greek letter sigma).

2.1.2.1 Interrupt Processing

Interrupt processing involves steps to handle the condition that caused an interrupt, and to continue operation of the OS. An interrupt processing routine performs the following actions:

1. Save contents of CPU registers for use when the program is scheduled again. (This action is not necessary if the interrupt action saves CPU registers.)
2. Take appropriate actions to handle the situation that caused the interrupt. The *interrupt code* field of the *saved PSW information* unit corresponding to the interrupt contains useful information for this purpose. An interrupt may be caused by three kinds of situations
 - (a) An event like end of an I/O operation occurs.
 - (b) The program executing on the CPU makes a request to the OS.
 - (c) An error condition like protection violation occurs.

The interrupt processing routine invokes modules of the kernel that handle the event, satisfy the program's request or terminate the erring program.

3. Pass control to the scheduler to switch the CPU to a user program.

Scheduling The *scheduler* is the kernel component responsible for deciding which program should be executed on the CPU. When it receives control, the scheduler selects a user program according to the prevailing policy for scheduling, obtains the CPU state of the selected program (recall that the CPU state was saved when the program's execution was interrupted sometime in the past), which consists of the contents of CPU registers and the PSW, and loads it into the CPU. This action switches the CPU to the selected program.

Example 2.3 illustrates interrupt processing when an I/O interrupt occurs.

Example 2.3 Figure 2.9 shows interrupt processing routines and interrupt vectors used in a kernel. The user program is about to execute the instruction with the address *ddd* when an I/O interrupt occurs. Contents of the PSW are saved in the *saved PSW information* area. These include the '0' in the privileged mode field, *ddd* in the PC field, and the interrupt code in the IC field. The interrupt vector for the I/O completion interrupt contains *bbb* in the field corresponding to the PC field of the PSW and '1' in the field corresponding to the *privileged mode* field. The contents of this interrupt vector are loaded into the PSW. Effectively, control is transferred to the routine with the start address *bbb*. The P field of the PSW now contains '1', so the CPU is put in the privileged mode.

The interrupt processing routine first saves contents of the CPU registers. It then queries the IC field of the *saved PSW information* unit to find the address of the I/O device which has completed its operation, and performs the necessary control functions. At end, it transfers control to the scheduler. If the scheduler selects the interrupted program itself for execution, it would switch the CPU to execution of the program by loading back contents of CPU registers and loading the saved PSW. The program would resume execution at the place where it was interrupted (see Example 2.1).

Nested interrupt processing Figure 2.10(a) illustrates interrupt processing actions of Example 2.3 if interrupt routine 'a' handles the interrupt and scheduler selects the interrupted program itself for execution. If another interrupt occurs while interrupt routine 'a' is processing the first interrupt, identical actions would occur in the hardware and software, so the CPU would be switched to execution of another interrupt routine, say interrupt routine 'b' (see Figure 2.10(b)). This situation delays processing of the first interrupt, and it also requires careful coding of the kernel to avoid a mix-up if the same kind of interrupt were to arise again (also see Problem 7). These consequences can be avoided by masking off all interrupts using the interrupt mask (IM) field in the interrupt vector (see Figure 2.9). However, masking of an interrupt delays OS response to the condition which caused it. Therefore, to avoid delays in interrupt processing, the kernel defines the interrupt mask in each interrupt vector to mask off only less critical interrupts. More critical interrupts would be processed in a nested manner.

Program preemption Preemption of a program occurs when an interrupt arises during its execution and the scheduler selects some other program for execution. The PSW and the CPU registers are saved by the interrupt processing routine, so

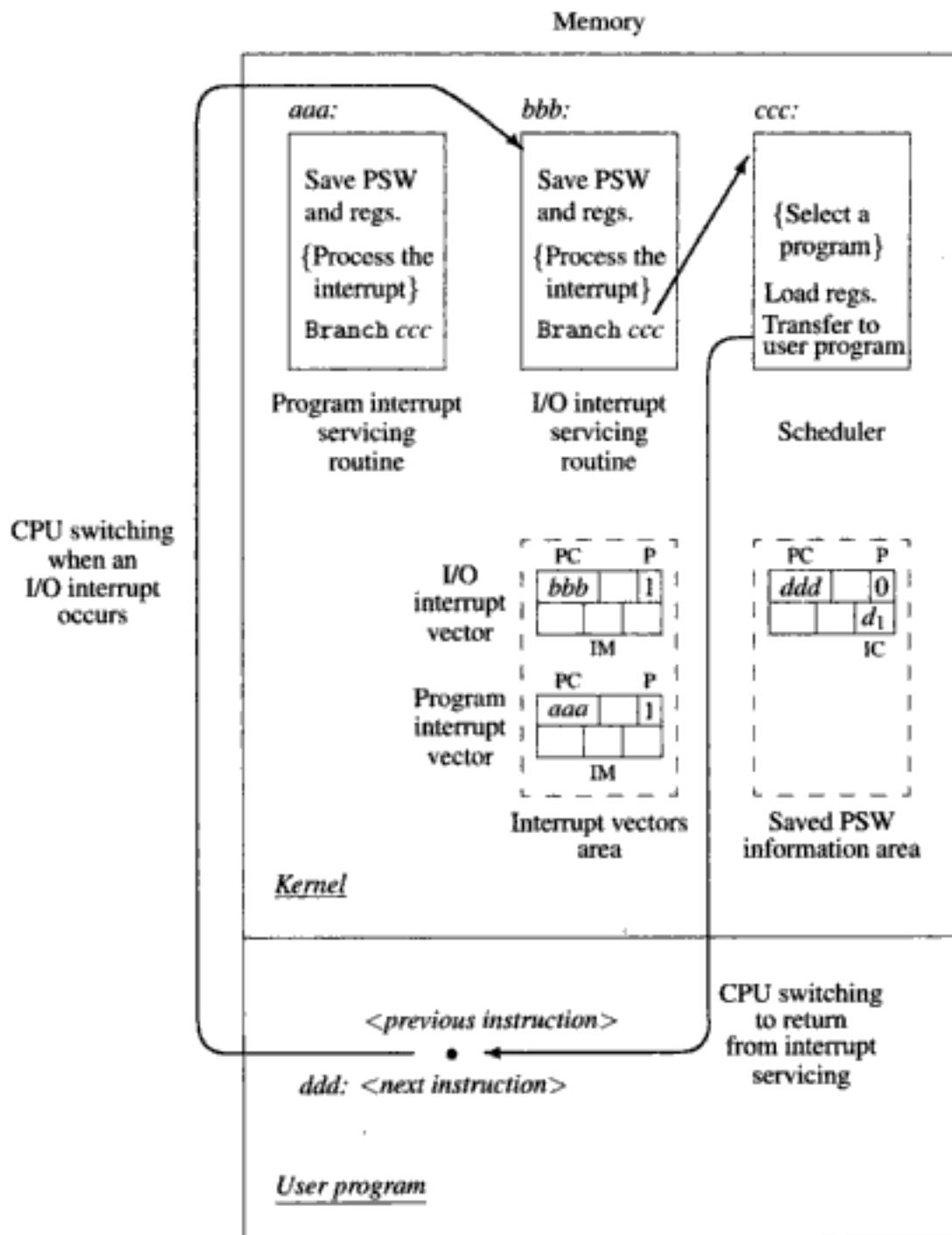


Fig. 2.9 Servicing of an I/O interrupt and return to the same user program

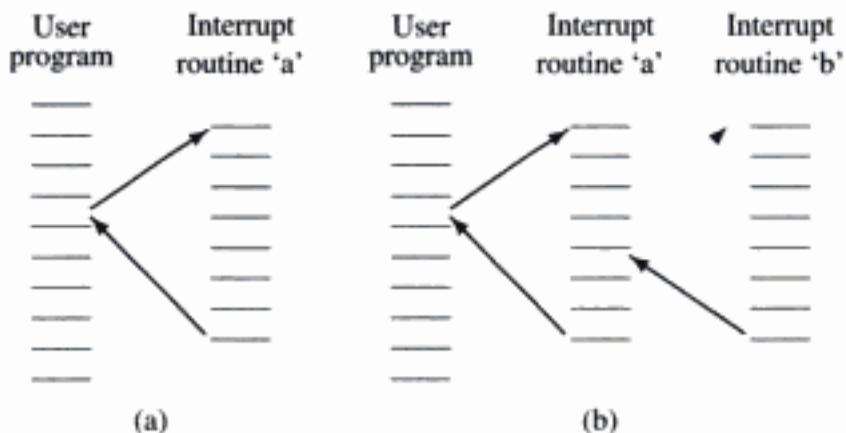


Fig. 2.10 Simple and nested interrupt processing

there is no difficulty in resuming its execution when scheduled again.

2.1.2.2 System Calls

Definition 2.1 (System call) *A system call is a request that a program makes to the operating system.*

A system call uses the special instruction called software interrupt instruction described in Section 2.1.1. This instruction has the format

$\langle SI.instrn \rangle \quad \langle int_code \rangle$

where values of $\langle int_code \rangle$ have standard meanings in the OS. A program interrupt occurs when a program executes this instruction. Step 1 of the interrupt action copies $\langle int_code \rangle$ from the SI instruction into the interrupt code field of the PSW. Step 2 copies the PSW into the *saved PSW information* area. The interrupt processing routine analyses this code to determine the request made by the program. In most computer systems $\langle int_code \rangle$ is at least 8 bits in size, so upto 256 distinct requests can be defined. Example 2.4 describes execution of a system call to obtain the current time.

Example 2.4 A computer system contains a real time clock. Access to the clock is provided through privileged instructions, hence a system call is provided for obtaining the current time. Let the code for this time-of-day service be 10. When a program wishes to know the time, it executes the instruction

$\langle SI.instrn \rangle \quad 10$

which causes a software interrupt. 10 is now entered in the interrupt code field of the PSW before the PSW is saved. As shown in Figure 2.9, the interrupt vector for the software interrupt contains *aaa* in its PC, so control is passed to the routine with the start address *aaa*. The routine analyses the interrupt code and finds that the program wishes to know the time of the day. According to the conventions defined in the OS, the time information is to be returned to the program in a standard location, typically

in a data register. The interrupt routine takes the necessary actions for this purpose and passes control to the scheduler. The scheduler selects a user program and switches the CPU to its execution.

If a computer system does not provide an operand in the SI instruction, a program pushes a code corresponding to the system call being made on the stack before executing the SI instruction. The kernel analyzes the code and performs the necessary action. In Example 2.4, the program pushes 10 on the stack before executing the SI instruction.

Table 2.4 indicates some generic types of system calls and lists a few examples of each type. Resource related calls provide resource allocation and deallocation services. If the requested resource is available, the resource allocation request is honored straightaway; otherwise, the requesting program's execution is delayed until the resource becomes available. Therefore, a program may wish to check for availability of a resource before making a request. Program related calls provide services to start or terminate execution of other programs, and to wait for a certain period of time. Communication related calls set up communication with other programs in the system and realize exchange of messages. Apart from these generic types, each OS provides a host of specialized system calls.

Table 2.4 System calls

Type	Examples
Resource	Allocate/deallocate resource, check resource availability
Program	Set/await timer interrupt, execute/terminate program
File	Open/close a file, read/write a file
Information	Get time and date, get OS information, get resource information
Communication	Send/receive message, setup/terminate connection

2.2 EFFICIENCY, SYSTEM PERFORMANCE AND USER CONVENIENCE

In Chapter 1 we discussed aspects of efficient use of a computer system and user convenience provided by an operating system. We also introduced three fundamental computational structures—a single program, a sequence of single programs and co-executing programs—to illustrate how an OS can provide user convenience. In this Section, we discuss the nature of computations performed in an operating system and describe some metrics that are used to measure efficiency, system performance and user convenience. The concepts and terms introduced here will be useful in later sections of this chapter, and in the following chapters.

2.2.1 Nature of Computations in an OS

In a non-interactive environment, a computation is realized by submitting it to the OS for processing and obtaining its results at the end of its processing. In the classical

non-interactive environment, computations were classified into programs and jobs. A *program* is a set of functions or modules, including some obtained from libraries. A *job* is a sequence of single programs (see Section 1.3.1 and Example 1.1). It consists of a sequence of *job steps*, where each job step constitutes execution of one program. Thus, a job for compiling and executing a C++ program consists of three job steps to compile, link and execute the program. It is not meaningful to execute a job step unless each of the previous job steps has executed successfully, e.g., linking is not meaningful unless compilation was successful. The notion of a job is of less relevance in interactive computing environments because a user typically submits one command at a time to the command processor.

A *process* represents an execution of a program. We illustrate some advantages of processes when we discuss real time systems in Section 2.7. In the interest of simplicity, we defer discussion of other features of processes until Chapter 3.

In an interactive environment, a user interacts with his program. One interaction consists of presentation of a computational requirement by the user to a program—we call this a *subrequest*—and the computation of a response by the program. Depending on the nature of a subrequest, the response may be in the form of a result or it may be an action performed by the program, e.g., a data-base update.

Table 2.5 Computations in an OS

Computation	Description
Program	A <i>program</i> is a set of functions or modules, including some functions or modules obtained from libraries.
Job	A <i>job</i> is a sequence of job steps, where each job step is comprised of the execution of one program. It is not meaningful to execute the program in a job step unless programs in the previous job steps were successfully executed (see Section 1.3.1).
Process	A <i>process</i> is an execution of a program.
Subrequest	A <i>subrequest</i> is the presentation of a computational requirement by a user to a process. Each subrequest produces a single response, which is either a set of results or a set of actions.

Table 2.5 describes the job, program, process and subrequest computations. Notions of system performance and user convenience in an operating system depend on the nature of computations. We discuss this issue in the next section.

2.2.2 Measuring Efficiency, System Performance and User Convenience

Measurement provides a quantitative basis for comparing alternative techniques used in the design and implementation of an OS. However, several practical difficulties arise in making quantitative comparisons. Some attributes of an OS may be intangible hence numerical comparisons may not be possible, or techniques may possess different attributes hence comparisons may not be meaningful.

We encounter both difficulties when we consider efficiency of use and user convenience in different classes of operating systems. Some features like user friendly interfaces provide intangible conveniences, and different computing environments use different notions of efficiency and user convenience. However the measures summarized in Table 2.6 are useful to understand the impact of a technique on operating systems of a specific class. We use these measures in the following sections.

Table 2.6 Measures of efficiency and user convenience

Attribute	Concept	Description
Efficiency of use	CPU efficiency	Percent utilization of the CPU.
System performance	Throughput	Amount of 'work' done per unit time.
User service	Turn around time	Time to complete a job or process.
	Response time	Time to implement one interaction between a user and his/her process.

Efficiency and system performance Although good efficiency of use is important, it is rarely a design goal of an operating system. Good performance in its computing environment is an important design goal. Efficiency of use may be a means to this end, for example, efficiency of resources like the CPU and disks are important parameters for fine-tuning the performance of a system. In Chapters 6 and 4, we see instances of such usage in the context of virtual memories and scheduling.

System performance is typically measured as throughput.

Definition 2.2 (Throughput) *The throughput of a system is the number of jobs, programs, processes or subrequests completed by it per unit time.*

The unit of work used in computing throughput depends on the nature of the computing environment. In a non-interactive environment, throughput of an OS is measured in terms of number of jobs, programs or processes completed per unit time. In an interactive environment, throughput may be measured in terms of the number of subrequests completed per unit time. In a specialized computing environment, performance may be measured in terms meaningful to the application being serviced, e.g., the number of transactions in a banking environment. Throughput can also be used as a measure of performance for I/O devices. For example, the throughput of a disk can be measured as the number of I/O operations completed per unit time or the number of bytes transferred per unit time.

User service User service is a measurable aspect of user convenience. It indicates how a user's computation has been treated by the OS. We define two measures of user service—turn-around time and response time. These are used in non-interactive and interactive computing environments, respectively.

Definition 2.3 (Turn-around time) *The turn-around time of a job, program or process is the time since its submission for processing to the time its results become available to the user.*

Definition 2.4 (Response time) *The response time provided to a subrequest is the time between the submission of the subrequest by a user and the formulation of the process's response to it.*

2.3 CLASSES OF OPERATING SYSTEMS

2.3.1 Key Features of Different Classes of Operating Systems

Table 2.7 summarizes key features of five fundamental classes of operating systems.

Table 2.7 Key features of classes of operating systems

OS class	Period	Prime concern	Key concepts
Batch Processing	1960s	CPU idle time	Spooling, command processor
Multiprogramming	1970s	Resource utilization	Program priorities, preemption
Time sharing	1970s	Good response time	Time slice, round-robin scheduling
Real time	1980s	Meet the deadline	Real time scheduling
Distributed	1990s	Resource sharing	Transparency, distributed control

The *period* column shows the period when operating systems of that class first came into widespread use. Interestingly, key features of the early OS classes can be found in today's operating systems as well! In fact, that is why we study them today. The *prime concern* column of Table 2.7 shows the fundamental effectiveness criterion that motivated development of that OS class. The *key concepts* column indicates concepts that were evolved to help in implementing the prime concerns of an OS.

Batch processing and multiprogramming systems A cursory look at the prime concerns in different classes of operating systems reveals an interesting trend. The first two OS classes, viz. batch processing and multiprogramming, were concerned with efficient use of a computer system. In batch processing, the concern was limited to CPU efficiency, so avoiding wastage of CPU time was the prime concern. In multiprogramming, the concern was broadened to include other resources, mainly memory and the I/O subsystem, so a balanced utilization of all resources in the system was the prime concern. These concerns arose from the high cost of hardware in those days, and satisfying these concerns did not provide any direct benefits to users.

Time sharing and real time systems In mid-seventies, the focus shifted from efficient use of a computer system to productivity of computer users. This shift was necessitated by the clamor for efficient service by users, and it became practical due to reducing costs of hardware. Time sharing systems focused on providing a quick

response to a subrequest. This focus provided tangible benefits to computer users. About the same time, applications of computers in time-critical environments led to the development of real time OSs that focused on completing a computational task before the deadline specified by an application. Due to the need to meet deadlines, a real time system is often dedicated to a single time-critical application.

Distributed systems In the 1990s, declining hardware costs led to the development of distributed systems that consisted of several computer systems with varying number and sophistication of resources. A distributed system permits resource sharing across the boundaries of individual computer systems. Thus, a user of a low cost computer system can use expensive resources existing in other computer systems. Some features of resource sharing resemble efficient use of resources. However, it possesses other important features like fault tolerance that have much to do with user convenience.

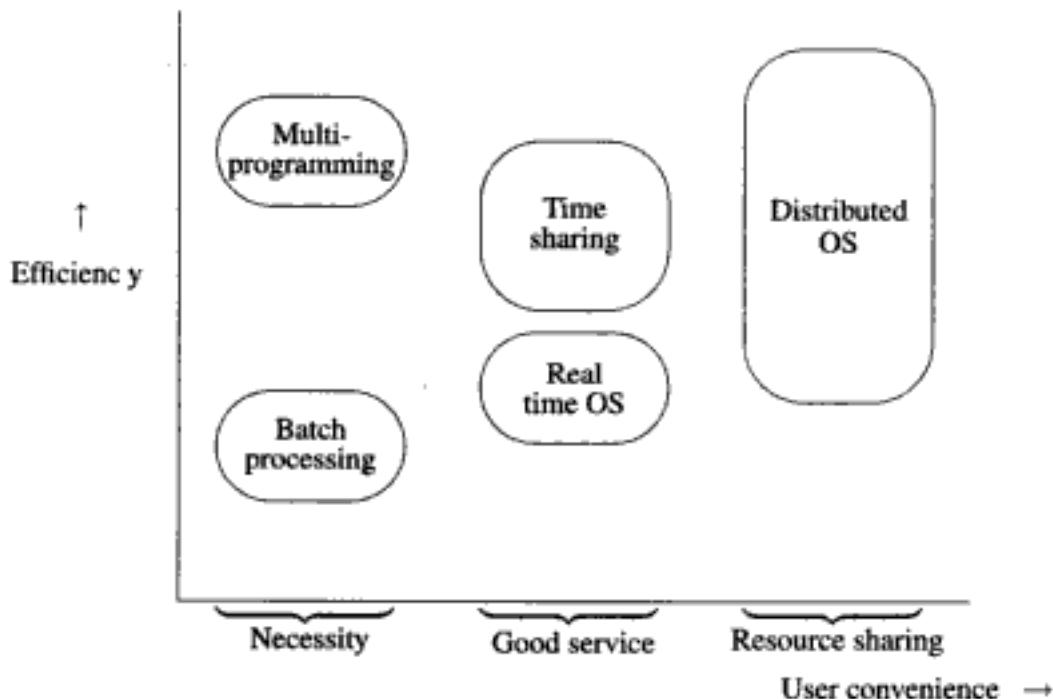


Fig. 2.11 Efficiency and user convenience in different OS classes

2.3.2 Efficient Use and User Convenience

As discussed in the Introduction, efficient use and user convenience often exert contrary pressures on the design of an OS. However, the trend in OS design has been towards increasing user convenience. Figure 2.11 shows the balance between efficiency and user convenience achieved by different OS classes. Note that neither axis in this plot is homogeneous. The notion of user convenience changes from one OS class to another. Resources whose efficiency has been plotted also change across

OS classes. As discussed in Section 1.2, necessity is at the lowest end of user convenience. Batch processing and multiprogramming operating systems provide this level of user convenience. Batch processing provides low efficiency because it processes one user program at a time. Multiprogramming provides higher efficiency by processing many user programs in an interleaved manner over the same duration of time.

A time sharing system provides good service, which is rated higher on the scale of user convenience. A time sharing system processes many programs over the same period of time, hence it can provide better efficiency than pure batch processing. However, efficiency may be lower than that of multiprogramming systems due to higher overhead of OS operation. A real time system may provide lower efficiency than a time sharing system because it may be dedicated to a single application.

A distributed OS provides resource sharing across the boundaries of computer systems, so efficiency provided by it covers a wider range than any other operating system. At the lower end, low cost computing may resemble batch processing in many ways. The resources available in a computer system may be adequate only for a small number of programs. Advantages of multiprogramming become available if a computer system under the control of a distributed OS has reasonable computing resources. Further, resource sharing across boundaries of individual computer systems may enhance resource utilization. Therefore, a distributed OS has the potential to provide higher efficiency than a multiprogramming system.

2.3.3 Execution of User Computations in Different OS Classes

Table 2.8 summarizes the manner in which execution of user computations is organized in operating systems of different classes. A batch processing system operates in a strict one-job-at-a-time manner. Within a job, it executes the programs one after another. Thus only one program is under execution at any time. CPU efficiency is enhanced by efficiently initiating the next job when one job ends.

In a multiprogramming system, several programs are in a state of partial completion at any time. Resources allocated to a program are utilized when the program is executed. The OS switches the CPU between execution of different programs to ensure a balanced utilization of resources. It uses the notion of priority to decide which program should be executed on the CPU at any time.

A time sharing system also interleaves execution of processes. However, the purpose of interleaving is to provide good response times to *all* processes. Hence a time sharing OS does not assign priorities to processes; it provides fair execution opportunity to each process.

A real time system permits a user to create several processes *within* an application program and assign them execution priorities. It interleaves the execution of processes to meet the deadline of the application.

A distributed OS provides sharing and efficient use of resources located in all computers in the system. One way to achieve this is to let a process access resources

Table 2.8 Execution of user computations in different operating systems

OS class	Computations	Key execution concept
Batch processing	Jobs	One job is executed at a time. Programs in a job are executed sequentially.
Multiprogramming	Programs	OS interleaves execution of several programs to improve resource utilization and system performance.
Time sharing	Processes	OS interleaves execution of processes to provide good response to all processes.
Real time	Processes	OS interleaves execution of processes in an application program to meet its deadline.
Distributed	Processes	Access to remote resources using networking. Execution of processes of an application in different computers to achieve sharing and efficient use of resources.

located in a remote computer using the networking component. However, use of networking causes delays, so the OS may execute processes of a program in different computers to achieve good utilization of resources.

Thus, the five fundamental OS classes cater for different notions of effectiveness, and use different concepts and techniques to meet their design goals. In Sections 2.4–2.8, we discuss features of the fundamental OS classes. New concepts and techniques used in their implementation are defined and illustrated with the help of examples.

2.4 BATCH PROCESSING SYSTEMS

Computer systems of the 1960's used punched cards as the primary input medium. A program and its data consisted of a deck of cards. A programmer would submit a program at a counter in the computer center. The computer center staff would handle the card decks and eventually the execution of a program would be set up on the computer by a computer operator. The operator too had to handle the card decks and perform physical actions like loading them into the card reader and pressing some switches on the console to initiate processing of a program. The CPU of the computer system was idle while the operator performed these actions. A lot of CPU time was wasted in this manner. Batch processing was introduced to avoid this wastage.

A *batch* is a sequence of user jobs formed for the purpose of processing by a batch processing operating system. Note that a batch is not a computational structure of a user. Each job in the batch is independent of other jobs in the batch; jobs typically belong to different users. A computer operator forms a batch by organizing a set of user jobs in a sequence and inserting special marker cards to indicate the start and end of the batch. The operator submits a batch as a unit of processing by the batch

processing operating system. The primary function of the batch processing system is to service the jobs in a batch one after another without requiring the operator's intervention. This is achieved by automating the transition from execution of one job to that of the next job in the batch.

Batch processing is implemented by the kernel (also called the *batch monitor*), which resides in one part of the computer's memory. The remaining memory is used for servicing a user job—the current job in the batch. When the operator gives a command to initiate the processing of a batch, the batch monitor sets up the processing of the first job of the batch. At the end of the job, it performs job termination processing and initiates execution of the next job. At the end of the batch, it performs batch termination processing and awaits initiation of the next batch by the operator. Thus the operator needs to intervene only at the start and end of a batch.

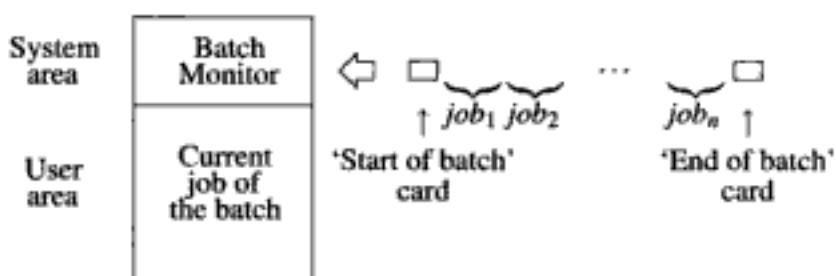


Fig. 2.12 Schematic of a batch processing system

Figure 2.12 shows a schematic of a batch processing system. The batch consists of n jobs, $job_1, job_2, \dots, job_n$, one of which is currently in execution. The figure depicts a *memory map* showing the arrangement of the batch monitor and the current job of the batch in the computer's memory. The part of memory occupied by the batch monitor is called the *system area*, and the part occupied by the user job is called the *user area*.

Batch processing systems used the notion of virtual devices described in Section 1.3.2.1 to conserve the CPU time of a powerful computer system as follows: The computer system used virtual devices for input and output—typically tapes or disks instead of punched cards and printers. To make this feasible, a program first recorded a batch of jobs on a magnetic tape. The batch processing system processed these jobs and wrote their results on another magnetic tape. The contents of this magnetic tape were then printed by another program. The two spooling operations, called *inspooling* and *outspooling*, respectively, were typically performed on a smaller computer system having a slow CPU. The main computer used the faster magnetic tape media hence it suffered smaller CPU idle times. Results of a job were not released to a user immediately after the job was processed; they were released only after printing was completed.

2.4.1 User Service

The notion of turn-around time is used to quantify user service in a batch processing system. Due to spooling, the turn-around time of a job job_i processed in a batch processing system includes the following time intervals:

1. Time until a batch is formed (i.e., time until the jobs job_{i+1}, \dots, job_n are submitted)
2. Time spent in executing all jobs of the batch
3. Time spent in printing and sorting the results belonging to different jobs.

Thus, the turn-around time for job_i is a function of many factors, its own execution time being only one of them. It is clear that use of batch processing does not guarantee improvements in the turn-around times of jobs. In fact, the service to individual users would probably deteriorate due to the three factors mentioned above. This is not surprising because batch processing does not aim at improving user service—it aims at improving CPU utilization.

Example 2.5 Figure 2.13 illustrates different components in the turn-around times of a job. The user submits the job at time t_0 . However, the batch is not formed immediately. The operations staff of the computer center form a batch only after a sufficient number of jobs have been submitted. Consequently, the batch actually gets formed at time t_1 . Its processing starts at time t_2 and ends at t_3 . Printing of the results is commenced at time t_4 and completes at t_5 . The results are returned to the user only at time t_6 . Thus the turn-around time of the job is $(t_6 - t_0)$. It has no direct relation to its own execution time.

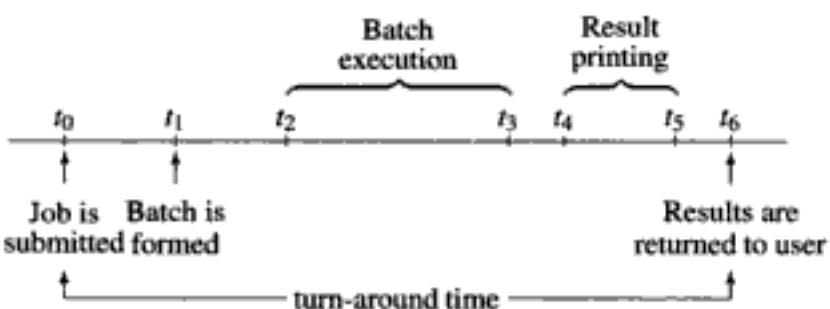


Fig. 2.13 Turn around time in a batch processing system

2.4.2 Batch Monitor Functions

To exercise effective control over the batch processing environment, the batch monitor performs three functions—scheduling, memory management, and sharing and protection. The first two functions are trivial. Scheduling is implicit in formation of a batch, while memory allocation is performed at system boot time by allocating all memory not used by the OS to the processing of user jobs. The memory is shared sequentially by user jobs.

The protection function is more complex. User jobs cannot interfere with each other's execution directly because they never coexist in a computer's memory. However, contrary to one's intuition, even sequential sharing of resources can lead to loss of protection. This problem is discussed in the following.

Protection problems in card based systems As mentioned at the start of this Section, early batch processing systems were card-based systems of the 1960's. These systems possessed few secondary memory units like tapes and disks, so commands, user programs, and data were all derived from a single input source—the card reader. This feature could lead to interference between consecutive jobs in a batch. Example 2.6 illustrates how this can happen.

Example 2.6 A and B are consecutive jobs in a batch. Each job consists of a program and its data. The program of job A requires ten data cards whereas the user who submitted job A has included only five data cards in it. When job A executes, it reads its five data cards and the first five cards of job B as its own data. When job B is processed, the compiler finds many errors since the first five cards of B are missing. Thus job A has interfered with the execution of job B.

Control statements and the command processor A batch processing system requires a user to insert a set of control statements in a job. The control statements are used for two purposes—to implement a job as a 'sequence of programs' (see Section 1.3.1), and to avoid mutual interference between jobs. Figure 2.14 shows the control statements used to compile and execute a Pascal program. For simplicity statements concerning data-set definitions or device assignments that were necessary in most batch processing systems are omitted.

The // JOB statement indicates the start of a job. It contains accounting information for the user like his name, user category, and resource limits. An // EXEC <program_name> statement, where <program_name> is the name of a program to be executed, constitutes a job step. Cards containing the data for <program_name> follow this statement. The statement /* marks the end of data. Thus, the statement // EXEC PASCAL in Figure 2.14 indicates that the Pascal compiler is to be executed to compile a Pascal program. The Pascal program to be compiled is placed between this statement and the first /* statement. The // EXEC statement without a program name in Figure 2.14 indicates that the just-compiled program should be executed. The second /* statement indicates the end of data for the program, and the '/&' statement indicates the end of the job.

A control statement is processed by the *command processor* component of the batch processing kernel. The command processor reads a control statement, analyses it and carries out the required action. It also checks for situations that might lead to interference between jobs. At a // JOB statement, it verifies the validity of the user's accounting information and initializes its own data bases to note that the processing of a new job has started. At an // EXEC statement it organizes loading and execution of the appropriate program.

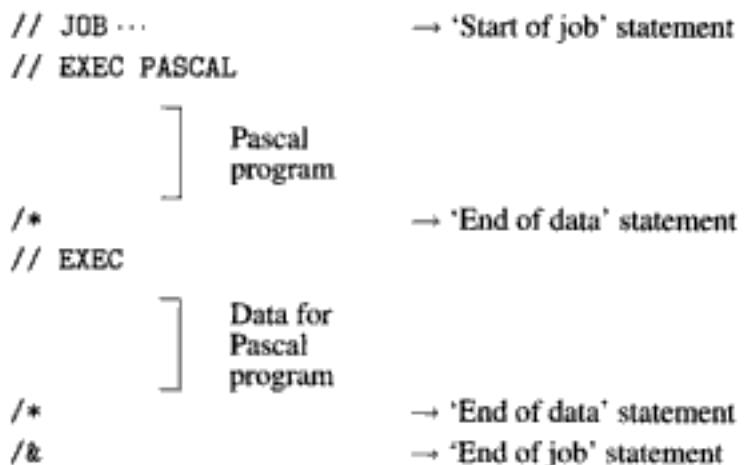


Fig. 2.14 Control statements in IBM 360/370 systems

If a '/*' or '&' statement is encountered during the execution of a program, the command processor realizes that the program has reached the end of its data. If the program tries to read more data cards, it is terminated by skipping to the '&' statement.

The command processor must process each control statement to implement these functionalities. To ensure that the command processor sees each control statement, a user program is not permitted to read cards directly. It must make a request to the kernel to read a card. (This is done through a *system call*—see Section 2.1.2.2.) Control now reaches the command processor, which reads the card and checks whether the card contains a control statement. If so, the command processor initiates abnormal termination of the job; otherwise, it hands over the card to the user program.

2.5 MULTIPROGRAMMING SYSTEMS

Concurrency of operation between the CPU and the I/O subsystem (see Section 2.1.1) can be exploited to get more work done in the system. The OS can put many user programs in the memory, and let the CPU execute instructions of one program while the I/O subsystem is busy with an I/O operation for another program. This technique is called *multiprogramming*.

Figure 2.15 illustrates operation of a multiprogramming OS. The memory contains three programs. An I/O operation is in progress for *program*₁, while the CPU is executing *program*₂. The CPU is switched to *program*₃ when *program*₂ initiates an I/O operation, and it is switched to *program*₁ when *program*₁'s I/O operation completes. The multiprogramming kernel performs scheduling, memory management and I/O management. It uses a simple scheduling policy, which we will discuss in Section 2.5.3.1, and performs simple partitioned or pool-based allocation of memory and I/O devices. (Scheduling, memory management and I/O management are discussed in detail in Chapters 4, 5–6, and 7–12, respectively.) As shown in Figure 2.12, each program in the memory could be a program in the current job of a

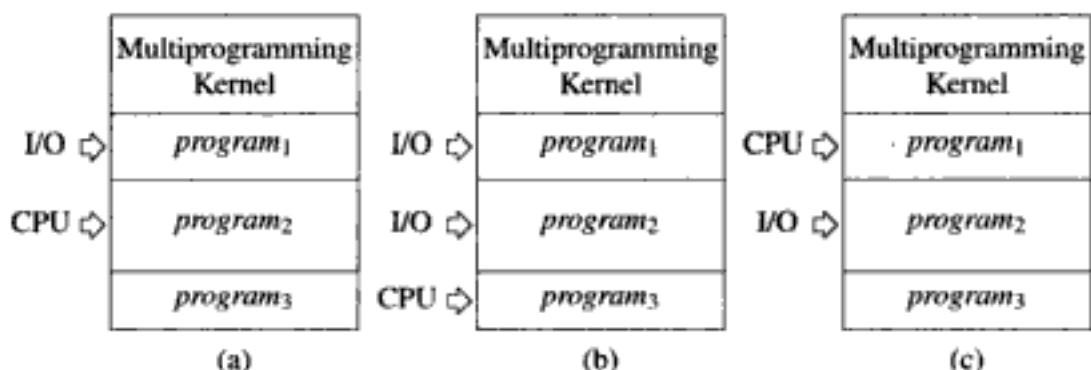


Fig. 2.15 Operation of a multiprogramming system.

batch of jobs. Thus one could have both batch processing and multiprogramming at the same time.

In principle the CPU and the I/O subsystem could operate on the same program. However, in this case the program must explicitly synchronize activities of the CPU and the I/O subsystem to ensure that the program executes correctly. Example 2.7 illustrates the need for synchronization.

Example 2.7 Consider the following program segment

Stmt no.	Statement
1.	read a;
...	
5.	b := a+5;

After initiating an I/O operation to implement the `read` statement, the CPU could continue with execution of the program. However, the value of `a` should not be used in the fifth statement before its reading is completed by the first statement! The CPU and I/O activities in the program are synchronized to ensure that this does not happen.

The multiprogramming arrangement ensures synchronization of the CPU and I/O activities in a simple manner—it allocates the CPU to a program only when the program is not performing an I/O operation.

2.5.1 Architectural Support for Multiprogramming

A computer must possess the features summarized in Table 2.9 to support multiprogramming (see Section 2.1.1). The DMA makes multiprogramming feasible by permitting concurrent operation of the CPU and I/O devices. Memory protection prevents mutual interference between programs. The privileged mode of the CPU provides a foolproof method of implementing memory protection and other measures that avoid interference between programs. For example, instructions to load addresses into the lower bound register (LBR) and upper bound register (UBR) of the CPU are privileged instructions. If a program tries to change contents of the LBR

and UBR by using these instructions, a program interrupt would be raised because the CPU is in the user mode while executing user programs; the kernel would abort the program while servicing this interrupt.

Table 2.9 Architectural support for multiprogramming

Feature	Description
DMA	The CPU initiates an I/O operation when an I/O instruction is executed. The DMA implements the data transfer involved in the I/O operation without involving the CPU, and raises an I/O interrupt when the data transfer completes.
Memory protection	Ensures that a program does not access or destroy contents of memory areas occupied by other programs or the OS.
Privileged mode of CPU	Certain instructions, called <i>privileged instructions</i> , can be performed only when the CPU is in the privileged mode. A program interrupt is raised if a program tries to execute a privileged instruction when the CPU is in the user mode.

Since several programs are in memory at the same time, the instructions, data, and I/O operations of one program should be protected against interference by other programs. It is achieved simply by putting the CPU in the user mode while executing user programs.

2.5.2 User Service

In a multiprogramming system, the turn-around time of a job is affected by the amount of CPU attention devoted to other jobs executing concurrently with it, so the turn-around time of a job depends on the number of jobs in the system, and relative priorities assigned to different jobs by the scheduler. As in a batch processing system, the turn-around time of a job is not directly related to its own execution requirements. The influence of priorities on user service is discussed in Section 2.5.3.1.

2.5.3 Functions of the Multiprogramming Kernel

Important functions of the multiprogramming kernel are:

1. Scheduling
2. Memory management
3. I/O management.

Scheduling is performed after servicing every interrupt (see Figure 2.8). Multiprogramming systems use a simple priority based scheduling scheme described in the next Section. Functions 2 and 3 involve allocation of memory and I/O devices. Simple partitioned or pool-based allocation can be used for this purpose. Resource sharing necessitates protection against mutual interference—the instructions, data, and I/O operations of one program should be protected against interference by other

programs. Two provisions are used to achieve this. Memory protection hardware is used to prevent a program from interfering with memory allocated to other programs, and the CPU is put in the non-privileged mode while executing user programs. Any effort by a user program to access memory locations situated outside its memory area, or to use a privileged instruction, now leads to an interrupt. Interrupt processing routines for these interrupts simply terminate the program that caused an interrupt.

2.5.3.1 Performance of a Multiprogramming System

An appropriate measure of performance of a multiprogramming OS is *throughput*, which is the ratio of the number of programs processed and the total time taken to process them. Throughput of a multiprogramming OS that processes n programs over the period of time that starts at t_0 and ends at t_f is $n/(t_f - t_0)$. This may be larger than the throughput of a batch processing system because activities in several programs may take place simultaneously—one program may execute instructions on the CPU, while some other programs perform I/O operations. However, actual throughput depends on the nature of programs being processed, i.e., how much computation and how much I/O they perform, and how well the kernel can overlap their activities in time. To optimize the throughput, a multiprogramming system uses the concepts and techniques described in Table 2.10. The system keeps a sufficient number of different kinds of programs in memory and overlaps their execution using priority based, preemptive scheduling. We discuss these concepts and techniques in the following.

Table 2.10 Concepts and techniques in multiprogramming

Concept/Technique	Description
Degree of multiprogramming	The number of user programs that the OS keeps in memory at any time.
Program mix	The kernel keeps a mix of CPU-bound and I/O-bound programs in memory, where
	<ul style="list-style-type: none"> • A <i>CPU-bound program</i> is a program involving a lot of computation and very little I/O. It uses the CPU in long bursts—that is, it uses the CPU for a long time before starting an I/O operation.
	<ul style="list-style-type: none"> • An <i>I/O-bound program</i> involves very little computation and a lot of I/O. It uses the CPU in small bursts.
Priority-based and preemptive scheduling	Every program is assigned a priority. The CPU is always allocated to the highest priority program that wishes to use it.
	A low priority program executing on the CPU is preempted if a higher priority program wishes to use the CPU.

Program mix To see why an OS must keep a proper mix of programs in execution, consider a multiprogramming system with $m = 2$. Let $prog_1$ and $prog_2$ be

the two programs under processing. Let us assume that both programs are heavily CPU-bound and $prog_1$ is currently executing. Being CPU-bound, $prog_1$ performs a long burst of CPU activity before performing an I/O operation. CPU is given to $prog_2$ when $prog_1$ initiates an I/O operation. $prog_2$ also performs a long burst of CPU activity before performing an I/O operation. $prog_1$ would have finished its I/O operation by this time, so it can use the CPU for another burst of computation. In this manner the CPU is kept busy most of the time. It remains idle only when both programs simultaneously perform I/O. The I/O subsystem is underloaded. In fact, it is idle most of the time because programs contain long CPU bursts, so periods of concurrent CPU and I/O activities are rare. Consequently, the throughput is low.

To analyze the throughput of this multiprogramming system, we will compare it with a batch processing system processing the same two programs. The batch processing system would service the programs one after another, i.e., either in the sequence $prog_1, prog_2$ or in the sequence $prog_2, prog_1$. The throughput would be identical in both cases. For comparing the throughput of the batch processing and multiprogramming systems, we introduce the notion of progress of a program. A program ‘makes progress’ when either the CPU is executing its instructions or its I/O operation is in progress. Throughput would be higher if several programs make progress concurrently.

In our example, $prog_1$ and $prog_2$ make progress concurrently only when one of them performs I/O and the other executes on the CPU, i.e., only when the CPU and the I/O subsystem execute concurrently. Such periods are rare, so periods when both programs make progress are also rare. Consequently, throughput of the multiprogramming system is likely to be much the same as the batch processing system.

A practical way to improve the throughput is to select a mix of programs that contains some CPU-bound programs and some I/O-bound programs. For example, let a multiprogramming system with $m = 2$ contain the following programs:

$prog_{cb}$:	CPU-bound program
$prog_{io}$:	I/O-bound program

$prog_{cb}$ can keep the CPU occupied while $prog_{io}$ keeps the I/O subsystem busy. Thus, both programs would make good progress, and the throughput would be higher than in a batch processing system.

Program priority We assume a simple implementation scheme in which each program has a numeric priority where a larger value implies higher priority.

Definition 2.5 (Priority) Priority is a tie-breaking notion used in a scheduler to decide which request should be scheduled on the server when many requests await service.

When many programs are ready to use the CPU, the multiprogramming kernel gives the CPU to the program with the highest priority. This rule leads to preemption

of a low priority program when a high priority program becomes ready to use the CPU.

Definition 2.6 (Preemption) Preemption is the forced deallocation of the CPU from a program.

Example 2.8 illustrates how preemption takes place in a multiprogramming system.

Example 2.8 In a multiprogramming system a high priority program is engaged in performing an I/O operation and a low priority program is in execution. As discussed in Example 2.3 and illustrated in Figure 2.9, when an interrupt occurs the CPU is switched to execution of the I/O interrupt processing routine that has the start address *bbb*. After the interrupt processing actions, control is transferred to the scheduler, which selects the high priority program for execution. Effectively, the low priority program that was executing on the CPU has been preempted.

Contrast Example 2.8 with Example 2.3 in the context of a multiprogramming system. In Example 2.3, it was assumed that the program that was executing when an interrupt occurred is scheduled again for execution. This assumption implies that the I/O operation whose completion was signaled by the interrupt must have belonged to some lower priority program. The interrupted program still remains the highest priority program that can use the CPU, so it is selected for execution by the scheduler.

Assignment of program priorities The kernel of the multiprogramming system has to assign priorities to programs. The program mix consists of some CPU-bound programs and some I/O-bound programs, so the kernel has to decide whether CPU-bound programs should have higher priority or whether I/O-bound programs should have higher priority. This is a crucial decision because it can influence system throughput. The priority assignment rule in multiprogramming systems is as follows:

- In multiprogramming systems I/O-bound programs should have higher priority than CPU-bound programs.

To illustrate this rule, and its influence on system throughput, we consider a multiprogramming system containing *prog_{cb}* and *prog_{iob}*. The CPU and I/O activities of these programs are plotted in the form of a timing chart in which the x-axis shows time and the y-axis shows CPU and I/O activities of the two programs (see Figure 2.16). We compare timing charts for different priority assignments to *prog_{cb}* and *prog_{iob}*. To emphasize why I/O-bound programs should have higher priority, we first discuss how a system would perform poorly when a CPU-bound program is given a higher priority.

Higher priority to CPU-bound programs Example 2.9 discusses features of system operation when CPU-bound programs have higher priority than I/O-bound programs.

Example 2.9 The timing chart of Figure 2.16 depicts operation of the system when $prog_{cb}$, the CPU bound program, has higher priority. Note that the chart is not to scale. The CPU activity of $prog_{lob}$ and the I/O activities of both programs have been exaggerated for clarity.

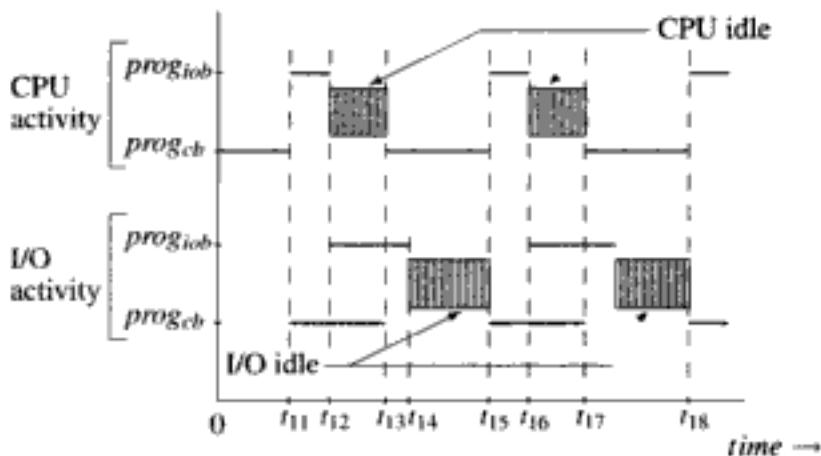


Fig. 2.16 Timing chart when CPU-bound program has higher priority

$prog_{cb}$ is the higher priority program, so it starts executing at time $t = 0$. After a long burst of CPU activity, it initiates an I/O operation (time instant t_{11}). The CPU is now switched to $prog_{lob}$, the I/O-bound program. Processing of $prog_{lob}$ by the CPU is thus concurrent with the I/O operation of $prog_{cb}$. $prog_{lob}$ initiates an I/O operation soon, i.e., at t_{12} . Two I/O operations are now in progress, and the CPU will be idle until one of them completes. Assuming $prog_{cb}$'s I/O to finish earlier, it would start executing on the CPU at t_{13} . Completion of $prog_{lob}$'s I/O at t_{14} makes no difference to the progress of $prog_{cb}$, which continues its execution since it is the higher priority program. Its CPU burst ends at time instant t_{15} when it initiates an I/O operation. The cycle of events of time interval $t_{11}-t_{15}$ can now repeat. Note that the CPU is idle over the intervals $t_{12}-t_{13}$ and $t_{16}-t_{17}$. Similarly the I/O subsystem is idle over the intervals $t_{14}-t_{15}$ and $t_{17}-t_{18}$.

From Example 2.9 one can make the following observations concerning system operation when CPU-bound programs have higher priorities than I/O-bound programs:

1. CPU utilization is reasonable.
2. I/O utilization is poor because the CPU-bound program monopolizes the CPU and the I/O-bound program does not get an opportunity to execute.
3. Periods of concurrent CPU and I/O activities are rare.

From observation 3 it is clear that most of the time only one of the two programs is able to make progress, so the throughput is not much better than in a batch processing system. Attempts to improve the throughput by increasing the degree of multiprogramming meet with little success. For example, let us increase the degree

of multiprogramming (m) to 3 and introduce a CPU-bound program $prog_3$ with a priority between those of $prog_{cb}$ and $prog_{io}$. This action improves the CPU utilization since the new program can keep the CPU busy over intervals where it would have been idle if m was 2 (see interval $t_{12}-t_{13}$). However, it leads to further deterioration of the I/O utilization because $prog_{io}$ receives lesser opportunity to execute. Addition of an I/O-bound program (say, $prog_4$) instead of $prog_3$ makes little difference to the CPU and I/O utilization because $prog_4$ would have the lowest priority, so it does not get much opportunity to execute.

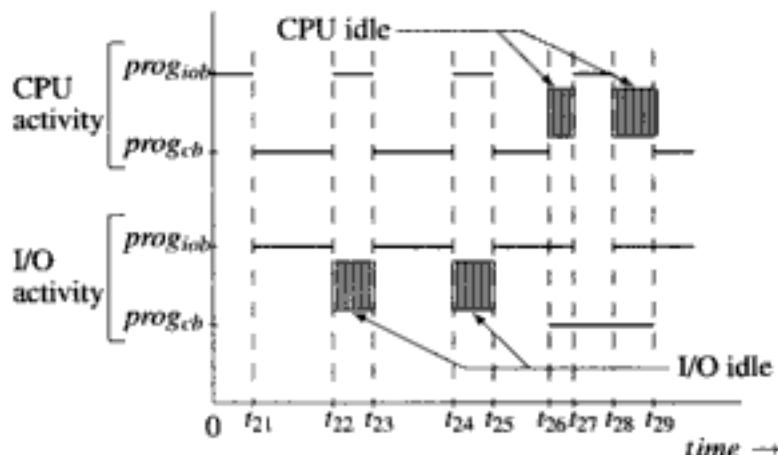


Fig. 2.17 Timing chart when I/O-bound program has higher priority

Higher priority to I/O-bound programs Example 2.10 discusses features of system operation when I/O-bound programs have higher priority than CPU-bound programs.

Example 2.10 Figure 2.17 depicts the operation of the system when the I/O-bound program has higher priority. $prog_{io}$ is the higher priority program, hence it is given the CPU whenever it needs, i.e., whenever it is not performing I/O. When $prog_{io}$ initiates an I/O operation, $prog_{cb}$ gets the CPU. Being a CPU-bound program, $prog_{cb}$ keeps the CPU busy until $prog_{io}$'s I/O completes. $prog_{cb}$ is preempted when $prog_{io}$ completes its I/O since $prog_{io}$ has a higher priority than $prog_{cb}$. This explains the system behavior in the period $0-t_{26}$. Deviations from this behavior occur when $prog_{cb}$ initiates an I/O operation. Now both programs are engaged in I/O and the CPU remains idle until one of them completes its I/O. This explains the CPU-idle periods $t_{26}-t_{27}$ and $t_{28}-t_{29}$. I/O-idle periods occur whenever $prog_{io}$ executes on the CPU and $prog_{cb}$ is not performing I/O (see intervals $t_{22}-t_{23}$ and $t_{24}-t_{25}$).

From Example 2.10, one can make the following observations concerning system operation when I/O-bound programs have higher priorities:

1. CPU utilization is reasonable.
2. I/O utilization is reasonable (however, I/O idling would exist if system contains many devices capable of operating in the DMA mode),

3. Periods of concurrent CPU and I/O activities are frequent.

$prog_{io}$ makes good progress since it is the highest priority program. It makes light use of the CPU, so $prog_{cb}$ also makes good progress. The throughput is thus substantially higher than in the batch processing system. Another important feature of this priority assignment is that system throughput can be improved by adding more programs. Table 2.11 shows how this can be achieved. Hence one can conclude that assignment of higher priorities to I/O-bound programs leads to good throughput. It also enables the OS to combat poor throughput in the system by increasing the degree of multiprogramming.

Table 2.11 Improving throughput in a multiprogramming system

Add a CPU-bound program	A CPU-bound program (say, $prog_3$) can be introduced to utilize some CPU time that is wasted in Example 2.10 (e.g. the intervals $t_{26}-t_{27}$ and $t_{28}-t_{29}$). $prog_3$ would have the lowest priority. Hence its presence would not affect the progress made by $prog_{cb}$ and $prog_{io}$.
Add an I/O-bound program	An I/O-bound program (say, $prog_4$) can be introduced. Its priority would be between the priorities of $prog_{io}$ and $prog_{cb}$. Presence of $prog_4$ would improve I/O utilization. It would also not affect the progress of $prog_{cb}$ very much since $prog_4$ does not use a significant amount of CPU time.

Degree of multiprogramming When a proper program mix is maintained, an increase in the degree of multiprogramming, m , would result in an increase in throughput. Memory capacity and the average memory requirements of user programs determine practical values of m . Figure 2.18 shows how the throughput of a system varies with the degree of multiprogramming.

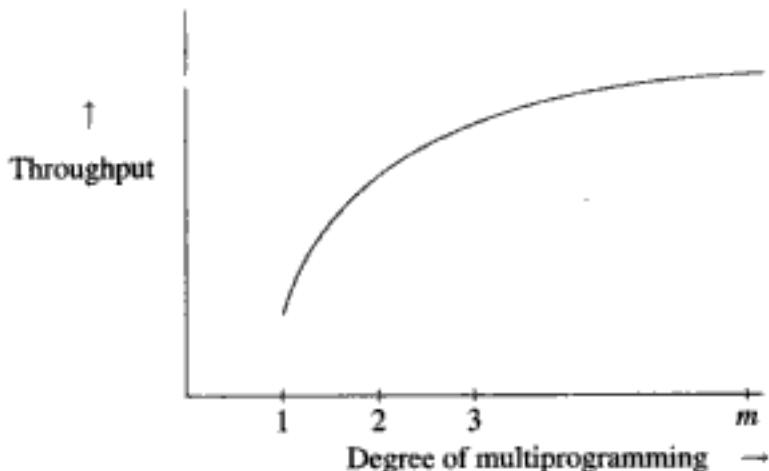


Fig. 2.18 Variation of throughput with degree of multiprogramming

When $m = 1$, the throughput is dictated by the elapsed time of the lone program

in the system. At higher values of m , lower priority programs also contribute to throughput. However their contribution is limited by their opportunity to use the CPU. Throughput stagnates with increasing values of m if low priority programs do not get any opportunity to execute.

2.6 TIME SHARING SYSTEMS

In a non-interactive computing environment, a user has no contact with his program during its execution. In such an environment it is natural to use the batch processing or multiprogramming paradigm. Both these systems provide poor user service. However, this is inevitable due to the nature of I/O devices in use.

In an interactive computing environment, a user can provide inputs to a program from the keyboard and examine its output on the monitor screen. A different OS paradigm is used in such environments to provide quick service to user requests; it creates an illusion that each user has a computer system at his sole disposal. This paradigm is called *time sharing*.

2.6.1 User Service

User service is characterized in terms of the time taken to service a subrequest, i.e., the response time (rt). Benefits of good response times are best seen during program development. A user engaged in program development compiles and tests a program repeatedly. A typical request issued by the user concerns compilation of a statement or an execution of a program on given data. The response consists of a message from the compiler or results computed by a program. A user issues the next request after receiving the response to the previous request. Good response times would lead to an improvement in the productivity of the user. Any application in which a user has to interact with a computation would derive similar benefits from good response times.

Emphasis on good response times, rather than on efficient use or throughput, requires use of new design principles and techniques. The notion of a job is no longer relevant, an interactive user interface has to be designed, and new scheduling and memory management techniques are needed to provide good response times to a large number of users. These changes are discussed in the following Sections.

2.6.2 Scheduling

User service in a time sharing system is characterized by response times to user requests, so the time sharing kernel must provide good response times to *all* users. To realize this goal all users must get an *equal* opportunity to present their computational requests and have them serviced. Each user must also receive *reasonable* service. Two provisions are made to ensure this.

1. Programs are not assigned priorities because assignment of priorities may deny OS attention to low priority programs. Instead, programs are executed by turn.

2. A program is prevented from consuming unreasonable amounts of CPU time when scheduled to execute. This provision ensures that every request will receive OS attention without unreasonable delays.

These provisions are implemented using the techniques of *round-robin scheduling* and *time slicing*, respectively.

Round-robin scheduling When a user makes a computational request to his program, the program is added to the end of a *scheduling list*. The scheduler always removes the first program from the scheduling list and gives the CPU to it. When a program finishes computing its response to a request, it is removed from the CPU and the first program in the new list is selected for execution. When the user makes another request, the program is once again added to the end of the scheduling list.

Time slicing The notion of a *time slice* is used to prevent monopolization of the CPU by a program.

Definition 2.7 (Time slice) *The time slice δ is the largest amount of CPU time any program can consume when scheduled to execute on the CPU.*

Time slicing is an implementation of the notion of a time slice. Every program is subjected to the time limit specified by the time slice. A program exceeding this limit is preempted. Figure 2.19 illustrates a schematic of round-robin scheduling with time slicing. A preempted program is added to the end of the scheduling list. A program may be scheduled and preempted a few times before it produces a response.

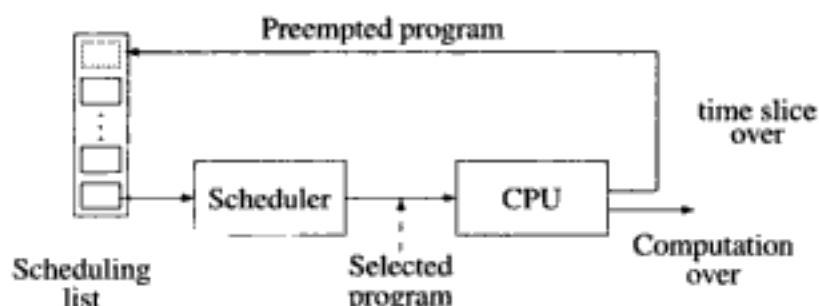


Fig. 2.19 A schematic of round-robin scheduling with time slicing

The time sharing kernel uses the interval timer of the computer to implement time slicing. The interval timer consists of a register called *timer register* that can store an integer number representing a time interval in hours, minutes and seconds. The contents of the register are decremented with an appropriate periodicity, typically a few times every second. A *timer interrupt* is raised when the contents of the timer register become zero, that is, when the time interval elapses.

A time sharing OS uses round-robin scheduling with time slicing. Algorithm 2.1 shows actions of the time sharing kernel.

Algorithm 2.1 (Time slicing)

1. Select the first program in the scheduling list for execution. Let the selected program be P. Remove P from the scheduling list.
 2. Load the value of the time slice (δ) into the interval timer.
 3. Start execution of program P on the CPU.
 4. If P initiates an I/O operation, goto step 1 to schedule another program for execution.
 5. When a timer interrupt occurs, preempt P and add it at the end of the scheduling list. Go to step 1.

In Step 5 the preempted program is put at the end of the scheduling list. If a program does not consume δ seconds of CPU time, e.g., if it starts an I/O operation, the kernel simply schedules the next program. When an I/O completion interrupt is raised the kernel identifies the program whose I/O operation has completed and adds the program at the end of the scheduling list.

Example 2.11 Figure 2.20 illustrates operation of Algorithm 2.1 in a time sharing system using time slice δ . It is assumed that scheduling overhead of the OS is σ seconds. Figure 2.20(a) depicts the situation when the time slice elapses during execution of a program. The kernel preempts this program and schedules another program for execution. The new program therefore starts executing σ seconds after the time slice elapses. In Figure 2.20(b), the program executing on the CPU makes an I/O request. After starting the I/O operation, the kernel schedules another program for execution. If we ignore the OS overhead in initiating an I/O operation, the newly scheduled program starts executing σ seconds after the I/O request was made.

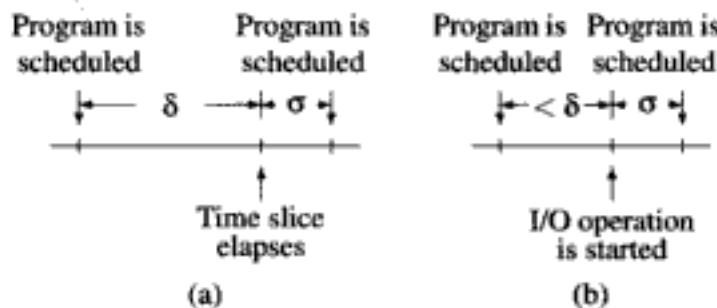


Fig. 2.20 Operation of a time slicing scheduler

When round-robin scheduling with time slicing is used, the response time for each user request can be estimated in the following manner: Let the number of users using the system at any time be n . Let each user request require exactly δ CPU seconds for completion. As in Example 2.11, let σ be the CPU time spent in switching from execution of one program to the next program. If we assume that an I/O operation completes instantaneously and a user submits the next request immediately

after receiving response to the previous request, the response time (rt) and the CPU efficiency (η) are given by

$$rt = n \cdot (\delta + \sigma) \quad (2.2)$$

$$\eta = \frac{\delta}{\delta + \sigma} \quad (2.3)$$

The actual response time may be different from the value of rt predicted by Eq. (2.2). There are several reasons for this. First, the value of rt may be smaller than that predicted by Eq. (2.2) because all users may not have made requests to their programs. Hence rt would not be influenced by n , the total number of users in the system; it would be influenced by the number of active users (n_a), $n_a < n$. Second, user requests do not require exactly δ CPU seconds to produce a response, so the relationship of rt and η with δ is more complex than shown in Eqs. (2.2) and (2.3). Table 2.12 summarizes the effect of small or large values of δ . From this one can conclude that rt may not be small for small δ and η may not be large for large δ , which makes the choice of δ a sensitive design decision.

Table 2.12 Influence of δ on response time

Value of δ	Influence on response time
δ very small	Value of δ may become smaller than the CPU time needed to satisfy a computational request. Such a program would have to be scheduled a few times before it can produce a response. Hence response time would be larger than $n_a \times (\delta + \sigma)$ seconds. Also, magnitudes of δ and σ may be comparable, which leads to low values of η from Eq. (2.3). Cache memory performance may also be poorer (see Section 2.1.1).
δ large	Most user requests may be processed in less than δ CPU seconds. Thus most programs release the CPU before δ seconds (see Step 4 of Algorithm 2.1). Hence $rt < n_a \times (\delta + \sigma)$ and η is worse than the value predicted by Eq. (2.3).

Example 2.12 Figure 2.21 shows operation of a time sharing system with two programs P_1 and P_2 in it. Parts (a) and (b) show execution of the programs all by themselves. Both programs have a cyclic behavior, each cycle containing a burst of CPU activity followed by a burst of I/O activity. The CPU bursts of programs P_1 and P_2 are 15 and 30 msec, respectively, while the I/O bursts are 100 and 60 msec, respectively. When executed by themselves, the programs have response times of 115 msec and 90 msec, respectively, since a programmer sees the results of a computation on the monitor at the end of an I/O operation.

Part (c) shows execution of the programs in a time sharing system using a time slice of 10 msec. Programs P_1 and P_2 now get CPU bursts limited to 10 msec. Both programs have to be scheduled a few times before they can complete the CPU bursts of their execution cycle and start I/O. Thus P_1 starts an I/O operation at 25 msec from start

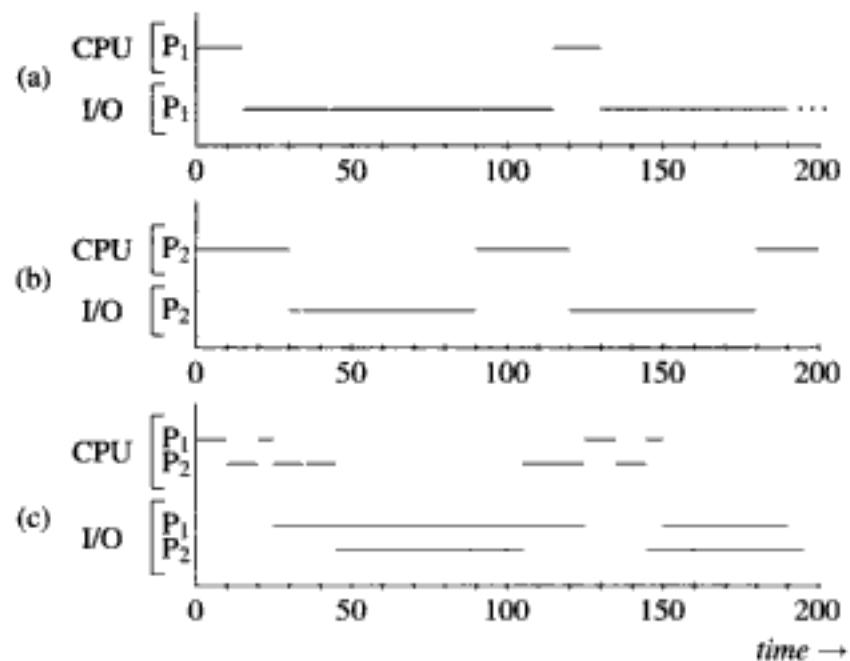


Fig. 2.21 Execution of programs P_1 and P_2 by themselves, and together

Table 2.13 Time sharing scheduling

Time	Scheduling list	Scheduled program	Remarks
0	{ P_1, P_2 }	P_1	P_1 is preempted at 10 msec
10	{ P_2, P_1 }	P_2	P_2 is preempted at 20 msec
20	{ P_1, P_2 }	P_1	P_1 starts I/O at 25 msec
25	{ P_2 }	P_2	P_2 is preempted at 35 msec
35	{ P_2 }	P_2	P_2 starts I/O at 45 msec
45	{}		CPU is idle
105	{ P_2 }	P_2	P_2 is preempted at 115 msec
115	{ P_2 }	P_2	P_2 is preempted at 125 msec
125	{ P_1, P_2 }	P_1	P_1 is preempted at 135 msec

of operation of the system. Now P_2 gets two consecutive time slices (separated, of course, by OS overhead for actions that first preempt program P_2 and then schedule it again because no other program in the system needs the CPU). P_1 's I/O operation completes at 125 msec. P_2 starts an I/O operation at 45 msec, which completes at 105 msec. Thus, the response times are 125 msec and 105 msec, respectively. Table 2.13 shows the scheduling list and scheduling decisions of the kernel between 0 and 105 msec assuming scheduling overhead to be negligible.

2.6.3 Memory Management

In Example 2.12, the CPU was idle between 45 msec and 105 msec, so the system can service a few more user programs. However, the computer system must possess a large memory to accommodate all these programs, which is an expensive proposition. The technique of *swapping* provides an alternative whereby a computer system can support a large number of users without having to possess a large memory.

Definition 2.8 (Swapping) Swapping is the technique of temporarily removing inactive programs from the memory of a computer system.

An inactive program is one which is neither executing on the CPU, nor performing an I/O operation. Figure 2.22 illustrates swapping as used in a practical situation. The programs existing in the memory are classified into three categories:

1. Active programs—one active program executes on the CPU while others perform I/O.
2. Programs being swapped out of the memory.
3. Programs being swapped into the memory.

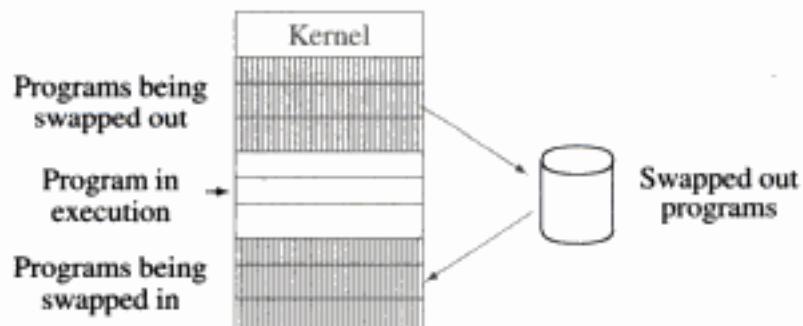


Fig. 2.22 A schematic of swapping

Whenever an active program becomes inactive, the OS swaps it out by copying its instructions and data onto a disk. A new program is loaded in its place. The new program is added at the end of the scheduling list; it receives CPU attention in due course.

Use of swapping is feasible in time sharing systems because the time sharing kernel can estimate when a program is likely to be scheduled next. It can use this estimate to ensure that the program is swapped in before its turn on the CPU. Note that swapping increases the OS overhead due to the disk I/O involved.

2.7 REAL TIME OPERATING SYSTEMS

In a class of applications called *real time applications*, users need the computer to perform some actions in a timely manner to control the activities in an external system, or to participate in them. The timeliness of actions is determined by the time

constraints of the external system. Accordingly, we define a real time application as follows:

Definition 2.9 (Real time application) A real time application is a program that responds to activities in an external system within a maximum time determined by the external system.

If the application takes too long to respond to an activity, a failure can occur in the external system. We use the term *response requirement* of a system to indicate the largest value of response time for which the system can function perfectly; a timely response is one whose response time is smaller than the response requirement of the system.

Consider a system that logs data received from a satellite remote sensor. The satellite sends digitized samples to the earth station at the rate of 10000 samples per second. The application process is required to simply store these samples in a file. Since a new sample arrives every 100 microseconds, the computer must respond to every "store the sample" request in less than 100 microseconds, or arrival of a new sample would wipe out the previous sample in the computer's memory. This system is a real time application because a sample must be stored in less than 100 microseconds to prevent a failure. Its response requirement is 99.9 microseconds. The *deadline* of an action in a real time application is the time by which the action should be performed. In the above example, if a new sample is received from the satellite at time t , the deadline for storing it on disk is $t + 99.9$ microseconds.

Examples of real time applications can be found in missile guidance, command and control applications like process control and air traffic control, data sampling and data acquisition systems like display systems in automobiles, multimedia systems, and applications like reservations and banking systems that are based on use of real time data bases. The response requirements of these systems vary from a few microseconds or milliseconds for guidance and control systems to a few seconds for reservations and banking systems.

2.7.1 Hard and Soft Real Time Systems

To take advantage of the features of real time systems while achieving maximum cost-effectiveness, two kinds of real time systems have evolved. A *hard real time system* is typically *dedicated* to processing real time applications, and provably meets the response requirement of an application under all conditions. A *soft real time system* makes the best effort to meet the response requirement of a real time application but cannot guarantee that it will be able to meet it under all conditions. Typically, it meets the response requirements in some probabilistic manner, say, 98 percent of the time. Guidance and control applications fail if they cannot meet the response time requirement, hence they are typically serviced using hard real time systems. Applications that aim at providing good quality of service, e.g., multimedia applications and applications like reservations and banking, do not have a notion of failure, so

they may be serviced using soft real time systems—the picture quality provided by a video on demand system may deteriorate occasionally, but one can still watch the video!

2.7.2 Features of a Real Time Operating System

A real time OS provides the special features summarized in Table 2.14. A real time application can be coded such that the OS can execute its parts concurrently. When priority-based scheduling is used, we have a situation analogous to multiprogramming *within* the application—if one part of the application initiates an I/O operation, the OS would schedule another part of the application. Thus, CPU and I/O activities of the application can be overlapped with one another, which helps to reduce the worst-case response time to the application. *Deadline scheduling* is a special scheduling technique that helps an application meet its deadline. Some aspects of deadline scheduling are described later in Section 4.2.

Table 2.14 Features of a real time operating system (RTOS)

- 1. Permits creation of multiple processes within an application
- 2. Permits priorities to be assigned to processes
- 3. Permits a programmer to define interrupts and interrupt processing routines
- 4. Uses priority driven or deadline oriented scheduling
- 5. Provides fault tolerance and graceful degradation capabilities.

Specification of *domain specific interrupts* and interrupt servicing actions for them enables a real time application to respond to special conditions and events in the external system in a timely manner. Predictability of policies and overhead of the OS enables an application developer to calculate the worst-case response time that the OS can provide and determine whether it can meet the response requirements of the application. When resources are overloaded, OS overhead can become unbounded, i.e., arbitrarily large. For example, if memory utilization is very high the memory allocator may take long to find a free memory area to meet a memory request. A hard real time OS must avoid such situations. Hard real time systems therefore partition their resources (see Section 1.3.2) and allocate them permanently to competing processes in the application. This approach reduces the allocation overhead and guarantees that the kernel response for a resource request made by one process would be independent of resource utilization by other processes in the system. Hard real time systems also avoid use of features whose performance cannot be predicted precisely, e.g. virtual memory (see Chapter 6).

A real time OS employs two techniques to ensure continuity of operation when faults occur—*fault tolerance*, and *graceful degradation*. A fault tolerant computer system uses redundancy of resources to ensure that the system will keep functioning even if a fault occurs, e.g., it may have two disks even though the application actually

needs only one disk. Graceful degradation is the ability of a system to fall back to a reduced level of service when a fault occurs and to revert to normal operations when the fault is rectified. The programmer can assign high priorities to crucial functions so that they would be performed in a timely manner even when the system operates in a degraded mode.

2.8 DISTRIBUTED OPERATING SYSTEMS

A distributed computer system consists of several individual computer systems connected through a network. Thus, many resources of a kind, e.g., many memories, CPUs and I/O devices, exist in the distributed system. A distributed operating system exploits the multiplicity of resources and the presence of a network to provide the advantages of resource sharing across computers, reliability of operation, speed-up of applications, and communication between users. However, the possibility of network failures or failures of individual computer systems complicates functioning of the operating system and necessitates use of special techniques in its design. Users also need to use special techniques to access resources over the network. We discuss these aspects in Section 2.8.1.

Table 2.15 Features of distributed operating systems

Feature	Description/Implication
Resource sharing	Improves resource utilization across boundaries of individual computer systems.
Reliability	Availability of resources and services despite failures.
Computation speed-up	Parts of a computation can be executed in different computer systems to speed-up the computation.
Communication	Provides means of communication between remote entities.
Incremental growth	Capabilities of a system (e.g., its processing power) can be enhanced at a price proportional to the nature and size of the enhancement.

Table 2.15 summarizes advantages of distributed operating systems. *Resource sharing* has been the traditional motivation for distributed operating systems. The earliest form of a distributed operating system was a network operating system, which enabled the use of specialized hardware and software resources by geographically distant users. Resource sharing continues to be an important aspect of distributed operating systems today, although the nature of distribution and sharing of resources has changed due to advances in the networking technology. Sharing of resources is now equally meaningful in a local area network (LAN). Thus, low-cost computers and workstations in an office or a laboratory can share some expensive resources like laser printers.

One aspect of *reliability* is availability of a resource despite failures in a sys-

tem. A distributed environment can offer enhanced availability of resources through redundancy of resources and communication paths. For example, availability of a disk resource can be increased by having two or more disks located at different sites in the system. If one disk is unavailable due to a failure, a process can use some other disk. Availability of a data resource, e.g., a file, can be similarly enhanced by keeping copies of the file in various parts of the system.

Computation speed-up implies obtaining better response times or turnaround times for an application. This is achieved by dispersing processes of an application to different computers in the distributed system, so that they can execute at the same time. This arrangement is qualitatively different from the overlapped operation of processes of an application in a conventional operating system (see Section 2.7).

Users of a distributed operating system have user ids and passwords that are valid throughout the system. This feature greatly facilitates *communication* between users in two ways. First, communication through user ids automatically invokes the security mechanisms of the OS and thus ensures authenticity of communication. Second, users can be mobile within the distributed system and still be able to communicate with other users of the system conveniently.

2.8.1 Special Techniques of Distributed Operating Systems

A distributed system is more than a mere collection of computers connected to a network—functioning of individual computers must be integrated to achieve effective utilization of the services and resources in the system in the manner summarized in Table 2.15. This is achieved through participation of all computers in the control functions of the operating system. Accordingly, we define a distributed system as follows:

Definition 2.10 (Distributed System) *A distributed system is a system consisting of two or more nodes, where each node is a computer system with its own memory, some networking hardware, and a capability of performing some of the control functions of an OS.*

The control functions performed by individual computer systems contribute to effective utilization of the distributed system. Table 2.16 summarizes three key concepts and techniques used in a distributed OS. *Distributed control* is the opposite of centralized control—it implies that the control functions of the distributed system are performed by several computers in the system in the manner of Def. 2.10, instead of being performed by a single computer. Distributed control is essential for ensuring that failure of a single computer, or a group of computers, does not halt operation of the entire system. *Transparency* of a resource or service implies that a user should be able to access it without having to know which node in the distributed system contains it. This feature enables the OS to change the position of a software resource or service to optimize its use by computations. For example, in a system providing

Table 2.16 Key concepts and techniques used in a distributed OS

Technique	Description
Distributed control	A control function is performed through participation of several nodes, possibly <i>all</i> nodes, in a distributed system.
Transparency	A resource or service can be accessed without having to know its location in the distributed system.
Remote procedure call (RPC)	A process calls a procedure that is located in a different computer system. The procedure call is analogous to a procedure or function call in a programming language, except that it is the OS that passes parameters to the remote procedure and returns its results. Operation of the process making the call is resumed when results are returned to it.

transparency, a distributed file system may move a file to the node that contains a computation using the file, so that the delays involved in accessing the file over the network would be eliminated. The *remote procedure call* (RPC) is used by an application to execute a procedure in another computer in the distributed system. The remote procedure may perform a part of the computation in the application, or it may use a resource located in that computer.

2.9 MODERN OPERATING SYSTEMS

Users engage in diverse activities in a modern computing environment. Hence a modern operating system cannot use a uniform strategy for all processes; it must use a strategy that is appropriate for each individual process.. For example, a user may open a mail handler, edit a few files, execute some programs, and watch a video at the same time. Here, execution of a program may be interactive, and may involve an activity in another node of a distributed computer system, and watching of a video is a soft real time activity, so the OS must use round-robin scheduling for program executions, use priority-based scheduling for processes of the video application, and implement remote procedure calls (RPC) to support activities in another node. Thus, a modern OS uses most concepts and techniques that we discussed in connection with the batch processing, multiprogramming, time sharing, real time and distributed operating systems. Table 2.17 contains a summary of such concepts and techniques.

An OS cannot use different strategies for different kinds of processes, so a modern OS actually uses a single complex strategy that adapts its working to suit each individual process. For example, the scheduling strategy may be priority-based, however instead of assigning a fixed priority to a process as in a multiprogramming system, the scheduling strategy might consider the recent behavior of a process to decide its current priority. This way, a real time process would get a high priority, all interactive processes would get a moderate priority, and non-interactive processes would

Table 2.17 Use of classical OS concepts in modern operating systems

Concept	Typical example of use
Batch processing (Batch processing OS)	Data base updates and queries in the back office are batch processed to avoid initiating a data base for every query or update. Batch processing is also routinely used for scientific computations in research organizations and clinical laboratories.
Priority-based preemptive scheduling (Multiprogramming OS)	Used by the kernel to provide favored treatment to high priority applications, and to achieve efficient use of resources by assigning high priorities to interactive processes and low priorities to non-interactive processes.
Time slicing (Time sharing OS)	Used by the kernel to prevent a process from monopolizing the CPU. It helps to provide good response times.
Swapping (Time sharing OS)	Used by the kernel to increase the number of processes that can be serviced at the same time. It also helps to improve system performance.
Multiple processes in an application (Real time OS)	Used by an application to speed up its execution by reducing its elapsed time. This approach is most effective when the application contains substantial CPU and I/O activities.
Resource sharing (Distributed OS)	Routinely used in a LAN environment to share centralized resources like laser printers.

get the lowest priority. Thus, the essence of strategies used in modern OSs is flexibility and an ability to adapt the strategy to suit the requirements of a process. This approach is reflected in the design of OS strategies presented in following chapters.

EXERCISE 2

1. Which of the following should be privileged instructions? Explain why.
 - (a) Put the CPU in privileged mode
 - (b) Load bound registers
 - (c) Load a value in a CPU register
 - (d) Mask off some interrupts
 - (e) Forcibly terminate an I/O operation.
2. The CPU should be in the privileged mode while executing the kernel code and in the user mode (i.e., non-privileged mode) while executing a user program. Explain how this is achieved during operation of an OS.
3. Justify the following statement: "Due to presence of the cache memory, a program requires more CPU time to execute in a multiprogramming or time sharing system than it would require if it were to be executed all by itself, i.e., without having any

other programs in the system."

4. Write a note on contents of various fields of an interrupt vector.
5. What is CPU state? Explain how the notion of CPU state is useful in implementing multiprogramming.
An interrupt occurs when a program is about to execute a 'branch on condition' instruction. Control is returned to the program after processing the interrupt. Explain how occurrence of the interrupt does not affect correct execution of the branch instruction.
6. The kernel of an OS masks off all interrupts during interrupt processing. Discuss its advantages and disadvantages.
7. A computer system organizes the saved PSW information area as a stack. It pushes contents of the PSW onto this stack during Step 2 of the interrupt action (see Figure 2.7). Explain advantages of the stack for interrupt processing.
8. When a program makes a request through a system call, the kernel suspends it if the request cannot be satisfied straightaway. Classify the system calls of Table 2.4 into two classes—those that may lead to suspension of the program making the system call, and those that will not.
9. An OS provides a system call for requesting allocation of memory. An experienced programmer offers the following advice: "If your program contains many requests for memory, you can speed up its execution by combining all these requests into a single system call." Explain why this is so.
10. A job contains 5 data cards in a job step. However, the job step program tries to read 10 cards. Clearly explain actions of the command processor when the program tries to read the sixth and subsequent cards.
11. A system is an overloaded system if its capacity to perform work is smaller than the work directed at it; otherwise, it is an underloaded system. The following policy is proposed to improve the throughput of a batch processing system: Classify jobs into small jobs and long jobs depending on their CPU time requirements. Form separate batches of short and long jobs. Execute a batch of long jobs only if no batches of short jobs exist.
Does this policy improve the throughput of a batch processing system that is: (a) underloaded, (b) overloaded?
12. Study the .bat files of MS DOS and the shells of Unix. Comment on their merits and demerits for batch processing.
13. Analyze the operation of a multiprogramming kernel and list all its actions that contribute to OS overhead.
14. The kernel of a multiprogramming system classifies a program as CPU-bound or I/O-bound and assigns an appropriate priority to it. What would be the consequence of a wrong classification of programs for throughput and response times in a multiprogramming system? What would be the consequences for the throughput vs. degree of multiprogramming characteristic of Figure 2.18?
15. When a high priority program initiates an I/O operation, a multiprogramming system switches to execution of a low priority program. Give a step-by-step explanation of how this is achieved. (*Hint:* A program initiates an I/O operation by making a system call.)

16. The CPU of a multiprogramming system is executing a high priority program when an interrupt signaling completion of an I/O operation occurs. Show all actions and activities in the OS following the interrupt if

- (a) The I/O operation had been started by a lower priority program
- (b) The I/O operation had been started by a higher priority program.

Illustrate each case with the help of a timing chart.

17. Draw a timing chart for a system containing two CPU-bound programs and two I/O-bound programs when (a) CPU-bound programs have a higher priority, (b) I/O-bound programs have a higher priority.

18. A program consists of a single loop that executes 50 times. The loop contains a computation that consumes 50 msec of CPU time, followed by an I/O operation that lasts for 200 msec.

If the program is executed in a multiprogramming OS whose overhead is negligible, prepare a timing chart showing the CPU and I/O activities of the program and compute its elapsed time in the following cases:

- (a) The program has the highest priority in the system.
- (b) The program is multiprogrammed with n other programs with identical characteristics and has the lowest priority. Consider cases (i) $n = 3$, (ii) $n = 4$, and (iii) $n = 5$.

19. A program is said to 'make progress' if either the CPU is executing its instructions or its I/O operation is in progress. The progress coefficient of a program is the fraction of its lifetime in the system during which it makes progress. Compute progress coefficients of the programs in Problem 18(b).

20. Comment on validity of the following statement:

"A CPU-bound program always has a low progress coefficient in a multiprogramming system."

21. A multiprogramming system uses a degree of multiprogramming ($m \gg 1$). It is proposed to double the throughput of the system by augmentation/replacement of its hardware components. Comment on the following four proposals in this context.

- (a) Replace the CPU by a CPU with twice the speed.
- (b) Expand the main memory to twice its present size.
- (c) Replace the CPU by a CPU with twice the speed and expand the main memory to twice its present size.

22. All programs in a multiprogramming system are cyclic in nature, with each cycle containing a burst of CPU activity and a burst of I/O activity. Let b_{cpu}^i and b_{io}^i be the CPU and I/O bursts of program P_i . The programs are named $P_1 \dots P_m$, where m is the degree of multiprogramming, such that priority of program $P_i >$ priority of program P_{i+1} . Comment on validity of the following statements

- (a) CPU idling occurs if $b_{io}^h > \sum_{j \neq h} (b_{cpu}^j)$ where P_h is the highest priority program.
- (b) Program P_m is guaranteed to receive CPU time if for all $i = 1 \dots m - 1$, $b_{io}^i < (b_{cpu}^{i+1} + b_{io}^{i+1})$ and $b_{io}^i > \sum_{j=i+1 \dots m} (b_{cpu}^j)$.

23. A program is said to starve if it does not receive any CPU time. Which of the following conditions implies starvation of the lowest priority program in a multiprogram-

- ming system? (Note: notation is identical with Problem 22.)
- (a) For some program P_i , $b_{io}^j < \sum_{j=i+1 \dots m-1} (b_{cpu}^j)$.
 - (b) For some program P_i , $b_{io}^j < \sum_{j=i+1 \dots m-1} (b_{cpu}^j)$ and $b_{cpu}^j > b_{io}^j$ for all $j > i$.
24. A time sharing system contains n identical programs, each executing a loop that contains a computation requiring t_p CPU seconds and an I/O operation requiring t_w seconds. Draw a graph depicting variation of response time with values of the time slice δ . (Hint: Consider cases for $t_p < \delta$, $\delta < t_p < 2 \times \delta$ and $t_p > 2 \times \delta$.)
25. Two persons using the same time sharing system at the same time notice that the response times to their programs differ widely. What are the possible reasons for this difference?
26. Comment on validity of the following statement: Operation of a time sharing system is identical with operation of a multiprogramming system executing the same programs if δ exceeds the CPU burst of every program.
27. (a) Does swapping improve/degrade efficiency of system utilization?
 (b) Does swapping increase the effective degree of multiprogramming?
 (c) Can swapping be used in a multiprogramming system?
- Clearly justify your answers.
28. A time sharing system uses swapping as the fundamental memory management technique. It uses the following lists to govern its actions: a scheduling list, a swapped out list containing programs that are swapped out, a being swapped-out list containing programs to be swapped out, and a being swapped-in list containing programs to be swapped in. Explain when and why the time sharing kernel should put programs in the being swapped-out and being swapped-in lists.
29. A real time application requires a response time of 2 seconds. Discuss the feasibility of using a time sharing system for the real time application if the average response time in the time sharing system is (a) 20 seconds, (b) 2 seconds, or (c) 0.2 seconds.
30. A time sharing system processes n programs. It uses a time slice of δ CPU seconds, and requires t_v CPU seconds to switch between programs. A real time application requires t_C seconds of CPU time, involves I/O operations that require t_I seconds, and requires a response to be produced within t_D seconds. What is the largest value of δ for which the time sharing system can satisfy the response requirements of the real time application?
31. In a multiprogramming system an I/O-bound activity is given a higher priority than non I/O-bound activities, however in a real time application an I/O bound activity may be given a lower priority. Why is this so?
32. An OS cannot meet the response requirement of a real time application if it is executed as a single process. Explain with an example how creation of multiple processes can help to meet the response requirements of the application in the same OS.
33. An application program is being developed for a microprocessor based controller for an automobile. The application is required to perform the following functions:
- (a) Monitor and display the speed of the automobile
 - (b) Monitor the fuel level and raise an alarm, if necessary
 - (c) Display the fuel efficiency, i.e., miles/gallon at current speed

- (d) Monitor the engine condition and raise an alarm if an unusual condition arises
- (e) Periodically record some auxiliary information like speed and fuel level (i.e., implement a 'black box' as in an airliner).

Answer the following questions concerning the application:

- (a) Is this a real time application? Justify your answer.
 - (b) It is proposed to create multiple processes to reduce the response time of the application. What are the processes in it? What should be their priorities?
 - (c) Is it necessary to define any application-specific interrupts? If so, specify the interrupts and their priorities.
34. If two independent events e_1 and e_2 have the probability of occurrence of pr_1 and pr_2 , where both pr_1 and $pr_2 < 1$, the probability that both events occur at the same time is $pr_1 \times pr_2$. A computer system containing two CPUs is to be designed such that the probability that both CPUs fail is 1 in 10000. What should be the probability of failure of a CPU?
35. To obtain computation speed-up in a distributed system, an application is coded as three parts that can be executed on three computer systems under control of a distributed operating system. However, the speed-up obtained is < 3. List all possible reasons for a poor speed-up.

BIBLIOGRAPHY

Smith (1982) and Handy (1998) describe cache memory organizations. Przybylski (1990) discusses cache and memory hierarchy design. Memory hierarchy and I/O organization are also covered in most books on computer architecture and organization, e.g., Hayes (1988), Patterson and Hennessy (1998), Hennessy and Patterson (2002), Hamacher *et al* (2002), and Stallings (2003).

Most books on operating systems discuss the system calls interface. Bach (1986) contains a useful synopsis of Unix system calls. O'Gorman (2003) describes interrupt processing in Linux. Beck *et al* (2002), Bovet and Cesati (2003) and Love (2005) contain extensive discussions of Linux system calls. Mauro and McDougall (2001) describe system calls in Solaris, while Russinovich and Solomon (2005) describe system calls in Windows.

Denning (1971) and Weizer (1981) are survey articles on operating systems. Metzner (1982) and Brumfield (1986) are comprehensive bibliographies covering all aspects of operating system literature. Several recent OS bibliographies are available on the Internet. All operating systems texts cover different classes of operating systems described in this chapter; some recent OS texts are Tanenbaum (2001), Stallings (2001), Nutt (2004) and Silberschatz *et al* (2005).

Other literature on batch processing, multiprogramming and time sharing systems dates back to the 1970's. Liu (2000) and Zhao (1989) are good sources for real time systems.

Tanenbaum and Renesse (1985) is a good starting point for a study of distributed operating systems. It discusses the major design issues in distributed operating systems and contains a survey of some distributed operating systems. Tanenbaum (2003) discusses the design principles of computer networks. Fortier (1988) discusses the design aspects of distributed operating systems. Tanenbaum (1995) discusses some well-known distributed operating sys-

tems in detail. Singhal and Shivaratri (1994) discuss all facets of distributed operating systems. Coulouris *et al* (2001) discuss the concepts and design of distributed systems.

Several books describe specific modern operating systems. Bach (1986), and Vahalia (1996) describe the Unix operating system. Beck *et al* (2002) and Bovet and Cesati (2003) discuss the Linux operating system, while Stevens and Rago (2005) describe Unix, Linux and BSD operating systems. Mauro and McDougall (2001) discuss Solaris. Russinovich and Solomon (2005) describe the Windows operating systems.

1. Arden, B., and D. Boettner (1969): "Measurement and performance of a multiprogramming system," *Second ACM Symposium on Operating System Principles*.
2. Bard, Y. (1971): "Performance criteria and measurement for a time-sharing system," *IBM Systems Journal*, **10** (3), 193.
3. Bach, M. J. (1986): *The Design of the Unix Operating System*, Prentice-Hall, Englewood Cliffs.
4. Beck, M., H. Bohme, M. Dziadzka, U. Kunitz, R. Magnus, C. Schroter, and D. Verworner (2002): *Linux Kernel Programming, Third edition*, Pearson Education.
5. Bic, L. and A. C. Shaw (1988): *The Logical Design of Operating Systems, Second edition*, Prentice-Hall, Englewood Cliffs.
6. Bovet, D. P., and M. Cesati (2003): *Understanding the LINUX Kernel, Second edition*, O'reilly, Sebastopol.
7. Brumfield, J. A. (1986): "A guide to operating systems literature," *Operating Systems Review*, **20** (2), 38–42.
8. Bull, G. M., and S. F. G. Packham (1971): *Time-Sharing Systems*, McGraw-Hill, London.
9. Coulouris, G., J. Dollimore, and T. Kindberg (2001): *Distributed Systems—Concepts and Design, Third edition*, Addison-Wesley, New York.
10. Crowley, C. (1997): *Operating Systems—A Design Oriented Approach*, McGraw-Hill, New York.
11. Denning, P. J. (1971): "Third generation operating systems," *Computing Surveys*, **4** (1), 175–216.
12. Fortier, P. J. (1988): *Design of Distributed Operating Systems*, McGraw-Hill, New York.
13. Goscinski, A. (1991): *Distributed Operating Systems—The Logical Design*, Addison-Wesley, New York.
14. Hamacher, C., Z. Vranesic, and S. Zaky (2002): *Computer Organization, Fifth edition*, McGraw-Hill.
15. Hamlet, R. G. (1973): "Efficient multiprogramming resource allocation and accessing," *Communications of the ACM*, **16** (6), 337–343.
16. Handy, J. (1998): *The Cache Memory Book, Second edition*, Academic Press.
17. Hayes, J. (1988): *Computer Architecture and Organization, Second edition*, McGraw-Hill, New York.
18. Hennessy, J., and D. Patterson (2002): *Computer Architecture : A Quantitative Approach, Third edition*, Morgan Kaufmann, San Mateo.
19. Liu, J. W. S. (2000): *Real-Time systems*, Pearson education.
20. Love, R. (2005): *Linux Kernel Development, Second edition*, Novell Press.

21. Mauro, J., and R. McDougall (2001): *Solaris Internals—Core Kernel Architecture*, Prentice-Hall.
22. McQuillan, J. M., and D. C. Walden (1977): "The ARPA network design decisions," *Computer Networks*, **1**, 243–289.
23. Metzner, J. R. (1982): "Structuring operating systems literature for the graduate course," *Operating Systems Review*, **16** (4), 10–25.
24. Nutt, G. (2004): *Operating Systems—A Modern Perspective, Third edition*, Addison-Wesley, Reading.
25. O'Gorman, J. (2003): *Linux Process Manager: The internals of Scheduling, Interrupts and Signals*, Wiley and Sons.
26. Patterson, D., and J. Hennessy (1998): *Computer Organization and Design: The Hardware/Software Interface*, Morgan Kaufman, San Mateo.
27. Przybylski, A. (1990): *Cache and Memory Hierarchy Design: A Performance-Directed Approach*, Morgan Kaufmann.
28. Rosin, R. F. (1969): "Supervisory and monitor systems," *Computing Surveys*, **1** (1), 15–32.
29. Russinovich, M. E., and D. A. Solomon (2005): *Microsoft Windows Internals, Fourth edition*, Microsoft Press, Redmond.
30. Silberschatz, A., P. B. Galvin, and G. Gagne (2005): *Operating System Principles, Seventh edition*, John Wiley, New York.
31. Singhal, M., and N. G. Shivaratri (1994): *Advanced Concepts in Operating Systems*, McGraw-Hill, New York.
32. Sinha, P. K. (1997): *Distributed Operating Systems*, IEEE Press, New York.
33. Smith, A. J. (1980): "Multiprogramming and memory contention," *Software—Practice and Experience*, **10** (7), 531–552.
34. Smith, A. J. (1982): "Cache memories," *ACM Computing Surveys*, **14**, 473–530.
35. Smith, L. B. (1967): "A comparison of batch processing and instant turnaround," *Communications of the ACM*, **10** (8).
36. Stallings, W. (2001): *Operating Systems—Internals and Design Principles, Fourth edition*, Pearson Education, New York.
37. Stallings, W. (2003): *Computer Organization and Architecture, Sixth edition*, Prentice Hall, Upper Saddle River.
38. Stevens, W. R., and S. A. Rago (2005): *Advanced Programming in the Unix Environment, Second edition*, Addison Wesley Professional.
39. Tanenbaum, A. S. (2003): *Computer Networks, Fourth edition*, Prentice-Hall, Englewood Cliffs.
40. Tanenbaum, A. S. (2001): *Modern Operating Systems, Second edition*, Prentice-Hall, Englewood Cliffs.
41. Tanenbaum, A. S., and R. Van Renesse (1985): "Distributed Operating Systems," *Computing Surveys*, **17** (1), 419–470.
42. Tanenbaum, A. S. (1995): *Distributed Operating Systems*, Prentice-Hall, Englewood Cliffs, N.J.
43. Vahalia, U. (1996): *Unix Internals: The New Frontiers*, Prentice Hall, Englewood Cliffs.

44. Watson, R. W. (1970): *Time Sharing System Design Concepts*, McGraw-Hill, New York.
45. Weizer, N. (1981): "A history of operating systems," *Datamation*, January, 119–126.
46. Wirth, N. (1969): "On multiprogramming, machine coding, and computer organization," *Communications of the ACM*, 12 (9), 489–491.
47. Wilkes, M. V. (1968): *Time Sharing Computer Systems*, Macdonald, London.
48. Zhao, W. (1989) : Special issue on real-time operating systems, *Operating System Review*, 23, 7.

Processes and Threads

The process concept helps to explain, understand and organize execution of programs in an OS. A process is *an* execution of a program. The emphasis on ‘an’ implies that several processes may represent executions of the same program. This situation arises when several executions of a program are initiated, each with its own data, and when a program that is coded using concurrent programming techniques is in execution.

A programmer uses processes to achieve execution of programs in a sequential or concurrent manner as desired. An OS uses processes to organize execution of programs. Use of the process concept enables an OS to execute both sequential and concurrent programs equally easily.

We discuss two views of processes in this Chapter: the programmer view, and the OS view. In the programmer view we discuss how concurrent processes are created and how they interact with one another to meet a common goal. In the OS view we discuss how an OS creates processes, how it keeps track of *process states*, and how it uses the process state information to organize execution of programs.

A *thread* is an execution of a program that uses the environment of a process, that is, its code, data and resources. If many threads use the environment of the same process, they share its code, data and resources. An OS uses this fact to reduce its overhead while switching between such threads.

3.1 PROCESSES AND PROGRAMS

A program is a passive entity that does not perform any actions by itself; it has to be executed to realize the actions specified in it. A *process* is an execution of a program; it actually performs the actions specified in a program. An operating system considers processes as entities for scheduling. This is how it realizes execution of user programs.

To understand what is a process, let us discuss how the OS executes a program. Program P shown in Figure 3.1(a) contains declarations of a fileinfo and a variable item, and statements that read values from info, use them to perform some calculations, and print a result before coming to a halt. During execution, instructions of this program use values in its data area and the stack to perform the intended calculations. Figure 3.1(b) shows an abstract view of its execution. The instructions, data and stack of program P constitute its *address space*. To realize execution of P, the OS allocates memory to accommodate P's address space, allocates a printer to print its results, sets up an arrangement through which P can access fileinfo, and schedules P for execution. The CPU is shown as a dashed box because it is not always executing instructions of P; the OS shares the CPU between execution of P and executions of other programs.

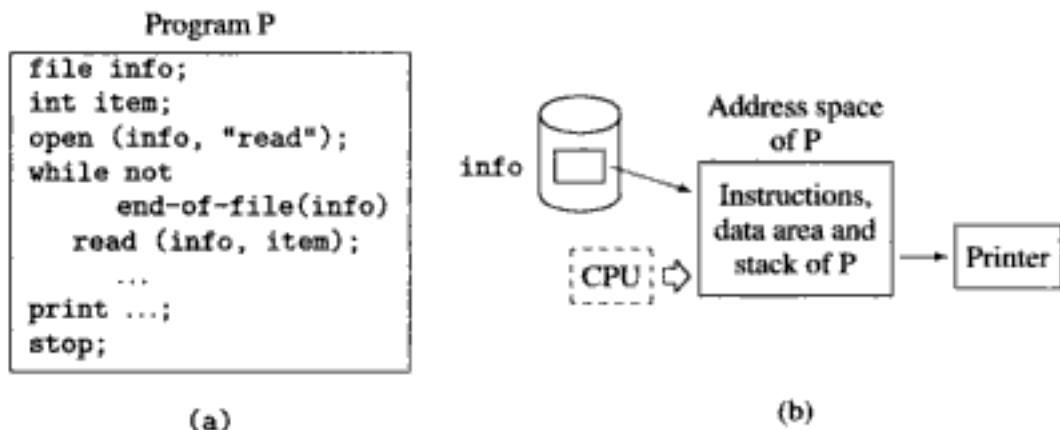


Fig. 3.1 A program and an abstract view of its execution

Table 3.1 shows two kinds of relationships that can exist between processes and programs. A one-to-one relationship exists when a single execution of a sequential program is in progress. The program consists of a main program and a set of functions. During its execution, control flows between the main program and the functions according to the logic of the program. The OS is not aware of the existence of functions. Hence execution of the program constitutes a single process. A many-to-one relationship would exist between many processes and this program if several executions of the program are in progress at the same time.

Table 3.1 Relationships between processes and programs

Relationship	Examples
One-to-one	A single execution of a sequential program
Many-to-one	Many simultaneous executions of a program, Execution of a concurrent program

Most programming languages contain features for *concurrent programming*. During execution, a program coded using these features informs the OS about its parts that are to be executed concurrently. The OS considers each of these executions as a process. Hence the processes have a many-to-one relationship with the program. We call such a program a *concurrent program*. Processes that co-exist in the system at some time are called *concurrent processes*. Concurrent processes have opportunities to interact with one another during their execution.

Child processes The OS initiates an execution of a program by creating a process for it. This is called the *main process* for the execution. The main process may create other processes, which become its child processes. A child process may itself create other processes, and so on. All these processes form a tree with the main process as its root. Example 3.1 illustrates creation of child processes.

Example 3.1 The real time data logging application of Section 2.7 receives data samples from a satellite at the rate of 10000 samples per second and stores them on a disk. We assume that each sample arriving from the satellite is put into a special register of the computer. The primary process of the application, which we will call the *data_logger* process, has to perform the following three functions:

1. Copy the sample from the special register into memory.
2. Write the sample into a disk file.
3. Perform some housekeeping operations, e.g., copy some selected fields of the incoming samples into another file used for statistical analysis.

It creates three child processes, leading to the process tree shown in Figure 3.2(a). As shown in Figure 3.2(b), *copy_sample* copies the sample from the register into a memory area named *buffer_area* which can hold, say, 50 samples. *disk_write* writes a sample from *buffer_area* into a disk file. *housekeeping* performs housekeeping operations. Arrival of a new sample is defined as an interrupt in the application, and a programmer defined interrupt routine is associated with this interrupt. The OS executes this routine whenever a new sample arrives. It activates *copy_sample*.

Execution of the three processes can overlap as follows: *copy_sample* can copy a sample into *buffer_area*, *disk_write* can write a previous sample to the disk, and *housekeeping* can copy fields from samples already stored on disk for statistical analysis. This arrangement provides a smaller worst-case response time of the application than if these functions were to be executed sequentially. So long as *buffer_area* has some free space, only *copy_sample* has to complete before the next sample arrives. The other processes can be executed later. This possibility is exploited by assigning the highest priority to *copy_sample*.

Table 3.2 describes three advantages of creating child processes—computation speed-up, higher priority for critical functions and protection of the parent process from malfunctioning of child processes. The third advantage needs some explanation. When a software system has to invoke an untrusted program, it must protect itself against errors in the program. It can achieve this by creating a child process to execute the program. If an error arises during its execution, the OS would abort the child process. Thus, the parent process would not be affected by the error. The OS

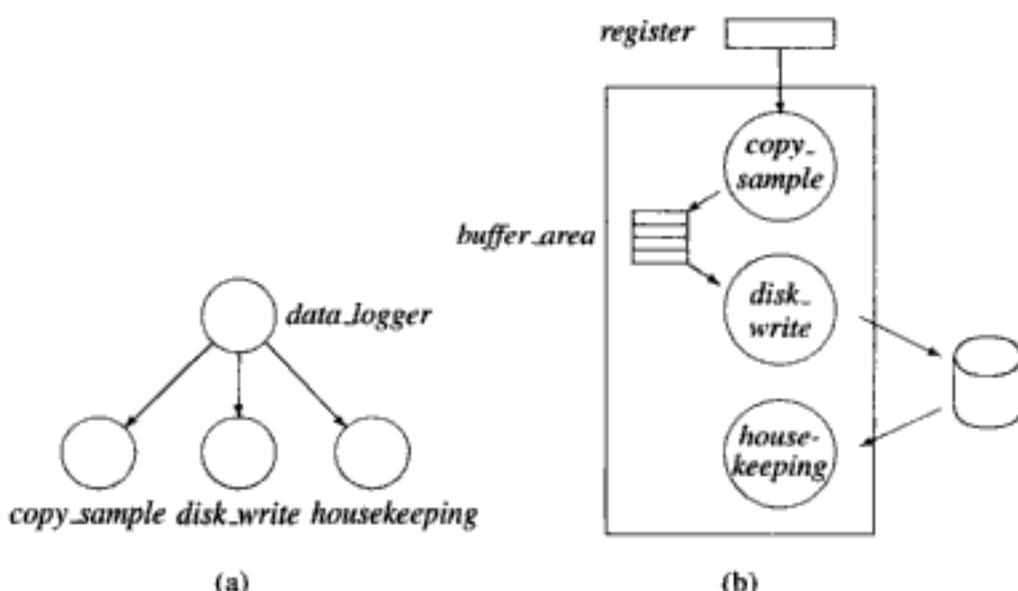


Fig. 3.2 Real time application of Section 2.7: (a) Process tree, (b) Processes

Table 3.2 Advantages of child processes

Advantage	Explanation
Computation speed-up	Creation of multiple processes in an application provides <i>multi-tasking</i> . Its benefits are similar to those of multiprogramming; it enables the OS to interleave execution of I/O-bound and CPU-bound processes in an application, thereby providing computation speed-up.
Priority for critical functions	A child process created to perform a critical function in an application may be assigned a higher priority than other functions. Such priority assignments may help the OS to meet real-time requirements of an application.
Protecting parent process from errors	The OS cancels a child process if an error arises during its execution. This action does not affect the parent process.

command processor uses this feature to advantage. It creates a new process to execute a user program. This way the command processor does not come to any harm even if the program malfunctions.

3.2 PROGRAMMER VIEW OF PROCESSES

In the programmer view, processes are a means to achieve concurrent execution of a program. The main process of a concurrent program creates child processes. It should assign appropriate priorities to them to either achieve computation speed-up or to execute some functions at a high priority as desired (see Section 3.1). The

main process and the child processes also have to interact to achieve their common goal. This interaction may involve exchange of data or may require the processes to coordinate their activities with one another.

Accordingly, an operating system provides the following four operations to implement the programmer view of processes:

1. Creating child processes and assigning priorities to them
2. Terminating child processes
3. Determining the status of child processes
4. Sharing, communication and synchronization between processes.

We discuss the first three operations here with the help of an example. Sharing, communication and synchronization between processes are discussed in Section 3.2.1.

We illustrate features of processes with the help of the real time application discussed in Example 3.1. Figure 3.3 gives a rather simplistic pseudo-code of the application written for a hypothetical operating system. The code uses the following three system calls:

<i>create_process</i>	:	Creates a new process and assigns to it a priority, and a unique identifier called its process id. Returns the process id to the caller.
<i>status</i>	:	Checks the status of a process and returns a code <i>terminated</i> or <i>alive</i> .
<i>terminate</i>	:	Terminates the specified child process or terminates itself if no process is specified.

create_process takes two parameters—a procedure name and an integer value. The procedure becomes the code component of the new process, and the address of the procedure is noted as the address of the next instruction to be executed in the process. The priority of the new process is obtained by adding the integer value to the priority of the parent process. It is assumed that a larger numerical value represents a higher scheduling priority. The call returns the unique process id assigned to the new process.

The main process, which is assumed to have the name *main*, begins by creating three processes for the three functions in the application and stores their ids in the strings *copy_sample*, *disk_write* and *housekeeping*. In keeping with the discussion in Section 2.7, the highest priority is assigned to the process that moves samples to the buffer. The process that writes samples to the disk has the next highest priority and the process performing statistical analysis has the lowest priority. Other features of the code are described in the next Section.

The program of Figure 3.3 has a weak spot. The main process spends its time in a loop where it uses the *status* call to check whether all child processes have terminated and terminates itself when this happens. This is wasteful. In Section 3.5.1 we discuss features that avoid such looping. The program also does not specify when child processes should terminate.

```

begin
    size : integer value (...);
    buffer: array[1..size] of ...;
    copy_sample, disk_write, housekeeping : string;
    no_of_samples : integer;
    /* Create processes */
    copy_sample := create_process(move_to_buffer(), 3);
    disk_write := create_process(write_to_disk(), 2);
    housekeeping := create_process(analysis(), 1);
    /* Send size information to copy_sample and disk_write */
    send (copy_sample, size);
    send (disk_write, size);
    /* Check status of all processes */
    over := false;
    while (over = false)
        if status(copy_sample) = terminated and status(disk_write) =
           terminated and status(housekeeping) = terminated then
            over := true;
    terminate();
end;

procedure move_to_buffer();
    buf_size : integer;
    /* Signal handler for changing buf_size */
    receive (main, buf_size);
    /* Repeat until termination: Find an empty entry in the buffer, copy a sample */
    /* into it, and inform write_to_disk that the entry is occupied */
    ...
    terminate();
end;

procedure write_to_disk();
    buf_size : integer;
    /* Signal handler for changing buf_size */
    receive (main, buf_size);
    /* Repeat until termination: Find an occupied entry in the buffer, copy the sample */
    /* from it onto disk, and inform analysis that the entry is occupied */
    ...
    terminate();
end;

procedure analysis();
    /* Perform statistical analysis and housekeeping */
    ...
    terminate();
end;

```

Fig. 3.3 Pseudo-code for a real time application

3.2.1 Sharing, Communication and Synchronization Between Processes

Processes of an application need to interact with one another because they work towards a common goal. Table 3.3 describes four kinds of process interaction. We summarize their important features in the following.

Table 3.3 Four kinds of process interaction

Interaction	Description
Data sharing	Shared data may become inconsistent if several processes update the data at the same time. Hence processes must interact to decide when it is safe for a process to access shared data.
Message passing	Processes exchange information by sending messages to one another.
Synchronization	To fulfill a common goal, processes must coordinate their activities and perform their actions in a desired order.
Signals	A signal is used to convey occurrence of an exceptional situation to a process.

Data sharing As discussed later in Section 3.6.1, a shared variable may get inconsistent values if many processes update it concurrently. To avoid this problem, a data access in one process may have to be delayed if another process is accessing the data. Thus data sharing by concurrent processes comes at a cost.

Message passing A process may put some information into a message and send it to another process. The other process can copy the information from the message into its own data structures. Both the sender and the receiver process must anticipate the information exchange, i.e., a process must know when it is expected to send or receive a message, so the information exchange becomes a part of the convention or protocol between processes.

Synchronization If an action a_i is to be performed only after an action a_j , the process that wishes to perform action a_i is made to wait until some other process performs a_j . An OS provides facilities to check if another process has performed a specific action.

Signals A signal is used to convey an exceptional situation to a process so that the process may perform some special actions to handle the situation. The code that a process wishes to execute on receiving a signal is called a *signal handler*. The signal mechanism is modeled along the lines of interrupts. Thus, when a signal is sent to a process the OS interrupts execution of the process and executes a signal handler specified by the process. Operating systems differ in the way they handle pending signals and resumption of processes after executing a signal handler.

Details of sharing, communication and synchronization between processes are

discussed in Section 3.6. Example 3.2 illustrates interactions between processes in the real time application of Figure 3.3.

Example 3.2 In the real time application of Figure 3.3, the variable `no_of_samples` is shared by the process that moves samples to the buffer and the process that writes them to the disk. The value of `no_of_samples` indicates how many samples exist in the buffer. Both processes need to update `no_of_samples`, so its consistency should be protected by delaying an access to `no_of_samples` if another process is accessing it. These processes also need to synchronize their activities such that a new sample is moved into a buffer entry only after the previous sample contained in it is written to the disk, and contents of a buffer entry are written to the disk only after a new sample is moved into it.

These processes also need to know the size of the buffer, i.e., the number of samples it can hold. Like `no_of_samples`, `size` could be used as shared data. However, this would cause delays in accessing its value. This delay is not justified because the buffer size is not updated regularly; it changes only in exceptional situations. Hence these processes are coded to use a local data item `buf_size` to indicate the size of the buffer. Its value is sent to them by the main process through messages. The main process sends signals to these processes if the size of the buffer has to be changed.

3.2.2 Concurrency and Parallelism

Parallelism is the quality of occurring at the same time. Two events are parallel if they occur at the same time, and two tasks are parallel if they are performed at the same time. *Concurrency* is an illusion of parallelism. Thus, two tasks are concurrent if there is an illusion that they are being performed in parallel, whereas, in reality, only one of them may be performed at any time.

In an OS, concurrency is obtained by interleaving operation of processes on the CPU, which creates the illusion that these processes are operating at the same time. Parallelism is obtained by using multiple CPUs, as in a multi-processor system, and operating different processes on these CPUs.

How does mere concurrency provide any benefits? We have seen several examples of this earlier in Chapter 2. In Section 2.5 we discussed how the throughput of a multiprogramming OS increases by interleaving operation of processes on a CPU, because an I/O operation in one process overlaps with a computational activity in another process. In Section 2.6, we saw how interleaved operation of processes created by different users in a time sharing system makes each user think that he has a computer to himself, although it is slower than the real computer being used. In Example 3.1, we saw that a computation may be completed faster, i.e., with a smaller elapsed time, due to interleaving of processes. We call this property *computation speed-up*.

Parallelism can provide better throughput in an obvious way because processes can operate on multiple CPUs. It can also provide computation speed-up; however, the computation speed-up provided by it is qualitatively different from that provided

through concurrency—when concurrency is employed, speed-up is obtained by overlapping I/O activities of one process with CPU activities of other processes, whereas when parallelism is employed, CPU and I/O activities in one process can overlap with the CPU and I/O activities of other processes.

When processes are used as the model of execution of programs, both concurrency and parallelism depend on there being sufficient work for the OS to perform. In Section 3.4, we introduce an alternative model of execution of programs, called *threads*, where concurrency and parallelism depend not only on availability of sufficient amount of work to perform, but also on the model of threads implemented in the OS.

3.3 OS VIEW OF PROCESSES

In the operating system's view, a process is an execution of a program. To realize this view, the OS creates processes, schedules them for use of the CPU, and terminates them. As discussed earlier in Sections 2.5.3.1 and 2.6.2, to perform scheduling an operating system must know which processes require the CPU at any moment. So, the key to realizing the operating system's view of processes is to monitor all processes and know what each process is doing at any moment of time—whether executing on the CPU, waiting for the CPU to be allocated to it, waiting for an I/O operation to complete, or waiting to be swapped into memory. The operating system uses the notion of *process state* to keep track of what a process is doing at any moment.

In this section, we discuss the notion of process state, different states of a process, and the arrangements used by the operating system to maintain information about the state of a process. We do not discuss scheduling in this chapter; it is discussed later in Chapter 4.

3.3.1 Background—Execution of Programs

When a process, i.e., an execution of a program, is scheduled, the CPU executes instructions in the program. When the CPU is to be switched to some other process, the kernel saves the state of the CPU. This state is loaded back into the CPU when the process is scheduled again. This view leads to the following definition of a process.

Definition 3.1 (Process) A process is comprised of six components:

$(id, code, data, stack, resources, CPU\ state)$

where *id* is a unique name/id assigned to the program execution,
code is the program code,
data is the data and files used in the program's execution,
resources is the set of resources allocated by the OS,
stack contains parameters of functions and procedures called, and their return addresses,

CPU state is comprised of contents of the PSW fields and the CPU registers.

As described in Section 2.1.1, the CPU state contains information like address of the instruction to be executed next, the condition code, which is also called *flags*, and contents of CPU registers. The CPU state changes as the program execution proceeds. The code, data and stack of the process constitute its *address space*.

3.3.2 Controlling Processes

A process is a program execution. Hence following Def. 3.1, a process has the five components process id, code, data, stack, and CPU state. The process uses the CPU when it is scheduled. It also uses other resources. These include system resources like memory and user-created resources like files. The OS has to maintain information about all these features of a process.

The OS view of a process consists of two parts:

- Code and data areas of the process, including its stack, and resources allocated to it
- Information concerning program execution.

Figure 3.4 shows the arrangement used to control a process. It consists of a *process environment* and the *process control block* (PCB). The id of a process is used to access its process environment and PCB. This arrangement enables different OS modules to access process-related data conveniently and efficiently.

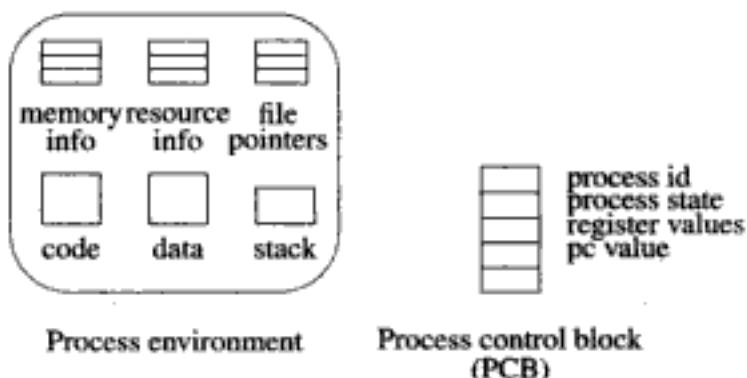


Fig. 3.4 OS view of a process

Process environment Also called the *process context*, the process environment contains the address space of a process, i.e., its code, data, and stack, and all information necessary for accessing and controlling resources allocated to the process. The OS creates a process environment by allocating memory to the process, loading the process code in the allocated memory and setting up its data space. The OS also

puts in information concerning access to resources allocated to the process, and its interaction with other processes and with the OS.

Table 3.4 describes components of the process environment. Contents of the process environment change during execution of a process due to actions like file open and close, and dynamic creation and destruction of data by the process.

Table 3.4 Components of the process environment

Process environment	
Process environment consists of the code and data of the process and all information necessary for operation of the process.	
Component	Description
Code and data	Code of the program, including its functions and procedures, and its data, including the stack.
Memory allocation information	Information concerning memory areas allocated to the process. This information is used to implement memory accesses made by the process.
Status of file processing activities	Pointers to files opened by the process, and current positions in the files
Process interaction information	All information necessary to control interactions of the process with other processes, e.g., interprocess messages, signal handlers, ids of parent and child processes.
Resource information	Information concerning resources allocated to a process.
Miscellaneous information	Miscellaneous information needed for interaction of a process with the OS.

Process control block (PCB) The process control block is a kernel data structure that contains information concerning the first and fifth component of Def. 3.1, that is, process id and CPU state. Details of the PCB are described in Section 3.3.4.

The kernel uses three fundamental functions to control processes:

1. *Scheduling*: Select the process to be executed next on the CPU.
2. *Dispatching*: Set up execution of the selected process on the CPU.
3. *Context save*: Save information concerning a running process when its execution is suspended.

The scheduling function selects a process based on the scheduling policy in force. Dispatching involves setting up the environment of the selected process, and loading information into the CPU so that CPU begins (or resumes) execution of instructions in the process code. The context save function performs housekeeping whenever a process releases the CPU or is preempted. It saves information concerning the CPU state and the process environment so that execution of the process may be resumed sometime in future. Effectively, context save is the converse of the dispatching function.

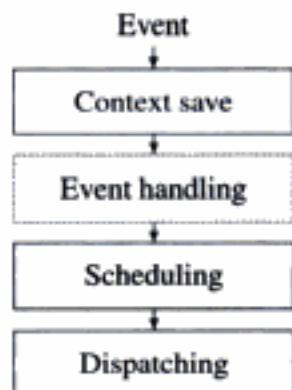


Fig. 3.5 Fundamental functions to control processes

The schematic in Figure 3.5 illustrates use of these functions by the kernel when an event occurs. Occurrence of an event invokes the context save function and an appropriate event processing routine. The event processing actions may activate some processes, hence the scheduling function is invoked to select a process and the dispatching function transfers control to the selected process.

3.3.3 Process States and Transitions

The notion of *process state* is introduced to simplify control of processes by the operating system.

Definition 3.2 (Process state) *The process state is an indicator of the nature of the current activity in a process.*

Table 3.5 describes four fundamental process states. A process is in the *blocked* state if it has made a resource request to the kernel and the request is yet to be granted, or if it wishes to wait until some event occurs. The process enters the *ready* state when the request is granted or the event occurs, and it enters the *running* state when it is dispatched. The process enters the *terminated* state when its execution completes or when it is terminated by the OS for some reason.

A conventional computer system contains one CPU, so at most one process can be in the *running* state. Any number of processes can however exist in the *blocked*, *ready* and *terminated* states. An OS may define more process states to simplify its own functioning or to support additional functionalities like swapping.

Process state transitions A *state transition* for a process P_i is a change in its state. A state transition is caused by the occurrence of some event in the system. When a process P_i in the *running* state makes an I/O request, it has to enter the *blocked* state until its I/O operation completes. At the end of the I/O operation P_i 's state changes from *blocked* to *ready*. Similar state changes occur when a process makes some

Table 3.5 Fundamental process states

State	Description
<i>Running</i>	A CPU is currently executing instructions in the process code.
<i>Blocked</i>	The process has to wait until a resource request made by it is granted, or it wishes to wait until a specific event occurs. A CPU should not be allocated to it until its wait is complete.
<i>Ready</i>	The process wishes to use the CPU to continue its operation; however, it has not been scheduled.
<i>Terminated</i>	The operation of the process, i.e., the execution of the program represented by it, has completed normally, or the OS has aborted it.

request that cannot be satisfied by the OS straightaway. The process state changes to *blocked* when the request is made, i.e., when the request event occurs, and changes to *ready* when the request is satisfied. A *ready* process becomes *running* when the CPU is allocated to it. A *running* process becomes *ready* when it is preempted either because a higher priority process becomes ready or because its time slice elapses (see Sections 2.5.3.1 and 2.6).

Figure 3.6 shows the fundamental state transitions for a process. Table 3.6 summarizes causes of state transitions. A new process is put in the *ready* state after all resources required by it have been allocated. It may enter the *running*, *blocked* and *ready* states a number of times due to events described in Table 3.6. Eventually it enters the *terminated* state.

**Fig. 3.6** Fundamental state transitions for a process

Example 3.3 Consider the time sharing system of Example 2.12, which uses a time slice of 10 msec. It contains two programs P_1 and P_2 . P_1 has a CPU burst of 15 msec followed by an I/O operation that lasts for 100 msec, while P_2 has a CPU burst of 30 msec followed by an I/O operation that lasts for 60 msec. Execution of P_1 and P_2 was described in Figure 2.21 and Table 2.13. For simplicity, we assume that processes created to execute programs P_1 and P_2 are themselves called P_1 and P_2 .

Table 3.6 Causes of fundamental state transitions for a process

State transition	Description
<i>ready</i> → <i>running</i>	The process is dispatched. CPU starts or resumes execution of its instructions
<i>blocked</i> → <i>ready</i>	A request made by the process is satisfied or an event for which it was waiting occurs.
<i>running</i> → <i>ready</i>	The process is preempted because the OS decides to schedule some other process. This transition occurs either because a higher priority process becomes <i>ready</i> , or because the time slice of the process elapses.
<i>running</i> → <i>blocked</i>	<p>The program being executed makes a system call to indicate that it wishes to wait until some resource request made by it is satisfied, or until a specific event occurs in the system. Five major causes of blocking are:</p> <ul style="list-style-type: none"> • Process requests an I/O operation • Process requests memory or some other resource • Process wishes to wait for a specified interval of time • Process waits for message from another process • Process wishes to wait for some action by another process
<i>running</i> → <i>terminated</i>	<p>Execution of the program is completed or terminated. Five primary reasons for process termination are as follows:</p> <ul style="list-style-type: none"> • <i>Self-termination</i>: The program being executed either completes its task or realizes that it cannot execute meaningfully and makes a 'terminate me' system call. Examples of the latter condition are incorrect or inconsistent data, and inability to access data in a desired manner, e.g., incorrect file access privileges. • <i>Termination by a parent</i>: A process makes a 'terminate P_i' system call to terminate a child process P_i, when it finds the execution of the child process is no longer necessary or meaningful. (The same effect can be achieved by the parent sending a 'terminate now' signal to the child process and the child process making a 'terminate me' system call.) • <i>Exceeding resource utilization</i>: An OS may limit the resources that a process may consume. A process exceeding a resource limit would be terminated by the kernel. • <i>Abnormal conditions during execution</i>: The kernel cancels a process if an abnormal condition arises in the program being executed, e.g., execution of an invalid instruction, execution of a privileged instruction, arithmetic conditions like overflow, memory protection violation, etc. • <i>Incorrect interaction with other processes</i>: The kernel may cancel a process for incorrect interaction with other processes, e.g., if a process gets involved in a deadlock.

respectively. Actual execution of programs proceeds as follows: Table 3.7 illustrates the state transitions in the system. System operation starts with both processes in *ready* state at time 0. The scheduler selects process P_1 for execution and changes its state to *running*. At 10 msec, P_1 is preempted and P_2 is scheduled, so P_1 's state is changed to *ready* and P_2 's state is changed to *running*. P_1 gets blocked at 25 msec because of an I/O operation. At 35 msec, P_2 is preempted because its time slice elapses, however it is scheduled again since no other process is in the *ready* state. P_1 's I/O operation is completed at 125 msec, so its state is changed to *ready*. The scheduler selects it for execution, hence its state is further changed to *running*.

Table 3.7 State transitions in a time sharing system

Time	Event	Remarks	New state	
			P_1	P_2
0		P_1 is scheduled	<i>running</i>	<i>ready</i>
10	P_1 is preempted	P_2 is scheduled	<i>ready</i>	<i>running</i>
20	P_2 is preempted	P_1 is scheduled	<i>running</i>	<i>ready</i>
25	P_1 starts I/O	P_2 is scheduled	<i>blocked</i>	<i>running</i>
35	P_2 is preempted	P_2 is scheduled	<i>blocked</i>	<i>ready</i>
45	P_2 starts I/O		<i>blocked</i>	<i>running</i>
125	I/O interrupt	P_1 becomes <i>ready</i> P_1 is scheduled	<i>ready</i> <i>running</i>	<i>blocked</i> <i>blocked</i>

3.3.3.1 Suspended Processes

In addition to the four fundamental process states described in Table 3.5, an OS uses a fifth state called *suspend* state for a process that is not to be considered for scheduling. The state of such a process is changed to *ready* or *blocked* when its operation is to be resumed. The *suspend* state differs from the *blocked* state in that the reason for suspension is external to the activity in the process, while the reason for blocking is internal to its own activity. Two typical causes of suspension are:

- A process is moved out of memory, i.e., it is swapped out.
- The user who initiated a process specifies that a process should not be scheduled until some condition is satisfied.

The process state is an abstract notion defined by an OS designer and used by the kernel to simplify control of processes. A kernel may use a single *suspend* state and use some auxiliary information to decide whether a process in the *suspend* state should enter the *ready* state or the *blocked* state when its operation is to be resumed. Alternatively, the kernel may split the *suspend* state into a number of states depending on the cause of suspension. This approach simplifies specification of state transitions into and out of a *suspend* state.

We restrict the discussion of suspended processes to swapped processes and use two suspend states called *ready swapped* and *blocked swapped*. Accordingly, Figure 3.7 shows process states and state transitions. Transitions from the *ready* to *ready*

swapped state and *blocked* to *blocked swapped* state are caused by a swap-out action. The reverse transitions take place when these processes are swapped back into the memory. The *blocked swapped* \rightarrow *ready swapped* transition takes place if the request for which the process was waiting is granted even while the process is in a suspended state, for example if a resource for which it was blocked is granted to it. However, the process continues to be swapped out. When it is swapped back into the memory, its state would change to *ready* and it would compete with other *ready* processes for the CPU's attention.

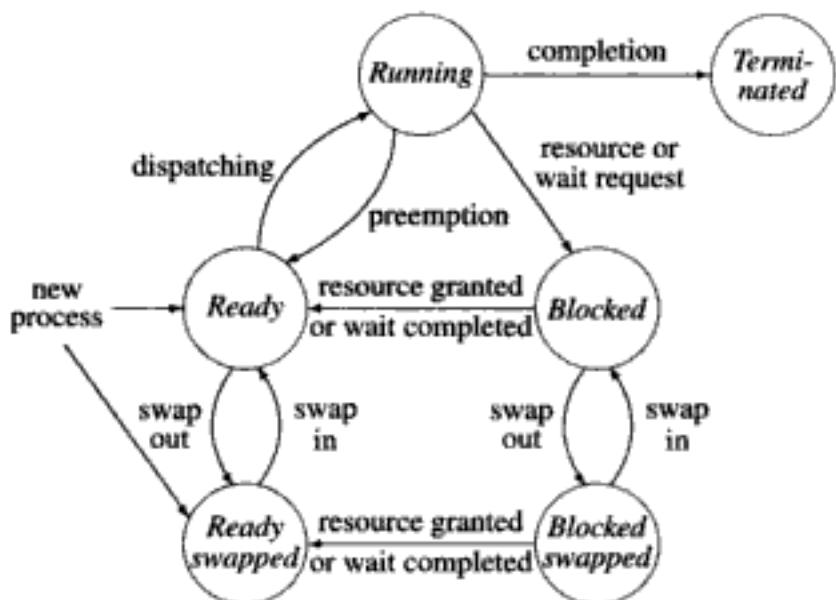


Fig. 3.7 Process states and state transitions using two swapped states

3.3.4 Process Control Block

The *process control block* (PCB) contains all information pertaining to a process that is used in controlling its operation—such as its id, priority, state, PSW and contents of CPU registers—as also information used in accessing resources and implementing communication with other processes. Table 3.8 describes fields of the PCB data structure.

The priority and state information is used by the scheduler. It passes the id of the selected process to the dispatcher. For a process that is not in the *running* state, the *PSW* and *CPU registers* fields together contain a snapshot of the CPU when it was last released by the process, i.e., they hold contents of the various control and data registers of the CPU when the process got blocked or was preempted. Execution of the process can be resumed by simply loading this information from its PCB into the

Table 3.8 Fields of the Process Control Block (PCB)

PCB field	Contents
Process id	The unique id assigned to the process at its creation.
Child and parent ids	These ids are used for process synchronization, typically for a process to check whether a child process has terminated.
Priority	The priority is typically a numeric value. A process is assigned a priority at its creation. Priority may change during the life of a process depending on its nature (whether CPU-bound or I/O-bound), its age and the resources consumed by it (typically CPU time).
Process state	The current state of the process.
PSW	This is a snapshot, i.e., an image, of the PSW when the CPU was last voluntarily released by the process or was last preempted by the kernel. Loading this snapshot back into the PSW would resume execution of the program. (See Fig. 2.3 for fields of the PSW.)
CPU registers	Contents of the registers when the CPU was last released by the process or last preempted by the kernel.
Event information	For a process in the <i>blocked</i> state, this field contains information concerning the event for which the process is waiting. When an event occurs, the kernel uses this information to identify the process that awaits the event and changes its state from <i>blocked</i> to <i>ready</i> .
Signal information	Information concerning locations of signal handlers (see Section 3.6.4).
PCB pointer	This field is used to form a list of PCBs. The kernel maintains several lists of PCBs, e.g., a list of PCBs of <i>ready</i> processes, a list of PCBs of <i>blocked</i> processes, etc.

CPU. This action would be performed the next time this process is to be dispatched.

The *event information* field of the PCB plays an important role in effecting appropriate state transitions in the system. Consider a process P_i that is blocked on an I/O operation. The *event information* field in P_i 's PCB indicates the id of the device, say device d , on which the I/O operation is being performed. When the I/O operation on device d completes, the kernel must find the process that awaits that event and change its state to *ready*. *event information* field of P_i 's PCB indicates that it awaits the end of I/O on device d , so the kernel concludes that the state of P_i should be changed from *blocked* to *ready*.

3.3.5 Context Save, Scheduling and Dispatching

The context save function performs housekeeping whenever a process releases the CPU or is preempted. It involves saving the PSW and CPU registers in appropriate

fields of the PCB of the process, and also saving information concerning its process environment (see Section 3.3.2). It also marks the state of the process *as ready*. An event processing action may later change the state to *blocked*.

The scheduling function uses the process state information from PCBs to select a *ready* process for execution. It changes the state of the process to *ready* and passes its id to the dispatching function. The dispatching function sets up the environment of the selected process and loads information from the *PSW* and *CPU registers* fields of its PCB into the CPU.

Example 3.4 An OS contains 2 processes P_1 and P_2 , with P_2 having a higher priority than P_1 . Let P_2 be *blocked* on an I/O operation and let P_1 be *running*. The following actions take place when the I/O completion event occurs for the I/O operation of P_2 :

1. The context save function is performed for P_1 and its state is changed to *ready*.
2. The I/O completion event is processed and the state of P_2 is changed from *blocked* to *ready*.
3. Scheduling is performed. P_2 is selected because it is the highest priority *ready* process.
4. P_2 is dispatched.

Process switching Functions 1, 3 and 4 of Example 3.4 collectively perform switching between processes P_1 and P_2 . Switching between processes also occurs when a running process gets blocked due to a request or gets preempted. An event does not lead to switching between processes if occurrence of the event either causes a state transition for a lower priority process, or does not cause any state transition (e.g., if the event is caused by a request that is satisfied straightaway). In the former case, the scheduling function selects the interrupted process itself for dispatching. In the latter case, scheduling need not be performed at all; after event processing, control can be given to the dispatcher, which would simply dispatch the interrupted process.

Switching between processes is more than saving the CPU state of one process and loading CPU state of another process. The process environment needs to be switched as well. Saving and loading of memory management information is an expensive part of process switching, particularly when virtual memory is used. Some computer systems provide special instructions to reduce the process switching overhead, e.g., an instruction that saves or loads the PSW and all CPU registers. We use the term *state information of a process* to refer to all the information that needs to be saved and restored during process switching. Process switching overhead depends on the size of this information.

Process switching has some indirect overhead as well. The newly scheduled process does not have any part of its address space in the cache, hence it performs poorly until it builds sufficient information in the cache. Virtual memory operation is also poorer initially because translation buffers do not contain any information relevant to the newly scheduled process.

3.3.6 Events Pertaining to a Process

The following events occur during the operation of an OS:

1. *Process creation event*: A new process is created.
2. *Process termination event*: A process finishes its execution.
3. *Timer event*: The timer interrupt occurs.
4. *Resource request event*: Process makes a resource request.
5. *Resource release event*: A resource is released.
6. *I/O initiation request event*: Process wishes to initiate an I/O operation.
7. *I/O completion event*: An I/O operation completes.
8. *Message send event*: A message is sent by one process to another.
9. *Message receive event*: A message is received by a process.
10. *Signal send event*: A signal is sent by one process to another.
11. *Signal receive event*: A signal is received by a process.
12. *A program interrupt*: An instruction executed in the running process malfunctions.

The timer and I/O completion events are caused by situations that are external to the running process. All other events are caused by actions in the *running* process. We group events 1–9 into two broad classes for discussing actions of event handling routines, and discuss events 10 and 11 in Section 3.6.4. The kernel performs a standard action like aborting the *running* process when event 12 occurs.

Events pertaining to process creation, termination and preemption A new process can be created in two ways—through the user interface or through another process. When a user issues a command to execute a program, the command interpreter of the user interface invokes some non-kernel program which makes the *create_process* system call mentioned in Section 3.2 with the name of the program as a parameter. When a process wishes to create a child process to execute a program, it itself makes a *create_process* system call with the name of the program as a parameter.

The event handling routine for the *create_process* system call creates a PCB for the new process, assigns a unique process id and a priority to it, and puts this information and id of the parent process into relevant fields of the PCB. It now determines the amount of memory required to accommodate the address space of the process, i.e., the code and data of the program to be executed and its stack, and arranges to allocate this much memory to the process (memory allocation techniques are discussed later in Chapters 5 and 6). Most operating systems associate some standard resources with each process, like a monitor and standard input and output files. The kernel allocates these standard resources to the process at this time. It now enters information about allocated memory and resources into the context of the new process. After completing these chores, it sets the state of the process to *ready* in its PCB and enters this process in an appropriate PCB list.

When a process makes a system call to terminate itself or terminate a child process, the kernel delays termination until the I/O operations that were initiated by the process are completed. It now releases the memory and resources allocated to it. This function is performed using the information in appropriate fields of the process context. The kernel now changes the state of the process to *terminated*. Two additional provisions are necessary because the parent of the terminated process may wish to check its status. First, if the parent of the process is already waiting for its termination, the kernel must activate the parent process. To perform this action, the kernel takes the id of the parent process from the PCB of the terminated process, and checks the *event information* field of the parent process' PCB to find whether the parent process is waiting for termination of the child process (see Section 3.3.4). Second, to enable the parent process to check the status of the terminated process sometime in future, the PCB of the terminated process is not destroyed now; it will be done sometime after the parent process terminates.

The process in the *running* state should be preempted if its time slice elapses. The context save function would have already changed the state of the running process to *ready* before invoking the event handler for timer interrupts, so the event handler simply informs the scheduler about the end of the time slice and the scheduler moves the PCB of the process to an appropriate scheduling list. Preemption should also occur when a higher priority process becomes *ready*, but that is implicit in scheduling, so an event handling routine need not perform any explicit action for it.

Events pertaining to resource utilization When a process requests a resource through a system call, the kernel may be able to allocate the resource immediately, in which case event handling does not cause any process state transitions, so the interrupted process would be scheduled and dispatched again. To reduce overhead, the kernel can skip scheduling in such cases and directly invoke the dispatcher to resume execution of the interrupted process. If the resource cannot be allocated, the event handling routine changes the state of the interrupted process to *blocked* and notes the id of the resource in the *event information* field of the PCB.

When a process releases a resource through a system call, the event handling routine need not change the state of the process that made the system call. However, it should check whether any other processes were blocked because they needed the resource, and, if so, it should allocate the resource to one of the blocked processes and change its state to *ready*. This action requires a special arrangement that we will discuss shortly.

A system call to request initiation of an I/O operation and an interrupt signaling end of the I/O operation lead to event handling actions that are analogous to a request for a resource and allocation of the resource at a later time. The state of the process is changed to *blocked* when the I/O operation is initiated and the cause of blocking is noted in the *event information* field of its PCB; its state is changed back to *ready* when the I/O operation completes. A request to receive a message from another

process and a request to send a message to another process also lead to analogous actions.

Event control block (ECB) When an event occurs, the kernel must find the process that is affected by it. For example, when an I/O completion interrupt occurs, the kernel must identify the process awaiting its completion. It can achieve this by searching the *event information* field of the PCBs of all processes. This search is expensive, and operating systems use various schemes to speed it up. We discuss a scheme that uses *event control blocks* (ECBs).

As shown in Figure 3.8, an ECB contains three fields. The *event description* field describes an event, and the *process id* field contains the id of the process awaiting the event. When a process P_i gets blocked for occurrence of an event e_i , the kernel forms an ECB and puts relevant information concerning e_i and P_i into it. The kernel can maintain a separate ECB list for each class of events like interprocess messages or I/O operations, so the *ECB pointer* field is used to enter the newly created ECB into an appropriate list of ECBs.

Event description
Process id of awaiting process
ECB pointer

Fig. 3.8 Event Control Block (ECB)

When an event occurs, the kernel scans the appropriate list of ECBs to find an ECB with a matching event description. The *process id* field of the ECB indicates which process is waiting for the event to occur. The state of this process is changed to reflect the occurrence of the event. The following example illustrates use of ECBs to handle an I/O completion event; their use in handling interprocess messages is described in Section 10.2.2.

Example 3.5 The actions of the kernel when process P_i requests an I/O operation on some device d , and when the I/O operation completes, are as follows:

1. The kernel creates an ECB, and initializes it as follows:
 - (a) Event description = End of I/O on device d
 - (b) Process awaiting the event = P_i .
2. The newly created ECB (let us call it ECB_j) is added to a list of ECBs.
3. The state of P_i is changed to *blocked* and the address of ECB_j is put into the 'Event information' field of P_i 's PCB (see Figure 3.9).
4. When the interrupt 'End of I/O on device d ' occurs, ECB_j is located by searching for an ECB with a matching event description field.
5. The id of the affected process, i.e., P_i , is extracted from the ECB_j . The PCB of P_i is located and its state is changed to *ready*.

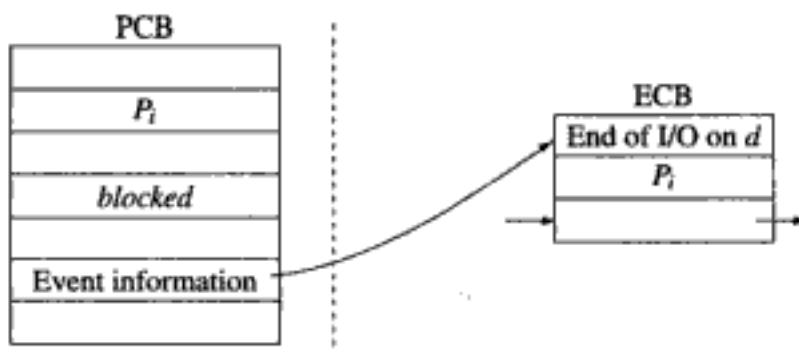


Fig. 3.9 PCB-ECB interrelationship

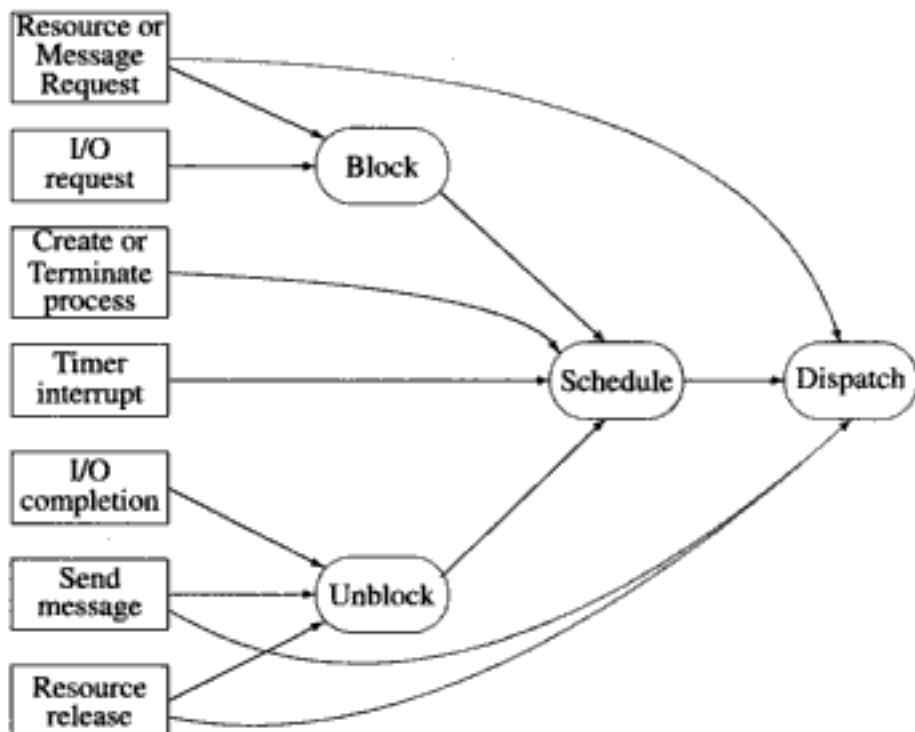


Fig. 3.10 Event handling actions of the kernel

Summary of event handling Figure 3.10 illustrates event handling actions of the kernel described earlier. The *block* action always changes the state of the process that made a system call from *ready* to *blocked*. The *unblock* action finds a process whose request can be fulfilled now and changes its state from *blocked* to *ready*. A system call for requesting a resource leads to a *block* action if the resource cannot be allocated to the requesting process. This action is followed by scheduling and dispatching because another process has to be selected for use of the CPU. The *block*

action is not performed if the resource can be allocated straightaway. In this case, the interrupted process is simply dispatched again. When a process releases a resource, an *unlock* action is performed if some other process is waiting for the released resource, followed by scheduling and dispatching because the unblocked process may have a higher priority than the process that released the resource. Again, scheduling is skipped if no process is unblocked due to the event.

3.4 THREADS

Use of processes to provide concurrency within an application incurs high process switching overhead (see Section 3.3.5). Threads provide a low cost method of implementing concurrency that is suitable for certain kinds of applications.

Process switching overhead has two components:

- *Execution related overhead*: A process is defined as an execution of a program. Hence, while switching between processes, the CPU state of the running process has to be saved and the CPU state of a new process has to be loaded in the CPU. This overhead is unavoidable.
- *Resource use related overhead*: As discussed in Section 3.3, the process environment contains information concerning resources allocated to a process and its interaction with other processes. It leads to a large size of process state information, which adds to the process switching overhead.

Switching overhead can be reduced by eliminating the resource related overhead in some situations. Consider the context save, scheduling and dispatching functions depicted in Figure 3.5 and discussed in Section 3.3.5. Occurrence of an event may result in switching from execution of a *running* process P_i to execution of some other process P_j . If both P_i and P_j belong to the same application, they share the code, data and resources; their state information differs only in the values contained in CPU registers and stacks. Much of the saving and loading of process state information while switching from P_i to P_j is thus redundant. This feature is exploited to achieve a reduction in switching overhead. The notion of a thread is used for this purpose.

Definition 3.3 (Thread) *A thread is a program execution that uses the resources of a process.*

Since a thread is a program execution, it has its own stack and CPU state (see Def. 3.1). We use the phrases ‘thread(s) of a process’ and ‘parent process of a thread’ to describe the relationship between a thread and the process whose environment it uses. Threads of the same process share code, data and resources with one another. The process abstraction continues to be used as before except that processes typically have distinct code and data parts.

Figure 3.11 depicts the relationship between threads and processes. Process P_i has three threads represented by the wavy lines. The kernel allocates a stack and

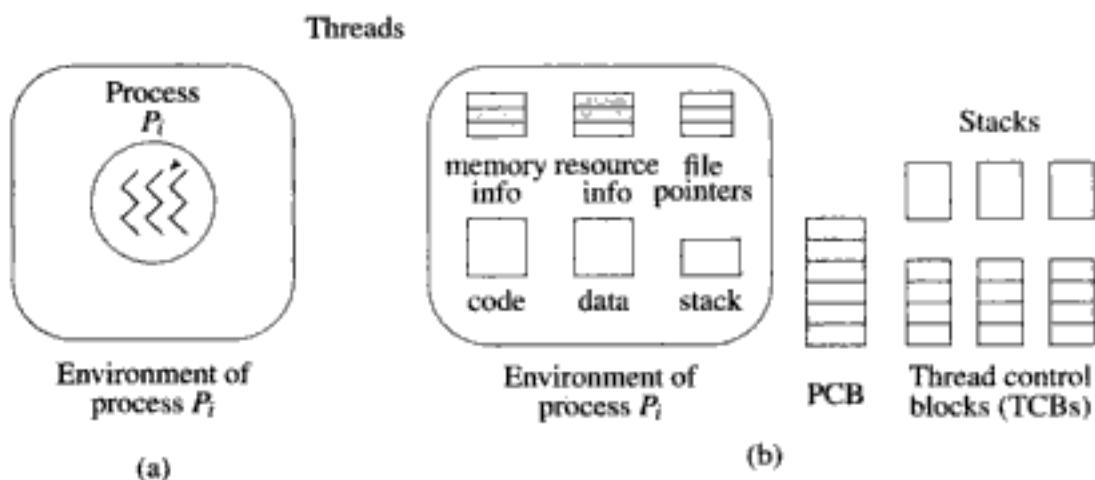


Fig. 3.11 Threads in process P_i : (a) concept, (b) implementation

a *thread control block* (TCB) to each thread. The threads execute within the environment of P_i . The OS is aware of this fact, so it saves only the CPU state and the stack pointer while switching between threads of the same process. In most computer systems, the stack pointer is maintained in a CPU register, hence only the CPU state needs to be switched. Use of threads effectively splits the process state into two parts—resource state remains with the process while execution state is associated with a thread. The cost of concurrency within the environment of a process is now merely replication of the execution state for each thread. The resource state is not replicated.

Thread states and state transitions Barring the difference that threads do not have resources allocated to them, threads and processes are analogous. Hence thread states and thread state transitions are analogous to process states and process state transitions. When a thread is created, it is put in the *ready* state because its parent process already has the necessary resources allocated to it. It enters the *running* state when it is scheduled. It does not enter the *blocked* state because of resource requests, because it does not make any resource requests; however, it can enter the *blocked* state because of process synchronization requirements. For example, if threads were used in the real time data logging application of Example 3.1, thread *disk_write* would have to enter the *blocked* state if no data samples exist in *buffer_area*.

Advantages of threads An application process can create many threads to execute its code. It is interesting to see the advantages of this scenario over creation of many processes to execute the application code. The fundamental advantage is that of low overhead while switching the CPU from one thread to another thread of the same process. The resource state is switched only while switching between threads of different processes.

Use of threads provides concurrency within a process. It can provide computa-

tion speed-up—if one thread of a process blocks on an I/O operation, the CPU can be switched to another thread of the same process. Use of threads can also simplify the design and coding of applications that need to perform concurrent servicing of requests. For example, in an airline reservation system or a banking system, a new thread can be created to handle each new request. The OS would schedule these threads to provide concurrency. This arrangement obviates complex logic to achieve concurrent processing of requests.

Creation of many processes within the application code can achieve the same effect. However, there is one significant difference. Concurrent activities, whether processes or threads, have to communicate or synchronize with one another. When these activities are implemented by processes, communication or synchronization involves the kernel. Processes have to make system calls to indicate their communication or synchronization requirements to the kernel. However, threads of a process share their data space. Hence they can communicate or synchronize using shared memory, thereby eliminating expensive system calls. Therefore in the real time application of Example 3.1, it would be preferable to use threads to implement functions 1 and 2, which share the buffer area.

Table 3.9 summarizes the advantages of threads. As discussed in Example 3.1, an application can create many threads to obtain concurrency and simplify its functioning. Threads can be assigned specific functions. They can also be created to perform concurrent servicing of requests received by a server, e.g., in an airline reservation system or a banking system. In such cases, threads would execute the same function.

Table 3.9 Advantages of threads

Advantage	Explanation
Low overhead	Thread state consists only of the state of a computation. Resource allocation state and communication state is not a part of thread state, which leads to low switching overhead.
Speed-up	Concurrency <i>within</i> a process can be realized by creating many threads in it. This technique can speed up execution of an application on both uniprocessors and multiprocessors.
Efficient communication	Threads of a process can communicate with one another through shared data space, thus avoiding the overhead of system calls for communication.

Coding for use of threads Threads should ensure correctness of data sharing and synchronization (see Section 3.2.1). Correctness of data sharing also has another facet. Functions or subroutines that use static or global data to carry values across their successive activations produce incorrect results when invoked concurrently, because the concurrent invocations effectively perform data sharing without mutual

exclusion. Such routines are said to be *thread unsafe*. An application that uses threads must be coded in a *thread safe* manner and must invoke routines only from a thread safe library.

Signal handling requires special attention in a multi-threaded application. Recall that the kernel permits a process to specify signal handlers (see Section 3.6.4). When several threads are created in a process, which thread should handle a signal? There are several possibilities. The kernel may select one of the threads for signal handling. This choice can be made either statically, e.g., either the first or the last thread created in the process, or dynamically, e.g., the highest priority thread. Alternatively, the kernel may permit an application to specify which thread should handle signals at any time.

A synchronous signal arises due to the activity in a thread, so it is best that the thread itself handles it. Ideally, each thread should be able to specify which synchronous signals it is interested in handling. However, to provide this feature, the kernel would have to replicate the signal handling arrangement of Figure 3.11 for each thread, so few operating systems provide it. An asynchronous signal can be handled by any thread in a process. To ensure prompt attention to the condition that caused the signal, the highest priority thread should handle such a signal.

Implementation of threads Threads are implemented in different ways. The main difference is in how much the kernel and the application program know about the threads. These differences lead to different implications for overhead and concurrency in an application program.

In this Section we discuss three methods of implementing threads. These are:

- Kernel-level threads
- User-level threads
- Hybrid threads

Experimental studies have shown that switching between kernel-level threads of a process is over 10 times faster than switching between processes, and switching between user-level threads is over 100 times faster than switching between processes. Kernel-level threads provide better parallelism and speed-up in a multiprocessor system.

3.4.1 Kernel-Level Threads

A kernel-level thread is implemented by the kernel. Hence creation and termination of kernel-level threads, and checking their status, is performed through system calls analogous to those discussed in Section 3.2. Figure 3.12 shows a schematic of how the kernel handles kernel-level threads. When a process makes a *create thread* system call, the kernel assigns an id to it, and allocates a thread control block (TCB). The TCB contains a pointer to the PCB of the process.

When an event occurs, the kernel saves the CPU state of the interrupted thread in its TCB. After event handling, the scheduler considers TCBs of all threads and

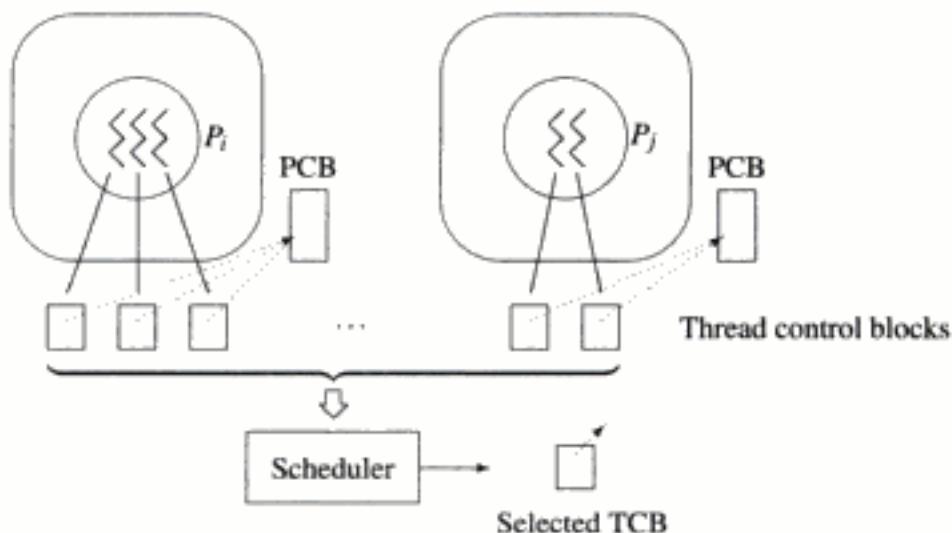


Fig. 3.12 Scheduling of kernel-level threads

selects one *ready* thread; the dispatcher uses the PCB pointer in its TCB to check if the selected thread belongs to a different process than the interrupted thread. If so, it saves the context of the process to which the interrupted thread belongs, and loads the context of the process to which the selected thread belongs. It then dispatches the selected thread. Actions to save and load the process context are unnecessary if both threads belong to the same process. This feature reduces the switching overhead.

Advantages and disadvantages of kernel-level threads A kernel-level thread is like a process except that it has a smaller amount of state information, i.e., it has a ‘thinner’ state. This similarity between threads and processes is convenient for programmers—programming for threads is no different from programming for processes. In a multiprocessor system, kernel-level threads provide parallelism (see Section 3.2.2), i.e., several threads belonging to a process can be scheduled simultaneously, which is not possible using the user-level threads described in the next section, so it provides better computation speed-up than user-level threads.

However, handling threads like processes has its disadvantages too. Switching between threads is performed by the kernel as a result of event handling. Hence it incurs the overhead of event handling even if the interrupted thread and the selected thread belong to the same process. This feature limits the savings in the switching overhead.

3.4.2 User-Level Threads

User-level threads are implemented by a *thread library*, which is linked to the code of a process. The library sets up the thread implementation arrangement shown in

Figure 3.11(b) without involving the kernel, and interleaves operation of threads in the process. Thus, the kernel is not aware of presence of user-level threads in a process; it sees only the process. The scheduler considers PCBs and selects a *ready* process; the dispatcher dispatches it. Most OSs implement the pthreads application program interface provided in the IEEE Posix standard in this manner.

An overview of creation and operation of threads is as follows: A process invokes the library function *create_thread* to create a new thread. The library function creates a TCB for the new thread and starts considering the new thread for 'scheduling'. When the thread in the *running* state invokes a library function to synchronize its functioning with other threads, the library function performs 'scheduling' and switches to another thread of the process. This amounts to a thread switch. Thus, the kernel is oblivious to switching between threads; it believes that the *process* is continuously in operation. If the thread library cannot find a ready thread in the process, it makes a 'block me' system call. The kernel now blocks the process. The process will be unblocked when some event activates one of its threads and will resume execution of the thread library function, which will perform 'scheduling' and switch to execution of the newly activated thread.

Scheduling of user-level threads Figure 3.13 is a schematic diagram of scheduling of user-level threads. The thread library code is a part of each process. It performs 'scheduling' to select a thread, and organizes its execution. We view this operation as mapping of the TCB of the selected thread into the PCB of the process.

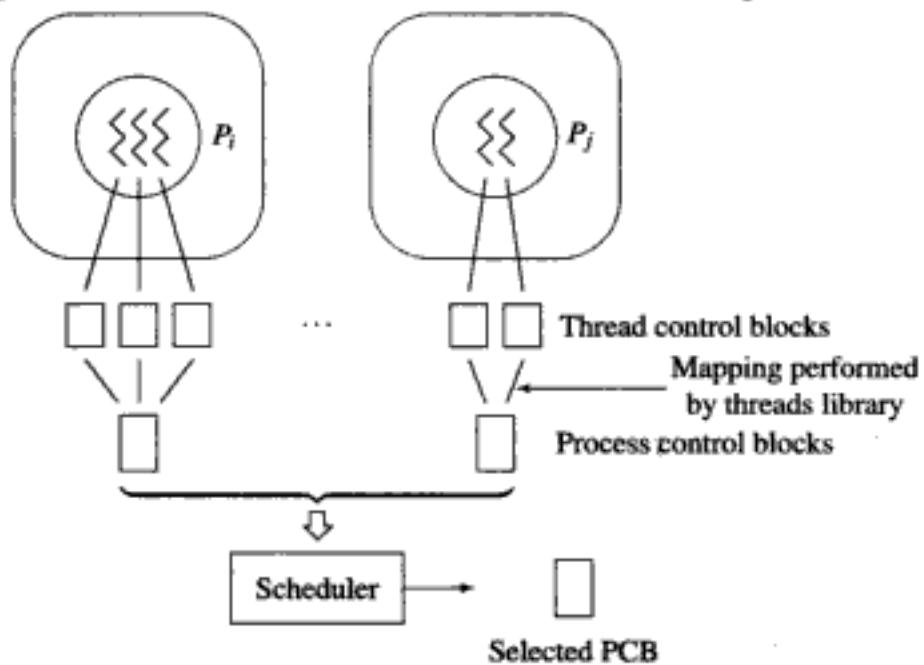


Fig. 3.13 Scheduling of user-level threads

The thread library uses information in the TCBs to decide which thread should

operate at any time. To ‘dispatch’ the thread, the CPU state of the thread should become the CPU state of the process, and the process stack pointer should point to the thread’s stack. Since the thread library is a part of a process, the CPU is in the non-privileged mode. Hence a thread cannot be dispatched by loading new information into the PSW; the thread library has to use non-privileged instructions to change PSW contents, so it loads the address of the thread’s stack into the stack address register, and executes a branch instruction to transfer control to the next instruction of the thread. The next example illustrates interesting situations during scheduling of user-level threads.

Example 3.6 Figure 3.14 illustrates how the thread library manages three threads in a process P_i . The codes N , R and B in the TCBs represent the states *running*, *ready* and *blocked*, respectively. Process P_i is in the *running* state and the thread library is executing. It dispatches thread h_1 . Process P_i is preempted sometime later by the kernel. Figure 3.14 (a) illustrates states of the threads and of process P_i . Thread h_1 is in the *running* state, and process P_i is in the *ready* state. Thread h_1 would resume its operation when process P_i is scheduled next. The line from h_1 ’s TCB to P_i ’s PCB indicates that h_1 ’s TCB is currently mapped into P_i ’s PCB. This fact is important for the dispatching and context save actions of the thread library.

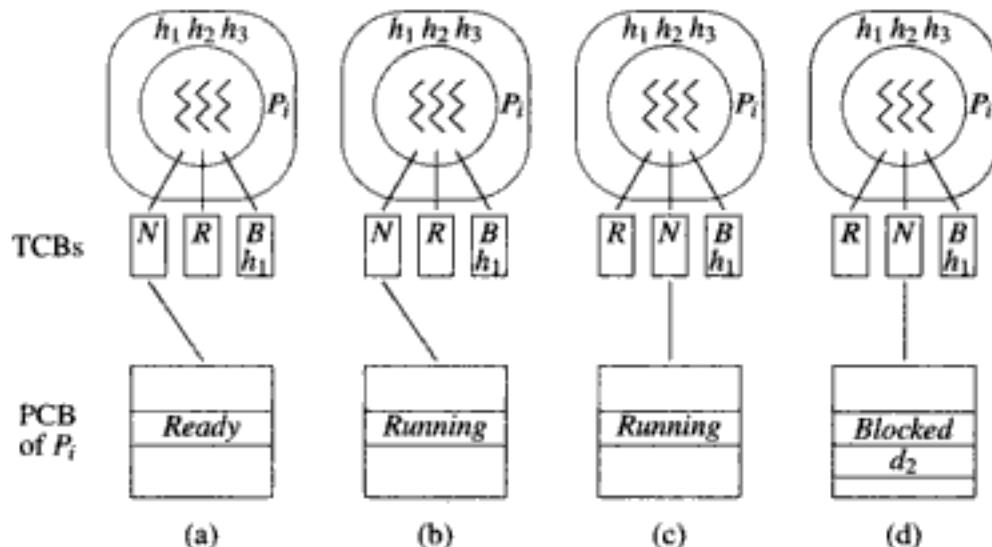


Fig. 3.14 Actions of the thread library (N, R, B indicate *running*, *ready* and *blocked*)

Thread h_2 is in the *ready* state in Figure 3.14(a), so its TCB contains the code R . Thread h_3 awaits a synchronization action by h_1 , so it is in the *blocked* state. Its TCB contains the code B , and h_1 to indicate that it is awaiting an event that is a synchronization action by h_1 . Figure 3.14(b) shows the situation when the kernel dispatches P_i and changes its state to *running*.

The thread library overlaps operation of threads using the timer. When thread h_1 was ‘scheduled’, it would have requested an interrupt after a small interval of time. When the timer interrupt occurs, it gets control through the event handling routine of the

kernel for timer interrupts, and decides to preempt h_1 . So it saves the CPU state in h_1 's TCB, and 'schedules' h_2 . Hence the state codes in the TCBs of h_1 and h_2 change to R and N , respectively (Figure 3.14(c)). Note that thread scheduling performed by the thread library is invisible to the kernel. All through these events, the kernel sees process P_i in the *running* state.

A user thread should not make a blocking system call; however, let us see what would happen if h_2 made a system call to initiate an I/O operation on device d_2 , which is a blocking system call. The kernel would change the state of process P_i to *blocked* and note that it is blocked due to an I/O operation on device d_2 (Figure 3.14(d)). Sometime after the I/O operation completes, the kernel would schedule process P_i , and operation of h_2 would resume. Note that the state code in h_2 's TCB remains N , signifying the *running* state, all through its I/O operation!

Advantages and disadvantages of user-level threads Thread synchronization and scheduling is implemented by the thread library. This arrangement avoids the overhead of a system call for communication and synchronization between threads, so the thread switching overhead is smaller than in kernel-level threads. This arrangement also enables each process to use a scheduling policy that best suits its nature. A process implementing a real time application may use priority-based scheduling of its threads to meet its response requirements, whereas a process implementing a multi-threaded server may perform round-robin scheduling of its threads.

Managing threads without involving the kernel also has a few drawbacks. First, the kernel does not know the distinction between a thread and a process, so if a thread were to block in a system call, the kernel would block its parent process. In effect, *all* threads of the process would get blocked until the cause of the blocking was removed—In Figure 3.14 (d) of Example 3.6, thread h_1 cannot be scheduled even though it is in the *ready* state because thread h_2 made a blocking system call. Hence threads must not make system calls that can lead to blocking. To facilitate this, an OS would have to make available a nonblocking version of each system call that would otherwise lead to blocking of a process. Second, since the kernel schedules a process and the thread library schedules the threads within a process, at most one thread of a process can be in operation at any time. Thus, user-level threads cannot provide parallelism (see Section 3.2.2), and the concurrency provided by them is seriously impaired if a thread makes a system call that leads to blocking.

3.4.3 Hybrid Thread Models

A hybrid thread model has *both* user-level threads and kernel-level threads and a method of associating user-level threads with kernel-level threads. Different methods of associating user- and kernel-level threads provide different combinations of the low switching overhead of user-level threads and the high concurrency and parallelism of kernel-level threads. In the following, we describe three such methods and their properties.

Figure 3.15 illustrates methods of associating user-level threads with kernel-level

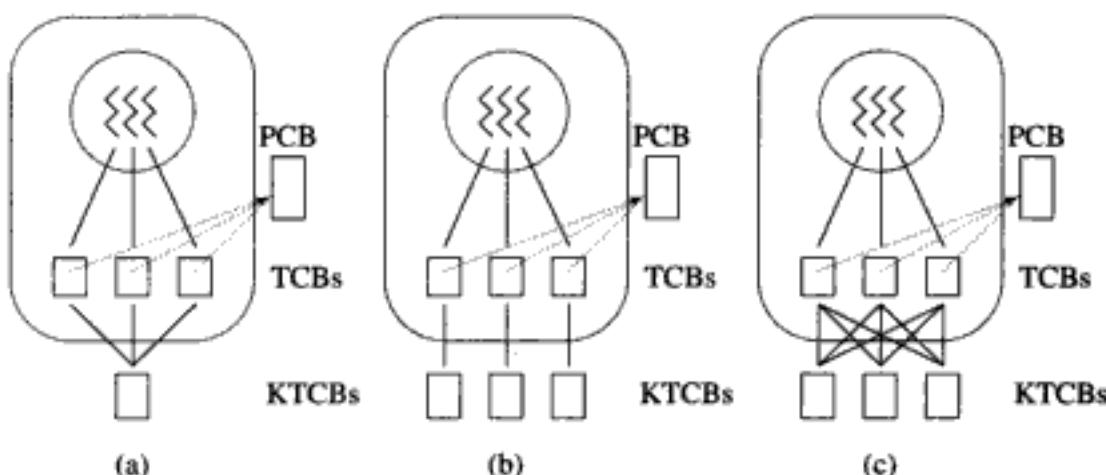


Fig. 3.15 (a) Many-to-one, (b) One-to-one, (c) Many-to-many associations in hybrid threads

threads. The thread library creates user-level threads in a process and associates a *thread control block* (TCB) with each user-level thread. The kernel creates kernel-level threads in a process and associates a *kernel thread control block* (KTCB) with each kernel-level thread. In the many-to-one association method, a single kernel level thread is created in each process by the kernel and all user-level threads created in a process by the thread library are associated with this kernel-level thread. This method of association provides an effect similar to mere user-level threads: User-level threads can be concurrent without being parallel, thread switching incurs low overhead, and blocking of a user-level thread leads to blocking of all threads in the process.

In the one-to-one method of association, each user-level thread is permanently mapped into a kernel-level thread. This association provides an effect similar to mere kernel-level threads: Threads can operate in parallel on different CPUs of a multi-processor system, however switching between threads is performed at the kernel level and incurs high overhead. Blocking of a user-level thread does not block other user-level threads of the process because they are mapped into different kernel-level threads.

The many-to-many association method produces an effect in which a user-level thread may be mapped into any kernel-level thread. Thus, it is possible to achieve parallelism between user-level threads by mapping them into different kernel-level threads, yet perform switching between user-level threads mapped into the same kernel-level thread without incurring high overhead. Also, blocking of a user-level thread does not block other user-level threads of the process that are mapped into a different kernel-level thread. This method requires a complex implementation. We discuss its details when we discuss the hybrid thread model used in the Sun Solaris operating system in Section 3.5.2.

3.5 CASE STUDIES OF PROCESSES AND THREADS

3.5.1 Processes in Unix

Data structures Unix uses two data structures to hold control data about processes:

- *u area* (stands for ‘user area’)
- *proc* structure.

These data structures together hold information analogous to the PCB data structure discussed in Section 3.3. The *proc* structure mainly holds scheduling related data while *u area* contains data related to resource allocation and signal handling. The *proc* structure of a process is always held in memory. *u area* needs to be in memory only when the process is executing. Table 3.10 describes important fields of the *proc* structure and the *u area*.

Table 3.10 Fields of proc structure and u area of Unix

proc structure	<ul style="list-style-type: none"> • Process id • Process state • Priority • Pointers to other <i>proc</i> structures • Signal handling mask • Memory management information
u area	<ul style="list-style-type: none"> • Process control block : For a blocked process this field stores the CPU state • Pointer to <i>proc</i> structure • User and group id's • Information concerning signal handlers for the process • Information concerning all open files and the current directory • Terminal attached to the process, if any • CPU usage information.

Types of processes Three types of processes exist in Unix—user processes, daemon processes and kernel processes.

User processes execute user computations. A user process is associated with the user’s terminal. When a user initiates a program, the kernel creates a main process, which in turn may create other processes. Thus execution of a program leads to a tree of user processes. These processes have a parent-child relationship between them, and they can coordinate their activities with respect to one another. In that sense, they execute in a tightly integrated manner and their lifetimes are related.

Daemon processes perform functions on a system-wide basis. The functions can

be of an auxiliary kind, but they are vital in controlling the computational environment of the system. Typical examples of these functions are print spooling and network management. Once created, daemon processes can exist throughout the lifetime of the OS.

Kernel processes execute code of the kernel. They are concerned with background activities of the kernel like swapping. They differ from daemon processes in two respects: They are created automatically when the system is booted and they can invoke kernel functionalities or refer to kernel data structures without having to perform a system call.

Process creation and termination Unix permits a user process to create child processes and to synchronize its activities with respect to its child processes. These features are analogous to those discussed in Section 3.3.6. We describe some relevant details in the following.

The system call *fork* creates a child process and sets up its environment (called the *user-level context* in Unix literature). It allocates a *proc* structure for the newly created process and marks its state as *ready*, and also allocates a *u area* for the process. *fork* returns the id of the child process to its creator, i.e., to its parent process. The child and parent processes have identical environments, hence data and files can be directly shared. A child process can create its own child processes, thus leading to a tree of processes. The kernel keeps track of the parent-child relationships using the *proc* structure.

The user-level context of the child process is a copy of the parent's user-level context. Hence the child process executes the same code as the parent. At creation, the program counter of the child process is set to contain the address of the instruction at which the *fork* call returns. The program counter of the parent process also contains the same address when the *fork* call returns. The only difference between the parent and child processes is that in the parent process *fork* returns the process id of the child process, whereas the child process returns a 0.

A child process can execute the same program as its parent, or it can execute some other program using the *exec* family of system calls. An *exec* call takes the name of a file as a parameter. This file is expected to contain the program that is to be executed by the process. The kernel loads the code of the program in the environment of the process that performed the *exec* call. Although this arrangement is cumbersome, it has the advantage of flexibility as a child process has the option of executing the parent's code in the parent's environment or choosing its own program for execution. The former alternative was used in older Unix systems to set up servers that could service many user requests concurrently.

The complete view of process creation and termination in Unix is as follows: After booting, the system creates a process *init*. This process creates a child process for every terminal connected to the system. After a sequence of *exec* calls, each child process starts running the login shell. When a programmer indicates the name of a

file from the command line, the shell forms a new process that executes an `exec` call for the named file, in effect becoming the main process of the program. Thus the main process is a child of the shell process. The shell process now executes `await` system call to wait for end of the main process of the program. Thus it becomes blocked until the program completes, and becomes active again to accept the next user command. If a shell process performs an `exit` call to terminate itself, `init` creates a new process for the terminal to run the login shell.

A process P_i can terminate itself through the `exit` system call

```
exit (status_code);
```

where `status_code` is a code indicating the termination status of the process. On receiving the `exit` call the kernel closes all open files of the process, releases the memory allocated to it and destroys the *u area* of the process. However, it does not destroy the *proc* structure; the *proc* structure is retained until the parent of P_i destroys it. The status code is saved in the *proc* structure of P_i . This way the parent of P_i can query its termination status at any time. In essence the terminated process is dead but it exists. It is called a *zombie* process.

The `exit` call also sends a signal to the parent of P_i , which may be ignored by the parent (more about this later). The child processes of P_i are made children of the kernel process `init`. This way `init` receives a signal when a child of P_i , say P_c , terminates so that it can release P_c 's *proc* structure.

Waiting for process termination A process P_i can wait for the termination of a child process through the system call

```
wait (addr(...));
```

where `addr(...)` is the address of a variable within the address space of P_i . If process P_i has child processes and at least one of them has already terminated, the `wait` call stores the termination status of a terminated child process in `xyz` and immediately returns with the id of the terminated child process. If more terminated child processes exist their termination status would be made available to P_i only when it repeats the `wait` call. Process P_i can now take an appropriate action. The state of process P_i is changed to *blocked* if it has children but none of them has terminated. It will be unblocked when one of the child processes terminates. The `wait` call returns with a '1' if P_i has no children. Example 3.7 illustrates advantages of these semantics of the `wait` call.

Example 3.7 Figure 3.16 shows a process that creates three child processes in the `for` loop and awaits their completion. Note that the `fork` call returns the id of the newly created child process in the calling process, but it returns a 0 in the child process. Due to this peculiarity, child processes execute the code in the `if` statement while the parent process skips the `if` statement and executes a `wait` in the `while` loop. The `wait` is satisfied whenever a child process terminates by executing the `exit` statement.

However, the parent process wishes to wait until the last process finishes, so it issues another `wait` if the value returned is something other than `-1`. The fourth `wait` call returns with a `-1`, which brings the parent process out of the loop. The parent process code does not contain an explicit `exit()` call. The language compiler would automatically add this at the end of `main()`.

```
main()
{
    int saved_status;
    for (i = 0; i < 3; i++)
    {
        if (fork() == 0)
        { /* code for child processes */
            ...
            exit();
        }
    }
    while(wait(&saved_status) != -1);
        /* loop till all child processes terminate */
}
```

Fig. 3.16 Process creation and termination in Unix

Interrupt processing Unix uses the notion of interrupt priority level (*ipl*) to avoid interrupts during sensitive kernel-level actions. This approach helps to eliminate inconsistency of the kernel data structures as follows: Each interrupt is assigned an interrupt priority level. Depending on the program being executed by the CPU, an interrupt priority level is also associated with the CPU. When an interrupt at a priority level *l* arises, it is handled only if *l* is larger than the interrupt priority level of the CPU; otherwise, it is kept pending until the CPU's interrupt priority level becomes $< l$. The kernel raises the *ipl* of the CPU to a high value before starting to update its data structures and lowers it only after the update is completed. The action of changing the value of *ipl* does constitute overhead, however this price has to be paid for consistency of kernel data structures.

System calls Each system call accepts parameters relevant to its functioning. When a call occurs, these parameters exist on the user stack of the process that issues the system call. The system call number is expected to be in CPU register 0 when the call is made. The system call handler obtains this number to determine which system functionality is being invoked. From its internal tables it knows the address of the handler for this functionality. It also knows the number of parameters this call is supposed to take. However, these parameters exist on the user stack, which is a part of the process context of the process making the call. So before passing control to the handler for the specific call these parameters are copied from the process stack into some standard place in the *u area* of the process. This simplifies operation of

individual event handlers.

Signals The concept of a signal is described in detail in Section 3.6.4. In Unix, a process can ignore a signal, specify a signal handler to perform actions of its choice when a signal is sent to it, or let the OS take a default action like producing a core dump or terminating the process. We begin by discussing how a Unix process performs the following actions:

- Send a signal to another process
- Specify a signal handler for a signal.

A signal can be sent to a process, or to a group of processes within the same group as the sender process. This is performed by the *kill* system call

kill (<pid>, <signum>)

where *<pid>* is an integer value that can be positive, zero or negative.

A positive value of *<pid>* is the id of a process to which the signal is to be sent. A 0 value of *<pid>* implies that the signal is to be sent to some processes within the same process tree as the sender process, i.e., some processes that share an ancestor with the sender process. This feature is implemented as follows: At a *fork* call, the newly created process is assigned a group id that is the same as the process group number of its parent process. A process may change its group number using the *setpgid* system call. Thus, at any moment, some processes in the same process tree as the sender of the signal may have the same group number. When *<pid>* = 0, a signal is sent to all these processes.

A signal sent with *<pid>* = -1 is intended to reach processes outside the process tree of the sender. This feature can be used to broadcast a signal to all processes in the system. We will not elaborate on this feature here.

A process specifies a signal handler by executing the statement

oldfunction = signal (<signum>, <function>)

where *signal* is a function in the C library that makes a *signal* system call, *<signum>* is an integer and *<function>* is the name of a function within the address space of the process. This call specifies that the function *<function>* should be executed on occurrence of the signal *<signum>*. The *signal* call returns with the previous action specified for the signal *<signum>*. A user can specify *SIG_DFL* as *<function>* to indicate that the default action defined in the kernel is to be executed on occurrence of the signal, or specify *SIG_IGN* as *<function>* to indicate that the occurrence of the signal is to be ignored.

The kernel uses the *u area* of a process to note the signal handling actions specified by it, and a set of bits in the *proc* structure to register the occurrence of signals. Whenever a signal is sent to a process, the bit corresponding to the signal is set to 1 in the *proc* structure of the destination process. The kernel now determines whether

the signal is being ignored by the destination process. If not, it makes provision to deliver the signal to the process. An ignored signal remains pending and is delivered when the process specifies its interest in receiving the signal (by specifying an action or by specifying that the default action should be used for it). In Unix, a signal remains pending if the process for which it is intended is in a *blocked* state. The signal would be delivered when the process comes out of the blocked state. In general, the kernel checks for pending signals at the following times:

- When a process returns from a system call or interrupt
- After a process gets unblocked
- Before a process gets blocked on an event.

Invocation of the signal processing action is implemented in an interesting manner. Knowing the signal that is to be delivered to a process, the kernel determines the address of the corresponding signal handling action specified by the process. It now changes the user stack of the process and the CPU state associated with the process to make it look as if the process had actually invoked the signal handling action as a function. These changes involve pushing the value in the program counter (PC) of the process state on the user stack and making it look like a return address for a function call, and putting the address of the signal handling action into the program counter component of the process state. The kernel also makes provision such that on return from this function call, the CPU would be put into the state in which it existed when the signal occurred. After making these changes, the kernel returns control to the user process.

Due to changes in the user stack and the CPU state, the process would now execute the signal handling action. If the process is not in execution when the signal occurred, these actions would take place the next time it is scheduled for execution. At the end of signal processing, the process should return to the point at which it was interrupted to deliver the signal. However, as explained in the following, it does not always do so.

A few anomalies exist in the way signals are handled. If a signal occurs repeatedly, the kernel simply notes that it has occurred, but does not count the number of its occurrences. Consequently, the signal handler may be executed once or several times depending on when the process gets scheduled to execute the signal handler. Another anomaly concerns a signal sent to a process that is blocked in a system call. After executing the signal handler, such a process does not resume its execution of the system call. Instead, it returns from the system call. If necessary, it would have to repeat the system call.

Table 3.11 lists some interesting Unix signals. Note that signals can arise asynchronously from sources outside a process, or can arise synchronously as a result of the execution of the process itself. Signals like segmentation fault, illegal instruction, and illegal system call in Table 3.11 belong to the latter category.

Table 3.11 Interesting signals in Unix

Signal	Description
SIGCHLD	child process died or suspended
SIGFPE	arithmetic fault
SIGILL	illegal instruction
SIGINT	tty interrupt (Control-C)
SIGKILL	kill process
SIGSEGV	segmentation fault
SIGSYS	invalid system call
SIGXCPU	CPU time limit is exceeded
SIGXFSZ	File size limit is exceeded

Process states and state transitions There is one conceptual difference between the process model described in Section 3.3.3 and that used in Unix. In the model of Section 3.3.3, a process in the *running* state is put in the *ready* state the moment its execution is interrupted. A system process then handles the event that caused the interrupt. If the running process had itself caused a software interrupt by executing an $\langle SI_{instrn} \rangle$, its state may further change to *blocked* if its request cannot be granted immediately. In this model a user process executes only user code; it does not need any special privileges. A system process is used to handle system calls because privileged instructions like I/O initiation and setting of memory protection information may have to be executed. Hence the system process executes with the CPU in the privileged mode.

Processes behave differently in the Unix model. When a process makes a system call, the process itself proceeds to execute the kernel code meant to handle the system call. To ensure that it has the necessary privileges, it needs to execute in a special mode called the *kernel mode*. While executing in this mode, the CPU is in the privileged mode of operation. A mode change is thus necessary every time a system call is made. The opposite mode change occurs after processing a system call. Similar mode changes occur when a process starts executing the interrupt processing code in the kernel due to an interrupt, and when it returns after processing an interrupt.

A process may get blocked while executing kernel code. This can happen if it makes a system call to initiate an I/O operation, or if it makes a request that cannot be granted immediately. The Unix kernel uses *reentrant code* to handle such situations. Reentrancy implies that many processes should be able to execute the same code concurrently. If this were not the case, a process blocking itself in the kernel mode would freeze the entire system until it gets unblocked and exits from the kernel mode. To ensure reentrancy of code a process executing in the kernel mode must use its own kernel stack. The kernel stack of a process has to be distinct from its user stack because most machine architectures determine which stack to use based on the mode

bit in the PSW. The kernel stack contains the history of function invocations since the time the process entered the kernel mode. If another process also enters the kernel mode, the history of its function invocations would be maintained on its own kernel stack.

Another consequence of the Unix process model is that Unix uses two distinct *running* states. These are called *user running* and *kernel running*. Another difference when compared to the states described in Section 3.3.3 is the *zombie* state discussed earlier. A user process executes user code while in the *user running* state, and kernel code while in the *kernel running* state. It makes the transition from *user running* to *kernel running* when it makes a system call, or when an interrupt occurs. It may get blocked while in the *kernel running* state due to an I/O operation or due to non-availability of a resource. When the I/O operation completes or the resource request is granted, the process returns to the *kernel running* state and completes the execution of the kernel code that it was executing. It now leaves the kernel mode and returns to the user mode. Accordingly, its state is changed from *kernel running* to *user running*.

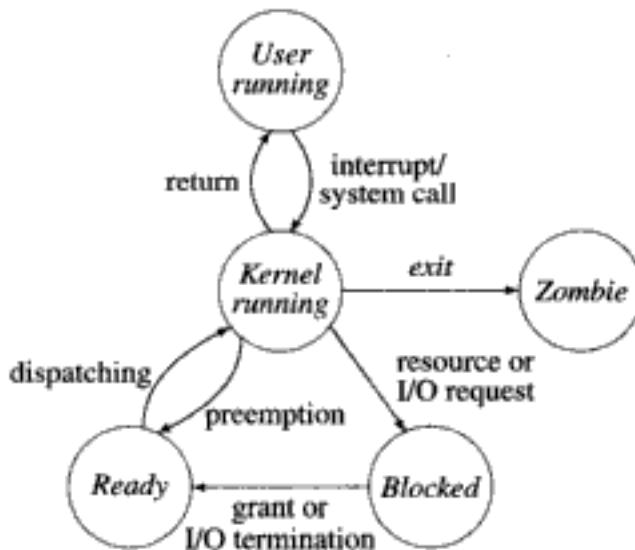


Fig. 3.17 Process state transitions in Unix

Due to this arrangement, a process does not get blocked or preempted in the *user running* state—it first makes a transition to *kernel running* and then gets blocked or preempted. In fact *user running* → *kernel running* is the only transition out of the *user running* state. Figure 3.17 illustrates process states and state transitions in Unix. As shown there, even process termination occurs when a process is in the *kernel running* state. This happens because it executes the system call *exit* while in the *user running* state. This call changes its state to *kernel running*. The process actually terminates and becomes a *zombie* process as a result of processing this call.

Processes in the *kernel running* state are non-interruptible in Unix. The reasoning is as follows: In the conventional process model discussed in Section 3.3.3, interrupt

processing and event handling is performed by the kernel through system processes. We have to make sure that these activities are not interrupted at inconvenient times by further interrupts as this could lead to inconsistency of data structures and race conditions. Operating systems using the conventional model therefore mask all interrupts when the kernel is about to access or update some sensitive data. In the Unix model kernel code is executed by user processes themselves when they are in the *kernel running* state. Therefore, processes in this state are made non-interruptible—the process is not interrupted even when its time slice elapses.

This feature does not cause any degradation of response times because a process executing in the *kernel running* state does not consume much CPU time—it either completes processing of the interrupt or system call that brought it into the *kernel running* state and returns to the *user running* state, or it gets blocked. This blocking is done carefully so that problems of data inconsistency and race conditions do not arise.

3.5.2 Threads in Solaris

Solaris, which is a Unix 5.4 based operating system, uses three kinds of entities to govern concurrency within a process. These are:

- *User threads*: User threads share some properties with user-level threads discussed in Section 3.4.2, however they differ from user-level threads in some other respects. User threads are created and managed by a threads library, and are invisible to the kernel.
- *Light weight processes*: A light weight process (LWP) is an intermediary between user threads and kernel threads. An LWP is a unit of parallelism within a process. User threads are mapped into LWPs by the threads library. This mapping is analogous to the mapping of user-level threads into processes described in Section 3.4.2. More than one LWP may be created for a process. The number of LWPs for a process and the nature of the mapping between user threads and LWPs is decided by the programmer and made known to the threads library through appropriate function calls.
- *Kernel threads*: A kernel thread is a thread created by the kernel for concurrency control. The kernel creates as many kernel threads as the total number of LWPs created for all active processes in the system, and associates a kernel thread with each LWP. The kernel also creates some kernel threads for its own use. A good example of this kind is a thread to handle disk I/O in the system.

Figure 3.18 illustrates an arrangement of user threads, LWPs and kernel threads. Two processes P_i and P_j exist in the system. Process P_i has created three user threads in it, and process P_j has created four user threads. One LWP is created for process P_i while three LWPs are created for process P_j . All three user threads of process P_i are mapped into the same LWP, whereas a complex mapping exists between the four user threads of process P_j and the three LWPs created for it. One user thread is exclusively

mapped into one LWP. The other LWPs are shared between the remaining three user threads.

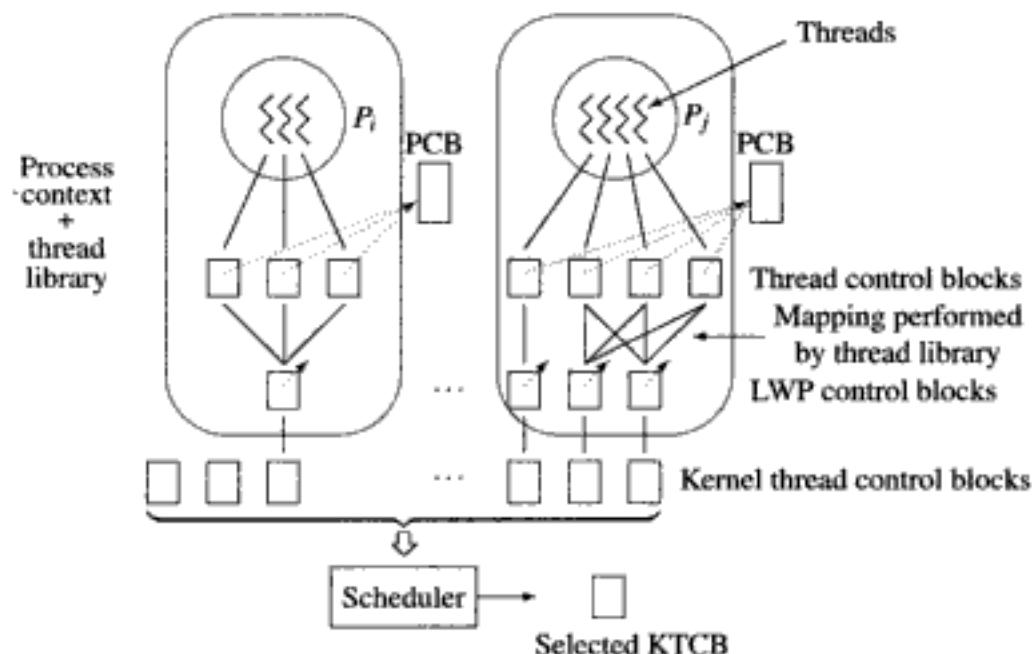


Fig. 3.18 Threads in Solaris

The kernel allocates one LWP control block for each LWP. One kernel thread is created for each LWP and associated exclusively with it. A few more kernel threads are created for purposes described earlier. The kernel allocates a kernel thread control block (KTCB) for each kernel thread. The scheduler selects a KTCB corresponding to a kernel thread in the *ready* state. The dispatcher dispatches the kernel thread corresponding to this KTCB. In effect, the LWP corresponding to this kernel thread is dispatched. The threads library can switch between user threads mapped into this LWP. In a multiprocessor OS, the scheduler selects one KTCB for each CPU in the system.

The Solaris arrangement consisting of user-level threads, LWPs and kernel threads provides concurrency between user-level threads and, in a multiprocessor system, provides parallelism between LWPs. The concurrency is provided by the threads library. Parallelism is provided through LWPs; however, it is under programmer control because the programmer decides how many LWPs should be created for a process. Thus, if a programmer creates two LWPs in a process, a 2-way parallelism would exist in the process.

A programmer can create n LWPs to obtain an n -way concurrency. A value of n in the range $1 \leq n \leq p$, where p is the number of CPUs, makes sense. However, is a value $n > p$ meaningful? The answer depends on the programmer's intentions. $1 \leq n \leq p$ can provide an n -way parallelism, as each LWP may execute on a processor at

the same time. $n > p$ cannot provide an n -way parallelism, however the concurrency provided by it may be quite valuable—if one LWP of a process is blocked (i.e., if all threads mapped into the LWP are blocked), another LWP of the same process may execute on a processor.

Due to the use of LWPs, Solaris provides a simplification over user-level threads. It permits a user thread to make a blocking call. This call would block the LWP with which the thread is associated. However, other user threads may be able to execute in other LWPs. Thus, blocking of a user thread does not necessarily block the process to which it belongs. The design of a program is simplified because no special effort needs to be made to eliminate blocking of user threads.

A complex arrangement of control blocks is used to control switching between kernel threads. The kernel thread control block contains the CPU state and stack pointer, priority, scheduling information, and a pointer to the next KTCB in a scheduling queue. In addition, it contains a pointer to the LWP control block and to the PCB of the process to which the LWP belongs. The LWP control block contains saved values of user registers of the CPU, signal handling information and pointer to the PCB of the owner process.

The signal handling arrangement is rather interesting. All LWPs of a process share common signal handlers. However, each LWP has its own signal mask that determines which signals should be delivered to it and which ones should be ignored. In addition, the threads library maintains a control block for each user thread. The information in this control block is similar to that described in Section 3.4.2.

Solaris performs scheduling at two levels. At the lower level, a threads library ‘schedules’ user threads to execute in an LWP. This scheduling is analogous to scheduling of user-level threads to execute in a process (see Section 3.4.2). LWPs are exclusively associated with kernel threads, hence they are not scheduled explicitly.

At the higher scheduling level, the kernel schedules kernel threads to execute on CPUs. Before dispatching a specific kernel thread, it has to find the process to which the LWP associated with the kernel thread belongs. Process environment of the process has to be made accessible before dispatching the kernel thread. Similarly, while switching between kernel threads, the kernel has to find whether the kernel thread to be dispatched belongs to the same process as the kernel thread that was executing. If not, it has to perform process switching by saving the environment of the process to which the executing kernel thread belonged and loading the environment of the process to which the selected kernel thread belongs.

3.5.3 Processes and Threads in Linux

Data structures The Linux kernel uses a *process descriptor*, which is a data structure of type `task_struct`, to contain all information pertaining to a process or thread. For a process, this data structure contains the process state, information about its parent and child processes, the terminal used by the process, its current directory, open files, the memory allocated to it, signals and signal handlers. The

kernel creates sub-structures to hold information concerning the terminal, directory, files, memory and signals and puts pointers to them in the process descriptor. This organization saves both memory and overhead when a thread is created.

Creation and termination of processes and threads Linux supports the system calls *fork* and *vfork* whose functionalities are identical to the corresponding Unix calls. These functionalities are actually implemented by the system call *clone* which is hidden from the view of programs. Both processes and threads are created this way, the only difference is in the flags passed to the *clone* call as a parameter, which we shall discuss shortly. Linux treats processes and threads alike for scheduling purposes; effectively, the Linux threads are kernel-level threads.

The *clone* system call takes four parameters: start address of the process or thread, parameters to be passed to it, flags, and a child stack specification. Some of the important flags are:

CLONE_VM	:	Shares the memory management information used by the MMU
CLONE_FS	:	Shares the information about root and current working directory
CLONE_FILES	:	Shares the information about open files
CLONE_SIGHAND	:	Shares the information about signals and signal handlers

The organization of *task_struct* facilitates selective sharing of this information since it merely contains pointers to the substructures where the actual information is stored. At a *clone* call, the kernel makes a copy of *task_struct* in which some of these pointers are copied and others are changed. A thread is created by calling *clone* with all these flags set, so that the new thread shares the address space, files and signal handlers of its parent. A process is created by calling *clone* with all these flags cleared; the new process does not share any of these components.

The Linux 2.6 kernel continues to support the 1:1 threading model (i.e., kernel-level threads) and the above method of creating processes and threads. It also includes support for the Native Posix Threading Library (NPTL), which provides a number of enhancements that benefit heavily threaded applications. It can support upto 2 billion threads whereas the Linux 2.4 kernel could support only upto 8,192 threads per CPU. A new system call *exit_group()* has been introduced to terminate a process and all its threads; it can terminate a process having a hundred thousand threads in about 2 seconds, as against about 15 minutes in the Linux 2.4 kernel. Signal handling is performed in the kernel space, and a signal is delivered to one of the available threads in a process. Stop and continue signals affect an entire process, while fatal signals terminate the entire process. These features simplify handling of multi-threaded processes. The Linux 2.6 kernel also supports a fast user-space mutex called *futex* that reduces the overhead of thread synchronization through a reduction in the number of system calls.

Parent-child relationships Information about parent and child processes or threads is stored in a `task_struct` to maintain awareness of the process tree. `task_struct` contains a pointer to the parent and to the deemed parent, which is a process to whom termination of this process should be reported if its parent process has terminated, a pointer to the youngest child, and pointers to the younger and older siblings of a process. Thus, the process tree of Figure 3.2 would be represented as shown in Figure 3.19.

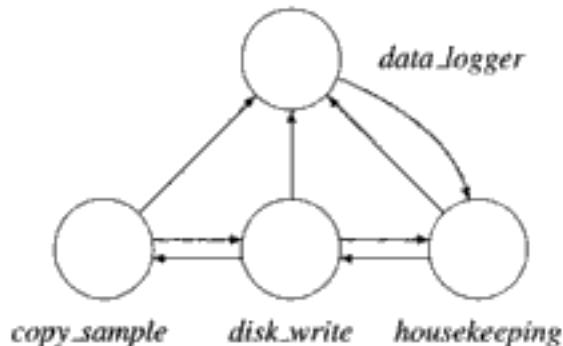


Fig. 3.19 Linux process tree for the processes of Figure 3.2(a)

Process states The `state` field of a process descriptor contains a flag indicating the state of a process. A process can be in one of five states at any time:

- | | |
|----------------------|---|
| TASK_RUNNING | : The process is either scheduled or waiting to be scheduled. |
| TASK_INTERRUPTIBLE | : The process is sleeping on an event, but may receive a signal. |
| TASK_UNINTERRUPTIBLE | : The process is sleeping on an event, but may not receive a signal. |
| TASK_STOPPED | : The operation of a process has been stopped by a signal. |
| TASK_ZOMBIE | : The process has completed, but the parent process has not yet issued a system call of the <code>wait</code> -family to check whether it has terminated. |

The `TASK_RUNNING` state corresponds to one of *running* or *ready* states described in Section 3.3.3. The `TASK_INTERRUPTIBLE` and `TASK_UNINTERRUPTIBLE` states both correspond to the *blocked* state. Splitting the *blocked* state into two states resolves the dilemma faced by an OS in handling signals sent to a process in the *blocked* state (see Section 3.6.4)—a process can decide whether it wants to be activated by a signal while waiting for an event to occur, or whether it wants the delivery of a signal to be deferred until it comes out of the *blocked* state. A process enters the `TASK_STOPPED` state when it receives a `SIGSTOP` or `SIGTSTP` signal to indicate

that its execution should be stopped, or a SIGTTIN or SIGTTOU signal to indicate that a background process requires input or output.

3.5.4 Processes and Threads in Windows

The flavor of processes and threads in Windows differs somewhat from that presented earlier in this chapter—Windows treats a process as a unit for resource allocation, and uses a thread as a unit for concurrency. Accordingly, a Windows process does not operate by itself; it must have at least one thread inside it. The OS maintains a handle table for each process in which it stores handles to resources. A process inherits some resource handles from its parent and may open new resources when needed; the OS adds their handles to the handle table when an open succeeds.

Windows uses three control blocks to manage a process. Each process has an *executive process block*. It contains fields that store the process id, memory management information, address of the handle table, a *kernel process block* for the process and a *process environment block*. The kernel process block contains scheduling information for threads of the process, such as the processor affinity for the process, the state of the process and pointers to the kernel thread blocks of its threads. The executive process block and the kernel process block are situated in the system address space. The process environment block is situated in the user address space because it contains information that is used by the loader to load the code to be executed, and by the heap manager.

The control blocks employed to manage a thread contain information about its operation, and about the process containing it. The *executive thread block* contains a *kernel thread block*, a pointer to the executive process block of the process that contains the thread and impersonation information for the thread. The kernel thread block contains information about the kernel stack of the thread and the thread-local storage, scheduling information for the thread, and a pointer to its *thread environment block*, which contains its id and information about its synchronization requirements.

Windows supports the notion of a *job* as a method of managing a group of processes. A job is represented by a job object, which contains information such as handles to processes in it, the job-wide CPU time limit, per process CPU time limit, job scheduling class which sets the time slice for the processes of the job, processor affinity for processes of the job, and their priority class. A process can be a member of only one job; all processes created by it automatically belong to the same job.

Process creation The Windows executive provides a general model of process and thread creation in which the process that issues a *create* call for a process or thread is not necessarily its parent—the *create* call takes a handle to the parent of the new process or thread as a parameter. This service is used by a server process to perform *impersonation*—it creates a thread in a client process so that it can access resources with the client's privileges.

The semantics of process creation depend on the environment subsystem used by

an application process. In the Win/32 and OS/2 operating environments, a process has one thread in it when it is created; it is not so in other environments supported by the Windows OS. Hence process creation is actually handled by an environment subsystem DLL that is linked to an application process. After creation, it passes the id of the new process or thread to the environment subsystem process so that it can manage the new process or thread appropriately.

Creation of a child process by an application process in the Win/32 environment proceeds as follows: The environment subsystem DLL linked to the application process makes a system call to create a new process. This call is handled by the executive. It creates a process object, initializes it by loading the image of the code to be executed, and returns a handle to the process object. The environment subsystem DLL makes a second system call to create a thread, and passes the handle to the new process as a parameter. The executive creates a thread in the new process and returns its handle. The DLL now sends a message to the environment subsystem process, passing it the process and thread handles, and the id of their parent process. The environment subsystem process enters the process handle in the table of processes that currently exist in the environment and enters the thread handle in the scheduling data structures. Control now returns to the application process.

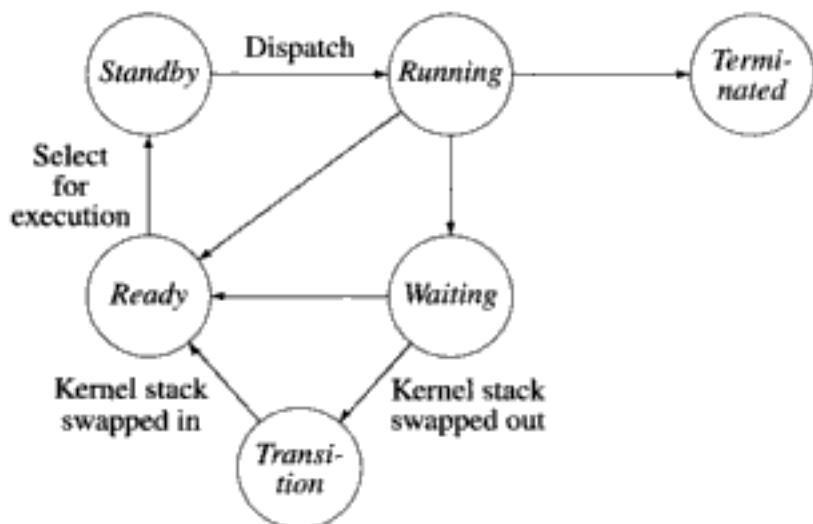


Fig. 3.20 Thread state transitions in Windows

Thread states and state transitions A thread can be in one of following six states:

1. *Ready*: The thread can be executed if a CPU is available.
2. *Standby*: This is a thread that has been selected to run next on a specific processor. If its priority is high, the thread currently running on the processor would be preempted and this thread would be scheduled.
3. *Running*: A CPU is currently allocated to the thread and the thread is in operation.

4. *Waiting*: The thread is waiting for a resource or event, or has been suspended by the environment subsystem.
5. *Transition*: The kernel stack of the thread has been removed from memory because the thread has been waiting for long. It enters the *ready* state when the kernel stack is brought back into memory.
6. *Terminated*: The thread has completed its operation.

Figure 3.20 shows the state transition diagram for threads. Causes of transitions only into and out of the *standby* and *transition* states are shown here. Causes of other transitions are as discussed in Section 3.3.3.

3.6 INTERACTING PROCESSES—AN ADVANCED PROGRAMMER VIEW OF PROCESSES

An application may use multiple processes to obtain the benefits of computation speed-up and improved response times (see Table 3.2). Another reason could be that the real world system serviced by the application may have several activities in progress at the same time. Example 3.8 illustrates this aspect.

Example 3.8 An airline reservations application uses a number of agent terminals connected to a central computer system. The computer stores the data in a centralized fashion and permits agents at all agent terminals to access this data in real time. Execution of the application may be structured in the form of multiple processes, each process servicing one agent terminal (see Figure 3.21). This arrangement permits an agent at one terminal to key in the requirements of a customer while other agents access and update the data on behalf of other customers.



Fig. 3.21 An airline reservation system

The processes of an application interact with one another to coordinate their activities or to share some data. The nature of these interactions is discussed in the following sections. We begin by defining the term *interacting processes* using the following notation:

$read_set_i$: set of data items read by process P_i
 $write_set_i$: set of data items modified by process P_i

Here data includes program data as well as interprocess messages and signals discussed in later Sections. (A message or a signal sent by process P_i is included in $write_set_i$, and a message or a signal received by it is included in $read_set_i$.)

Definition 3.4 (Interacting processes) Processes P_i and P_j are interacting processes if $\text{read_set}_i \cap \text{write_set}_j \neq \emptyset$ or $\text{read_set}_j \cap \text{write_set}_i \neq \emptyset$.

Processes that do not interact are said to be *independent processes*.

Implementing process interactions Different kinds of process interactions were summarized in Table 3.3. These interactions are realized by coordinating the activities of processes with respect to one another. A common theme in all kinds of process interactions is that a process makes system calls to indicate that it should be blocked and activated under certain conditions. The kernel implements blocking or activation of processes through appropriate state changes. We describe details of process interactions in the next Section. Chapter 9 describes techniques used in the design of interacting processes.

3.6.1 Race Conditions and Data Access Synchronization

An application may consist of a set of processes sharing some data d_s . Data access synchronization involves blocking and activation of these processes such that they correctly share d_s . The need for data access synchronization arises because accesses to shared data in an arbitrary manner may lead to wrong results in the processes and may also affect consistency of data. We begin by discussing these problems.

Let processes P_i and P_j perform operations a_i and a_j on d_s . Let a_i be an update operation that increments the value of d_s by 10, viz.

$$a_i : d_s := d_s + 10;$$

This operation may be implemented using three machine instructions. The first instruction loads the value of d_s in a data register, the second instruction adds 10 to the contents of the data register and the third instruction stores the contents of the data register back into the location assigned to d_s . We call this sequence of instructions the *load-add-store* sequence.

Let operation a_j be a simple copy operation that copies the value of d_s into another location. If a_j is executed before the first instruction of a_i , it will obtain the value of d_s before the commencement of a_i , i.e., the 'old' value of d_s . If a_j is executed after the last instruction of a_i , it will obtain the new value of d_s . However, one cannot predict whether a_j will obtain the old or the new value if it is executed while process P_j is engaged in executing a_i . By itself, this is not harmful; however, it may lead to situations of the following kind: Process P_j starts to take a certain action because d_s has a certain value, however the value of d_s is different by the time P_j completes the action! This situation may be considered inconsistent in certain applications.

A harmful situation called *race condition* may arise during execution of concurrent processes. (This term is used in electronics when attempts to examine a value while it is changing, or make measurements on a changing waveform, leads to wrong results.) We use the following notation to discuss race conditions. Let a_i and a_j be

operations on shared data d_s performed by two interacting processes P_i and P_j . Let $f_i(d_s)$, $f_j(d_s)$ represent the values of d_s after performing operations a_i , a_j , respectively.

Definition 3.5 (Race condition) A race condition on a shared data item d_s is a situation in which the value of d_s resulting from execution of two operations a_i and a_j may be different from both $f_i(f_j(d_s))$ and $f_j(f_i(d_s))$.

Let a_i, a_j be the update operations

$$\begin{aligned} a_i: \quad d_s &:= d_s + 10; \\ a_j: \quad d_s &:= d_s + 5; \end{aligned}$$

If processes P_i and P_j perform operations a_i and a_j , respectively, one would expect 15 to be added to the value of d_s . A race condition arises if this is not the case! Let us see how this could happen.

The result of performing a_i and a_j would be correct if one of them operates on the value resulting from the other operation, but would be wrong if both a_i and a_j operate on the old value of d_s . This could happen if one process is engaged in performing the load-add-store sequence, but the other process performs a load instruction before this sequence is completed. Example 3.9 illustrates race conditions in an airline reservations application and its consequences.

Example 3.9 Processes of the airline reservations system of Example 3.8 execute identical code shown in the left column of Figure 3.22. Processes share the variables *nextseatno* and *capacity*. Each process examines the value of *nextseatno* and updates it by 1 if a seat is available. Thus a_i and a_j are identical operations. The right column of Figure 3.22 shows the machine instructions corresponding to the code. Statement S_3 corresponds to 3 instructions $S_{3.1}$, $S_{3.2}$ and $S_{3.3}$ that form a load-add-store sequence of instructions.

Figure 3.23 shows three executions of processes P_i and P_j when $nextseatno = 200$ and $capacity = 200$. In case 1, process P_i executes the if statement that compares values of *nextseatno* with *capacity* and proceeds to execute statements S_2 and S_3 that allocate a seat to it and increment *nextseatno*. When process P_j executes the if statement, it finds that no seats are available, so it does not perform any seat allocation.

In case 2, process P_i executes the if statement and finds that a seat can be allocated. However, P_i gets preempted before it can perform allocation. Process P_j now executes the if statement and finds that a seat is available. It allocates a seat and exits. *nextseatno* is now 201. However, when process P_i is resumed, it proceeds to execute instruction $S_{2.1}$ because it had ascertained availability of a seat before it was preempted. Thus, it allocates seat numbered 201 even though only 200 seats exist! This is a wrong result as execution of the seat allocation logic when $nextseatno = 201$ should not lead to allocation of a seat.

In case 3, process P_i gets preempted after it loads 200 in *reg_j*. Now, both P_i and P_j allocate a seat each however *nextseatno* is incremented by only 1! Thus, cases 2 and 3 involve race conditions.

S_1	if $nextseatno \leq capacity$	$S_{1.1}$	Load $nextseatno$ in reg_k
	then	$S_{1.2}$	If $reg_k > capacity$ goto $S_4.1$
S_2	$allotedno := nextseatno;$	$S_{2.1}$	Move $nextseatno$ to $allotedno$
S_3	$nextseatno := nextseatno + 1;$	$S_{3.1}$	Load $nextseatno$ in reg_j
		$S_{3.2}$	Add 1 to reg_j
		$S_{3.3}$	Store reg_j in $nextseatno$
		$S_{3.4}$	Go to $S_5.1$
	else		
S_4	<i>display "sorry, no seats available"</i>	$S_{4.1}$	Display "sorry, ..."
S_5	<i>...</i>	$S_{5.1}$	<i>...</i>
	<i>Program</i>		<i>Machine instructions</i>

Fig. 3.22 Data sharing by processes of a reservations system

Existence of race conditions in a program leads to a practical difficulty. Such a program is incorrect, however it does not always produce incorrect results because its behavior depends on the order in which instructions of different processes are executed. Consequently, a program that produced incorrect results for some data may produce correct results if its execution is repeated on the same data! This feature complicates testing and debugging of programs containing concurrent processes. Hence the best way to handle race conditions is to prevent them from arising.

Preventing race conditions Race conditions would not arise if we ensure that operations a_i and a_j of Defn. 3.5 do not execute concurrently—that is, a_j will not be in execution if a_i is in execution, and vice versa. This requirement is called *mutual exclusion*. It is satisfied by permitting only one operation to access shared data d_s at any time. When mutual exclusion is used, we can be sure that the result of executing operations a_i and a_j would be either $f_i(f_j(d_s))$ or $f_j(f_i(d_s))$.

Data access synchronization is a technique user for implementing mutual exclusion over shared data. It delays a process wishing to access d_s if another process is accessing it. A set of processes $\{P_i\}$ is said to require data access synchronization if race conditions arise during their execution. We identify this set of processes by following Def. 3.6 for each pair of processes. We use the following notation for this purpose:

$update_set_i$: set of data items updated by process P_i , that is, the set of data items whose values are read, modified and written back by process P_i

Definition 3.6 (Processes containing race conditions) A race condition exists in processes P_i and P_j of an application if $update_set_i \cap update_set_j \neq \emptyset$.

Time instant	Actions of P_i		Actions of P_j		Actions of P_l	
	P_i	P_j	P_i	P_j	P_l	P_j
1	$S_1.1$	—	$S_1.1$	—	$S_1.1$	—
2	$S_1.2$	—	$S_1.2$	—	$S_1.2$	—
3	$S_2.1$	—	—	$S_1.1$	$S_2.1$	—
4	$S_3.1$	—	—	$S_1.2$	$S_3.1$	—
5	$S_3.2$	—	—	$S_2.1$	—	$S_1.1$
6	$S_3.3$	—	—	$S_3.1$	—	$S_1.2$
7	$S_3.4$	—	—	$S_3.2$	—	$S_2.1$
8	—	$S_1.1$	—	$S_3.3$	—	$S_3.1$
9	—	$S_1.2$	—	$S_3.4$	—	$S_3.2$
10	—	$S_4.1$	$S_2.1$	—	—	$S_3.3$
11	—	—	$S_3.1$	—	—	$S_3.4$
12	—	—	$S_3.2$	—	$S_3.2$	—
13	—	—	$S_3.3$	—	$S_3.3$	—
14	—	—	$S_3.4$	—	$S_3.4$	—

case 1 case 2 case 3

Fig. 3.23 Race conditions in the airline reservations system

The following method can be used to prevent race conditions in an application program:

- For every pair of processes P_i and P_j that share some data, check whether a race condition exists.
- If so, ensure that processes P_i and P_j access shared data in a mutually exclusive manner.

Data access synchronization is discussed in Chapter 9.

3.6.2 Control Synchronization

In control synchronization, interacting processes coordinate their execution with respect to one another. Control synchronization may be required at any point in the lifetime of a process, including at the start or end of its lifetime.

Definition 3.7 (Control synchronization) Control synchronization between a pair of processes P_i and P_j implies that execution of some instruction (statement) s_i in process P_j , and the instructions (statements) following it in the order of execution, are delayed until process P_i executes an instruction (statement) s_i .

Figure 3.24 illustrates different situations in control synchronization. Each part of the figure shows execution of processes P_i and P_j . Two assumptions are made in drawing this figure. First, it is assumed that the time axis extends vertically downwards in the figure, i.e., execution of a statement shown at a higher level in a process P_i or P_j occurs earlier than execution of a statement shown at a lower level in the same or another process. Second, execution of a statement in process P_i or P_j is shown at the earliest possible moment of time with respect to the execution of other statements in P_i and P_j .

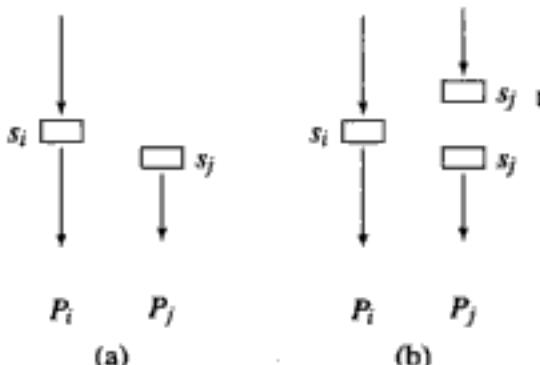


Fig. 3.24 Control synchronization between processes P_i and P_j

Figure 3.24(a) shows that statement s_j is the first statement of process P_j . Execution of this statement cannot take place until process P_i executes statement s_i . Thus, synchronization occurs at the start of process P_j . Part (b) shows synchronization occurring in the middle of process P_j because statement s_j of process P_j cannot be executed until process P_i executes statement s_i . Note that process P_j finished executing statement s_{j-1} much earlier, however execution of s_j was delayed until P_i executed s_i .

Example 3.10 illustrates the need for control synchronization when an application is structured into a set of interacting processes to improve its response time.

Example 3.10 A program is to be designed to reduce the elapsed time of a computation that consists of the following actions:

- Compute $Y = \text{HCF}(A_{\max}, X)$, where array A contains n elements and A_{\max} is the maximum value in array A .
- Insert Y in array A .
- Arrange A in ascending order.

The problem can be split into the following steps:

1. Read n elements of array A .
2. Find the maximum magnitude A_{\max} .
3. Read X .
4. Compute $Y = \text{HCF}(A_{\max}, X)$.
5. Include Y in array A and arrange the elements of A in ascending order.

To decide which of these statements can be performed concurrently with one another, execution of each statement is considered to be a separate process and Def. 3.7 is applied to find which of these processes interact. Processes for statements 1 and 3 can be performed concurrently. Processes of statements 2, 4 and 5 are interacting processes. They cannot be executed concurrently because they share array A and at least one of them modifies array A. However, concurrency can be achieved by splitting steps 2 and 5 into two parts each as

- 2(a). Copy array A into array B.
- 2(b). Find A_{max} .
- 5(a). Arrange array B in ascending order.
- 5(b). Include Y in array B at the appropriate place.

Now processes executing steps 2(b) and 5(a) are independent processes, so these steps can be executed concurrently. Once step 2(b) has been performed, step 4 can also be performed concurrently with step 5(a). Thus, the problem can be coded as the set of six processes shown in Figure 3.25.

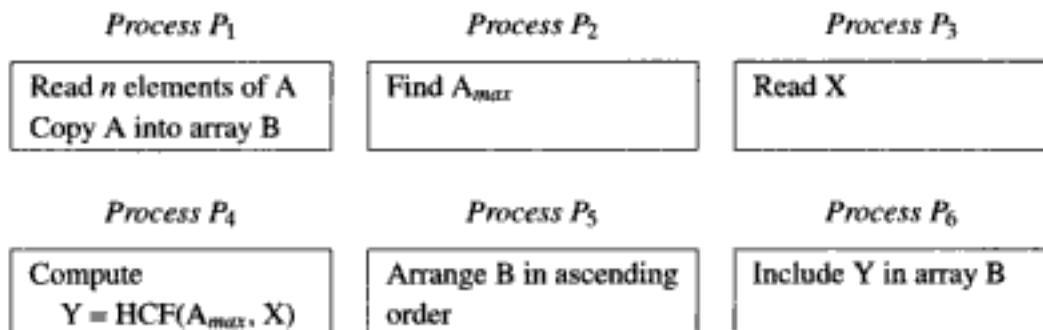


Fig. 3.25 Concurrent processes for Ex. 3.10

Processes P_1 and P_3 can be initiated concurrently. Processes P_1 and P_2 cannot be initiated concurrently because they share array A. Process P_2 must be initiated after process P_1 finishes because of the order in which these steps appear in the problem specification. For a similar reason process P_4 can be initiated only when both P_2 and P_3 terminate. Process P_5 can be initiated as soon as process P_1 terminates, while P_6 can start only after P_4 and P_5 have terminated.

The motivation for structuring the application into a set of concurrent processes in the manner of Example 3.10 is to obtain the benefits of multiprogramming and multiprocessing within a program. Process P_3 is an I/O-bound process whose execution partially overlaps with P_2 , which is a CPU-bound process, and P_1 , which is CPU-bound in later half of its lifetime. This enables the program to make faster progress than if it were to be coded as a single process. Overlapped execution of P_4 and P_5 —both of which are CPU-bound processes—is an advantage if the computer system has multiple CPUs. Note that we have not described how control synchronization is implemented. It is described in Chapter 9.

3.6.3 Message Passing

In Section 3.2.1 we discussed how processes use message passing to exchange information. Such messages are called *interprocess messages*. Message passing is implemented through two system calls. These calls are issued by the library functions `send` and `receive`, respectively.



Fig. 3.26 Interprocess messages

Figure 3.26 shows message passing between processes P_i and P_j . Process P_i sends a message msg_k to process P_j by executing the function call `send (Pj, msgk)` which leads to a system call `send`. The kernel has to ensure that the message msg_k reaches process P_j when it wishes to receive a message, i.e., when it executes the system call `receive`. To implement this, the kernel first copies the message into a buffer area and awaits a `receive` call from process P_j . This call occurs when P_j executes the function call `receive (Pi, alpha)`. The kernel now copies msg_k out of its buffer and into the data area allocated to `alpha`.

The `send` and `receive` calls are executed by different processes, so one cannot assume that the `send` call will always precede the `receive` call. The kernel has to ensure that the intended functionality is implemented regardless of the order in which these calls are executed. This is achieved as follows: If no message has been sent to P_j by the time P_j executes a `receive` call, the kernel blocks P_j and activates it when a message arrives for it. If many messages have been sent to P_j , the kernel queues them and delivers them in FIFO order when P_j executes `receive` calls.

In principle, processes can communicate using shared variables. Processes P_i and P_j can declare some shared variables, and agree to leave messages for each other in these shared variables. However, this arrangement would necessitate synchronization of their activities. Hence message passing is preferred. By avoiding shared variables, message passing also facilitates communication between processes of different applications and processes executing in different computer systems. Table 3.12 summarizes these and other advantages of message passing. Details of interprocess communication are described in Chapter 10.

3.6.4 Signals

The signals mechanism is implemented along the same lines as interrupts. A process P_i wishing to send a signal to another process P_j invokes the library function `signal` with two parameters: id of the destination process, i.e., P_j , and a signal number that indicates the kind of signal to be passed. This function uses the software interrupt instruction `<SI_instrn> <interrupt_code>` to make a system call called `signal`. The parameters (or their addresses) are put in the registers of the CPU before making the

Table 3.12 Advantages of message passing

1. Processes do not need to use shared data for exchanging information.
2. Message passing is tamper-proof as messages reside in the system buffers until delivery.
3. The kernel can give a warning if the data area mentioned in a *receive* call is smaller in size than the message to be received.
4. The kernel takes the responsibility to block a process executing a *receive* when no messages exist for it.
5. The processes may belong to different applications and may even exist in different computer systems.

call. The event handling routine for the *signal* call extracts the parameters to find the signal number. It now makes a provision to pass the signal to P_j and returns. It does not make any change in the state of the sender process, i.e., P_i .

Two interesting issues arise in the implementation of signals. The process sending a signal should know the id of the destination process. This requirement restricts the scope of signals to processes within a process tree. The second issue concerns kernel action if the process to which a signal is being sent is in the *blocked* state. The kernel would have to change the state of the process temporarily to *ready* so that it can execute its signal handling code. After the signal handling code is executed, the kernel would have to change the process state back to *blocked*. Some operating systems prefer a simpler approach that merely notes the arrival of a signal if the destination process is blocked and arranges to execute the signal code when the process is activated.

At the start of this subsection it was mentioned that the signals mechanism is implemented along the same lines as interrupts. Where does that come into the picture? In Section 2.1.1 we described how the interrupt hardware uses the notion of interrupt vectors. A similar arrangement can be defined for each process. Thus for each process a signal vector would exist for every signal that may be sent to it. It would contain the address of a handler. A process can provide handlers for different signals and make their addresses known to the kernel through the *init signal* call. When a signal is sent, the kernel can check whether the destination process has specified a handler for it. If so, the kernel would arrange to pass control to the handler; otherwise, the kernel would execute a default handler for that signal. This scheme can be implemented by adding a field to the PCB to hold the address of the signal vectors area.

Example 3.11 Figure 3.27 illustrates the arrangement used for handling signals. Process P_i makes a library call *init_sig(sig1, sh1)* where *sig1* is the id of the signal to be initialized and *sh1* is the address of the signal handler. The library routine *init_sig* makes the system call *init signal*. While handling this call, the kernel accesses the PCB of P_i , obtains the start address of the signal vectors area, and enters

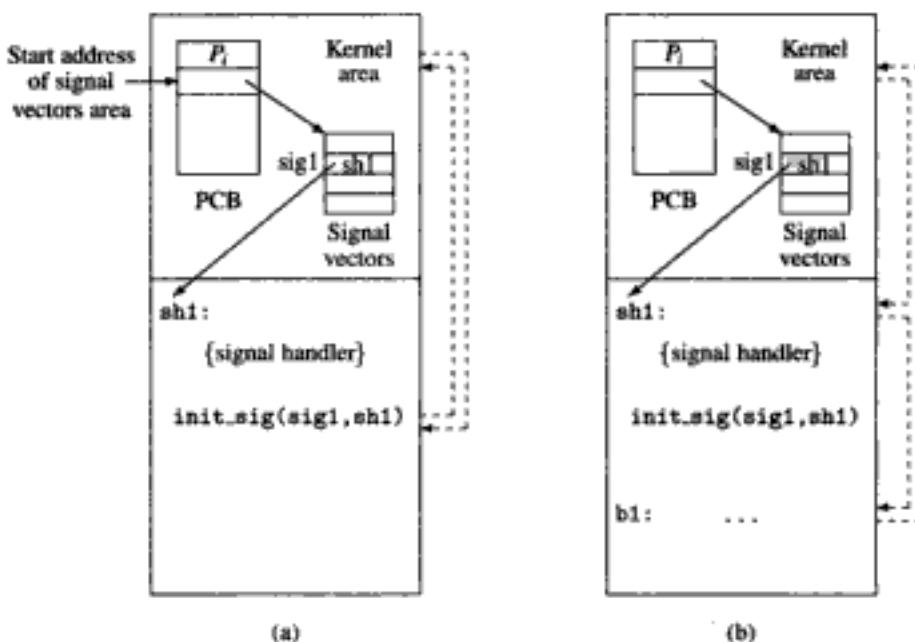


Fig. 3.27 Signal handling by process P_i : (a) Signal initialization, (b) Signal handling

the address sh1 in the signal vector of signal sig1 (see Figure 3.27(a)). Control now returns to P_i , which operates normally until signal sig1 is sent to it. The broken arrows indicate how the CPU is switched to the kernel when the system call is made and how it is switched back to P_i .

When some process P_j wishes to send signal sig1 to P_i , it makes the system call $\text{signal}(P_i, \text{sig1})$. The kernel locates the PCB of P_i , obtains the address of the signal vectors area and locates the signal vector for sig1 . It now arranges for process P_i to execute the signal handler starting at address sh1 before resuming its execution (see Figure 3.27(b)). This is achieved by saving the state of P_i (which shows that it is about to execute an instruction at address b1), changing the contents of the program counter in P_i 's state to address sh1 , and resuming operation of P_i . P_i would now execute the signal handler. The kernel would also have to make provision that after execution of the signal handler has been completed, the state of P_i is reset to the saved state to resume normal operation of P_i . In effect, as shown by the broken arrows in Figure 3.27(b), P_i 's execution is diverted to execution of the signal handler starting at address sh1 , and it is resumed after the signal handler is executed.

EXERCISE 3

1. In some situations, a change in the state of one process may cause a change in the state of another process. Describe all such situations.
2. Describe actions of the kernel when processes make system calls for following purposes:
 - (a) A receive request for a message
 - (b) A memory request

- (c) Request for status information concerning a process
 - (d) Request to create a process
 - (e) Request to terminate a child process.
3. Describe conditions under which state transitions between the *ready*, *blocked*, *ready swapped* and *blocked swapped* states of Figure 3.7 take place.
 4. Describe conditions under which a kernel may perform dispatching without performing scheduling.
 5. Describe how the number of scheduling actions performed in an OS depends on occurrence of events in the system. Develop a formula to calculate the number of times an OS performs scheduling.
 6. Comment on the following in the context of processes in Unix:
 - (a) Implementation of the *wait* call using the PCB data structure.
 - (b) Comparative lifetimes of a process and its PCB.
 7. Describe need for the *zombie* state in Unix.
 8. Describe how each signal listed in Table 3.11 is raised and handled in Unix.
 9. A signal sent to a process in the *blocked* state can be handled in two ways—either temporarily activate the process to handle the signal and put it back in the *blocked* state, or let the process handle the signal next time it enters the *running* state. Discuss how these alternatives can be implemented.
 10. In the reservations system of Example 3.8 and Figure 3.22, is it preferable to use threads rather than processes? Explain with reasons.
 11. Make a list of system calls a thread should avoid using if threads are implemented at the user level.
 12. Comment on validity of the following statement: “Concurrency increases the scheduling overhead without providing any speed-up of an application program.”
 13. An application is to be coded using threads. Describe conditions under which you would recommend use of (a) kernel-level threads, (b) user-level threads.
 14. Write a short note on how to decide the number of user threads and LWPs that should be created in an application.
 15. An OS supports both user-level threads and kernel-level threads. Justify the following recommendations found in a system programmer’s handbook:
 - (a) If a candidate for a thread is a CPU-bound computation, make it a kernel-level thread if the system contains multiple processors; otherwise, make it a user-level thread.
 - (b) If a candidate for a thread is an I/O-bound computation, make it a user-level thread if the process containing it does not contain a kernel-level thread; otherwise, make it a kernel-level thread.
 16. A scalable application server is to be implemented to handle a web-based application. The server handles requests received in the form of messages. It starts a new thread if the request queue exceeds a certain number of entries, and shuts down some thread(s) if the message load decreases. Explain how the server can be implemented using threads. Can signals be used to advantage?
 17. A process creates 10 child processes. It is required to organize the child processes into two groups of 5 processes each such that processes in a group can send signals to

- other processes in the group, but not to processes outside the group. Implement this requirement using features in Unix.
18. If the reservations system of Example 3.8 and Figure 3.22 also contains a query process that simply checks availability of seats on a flight, can a race condition arise due to execution of query and booking operations? Justify your answer.
 19. What are the control and data access synchronization requirements of the processes in Figure 3.25? Explain how these synchronization requirements can be implemented.

BIBLIOGRAPHY

The process concept is discussed in Dijkstra (1968), Brinch Hansen (1973) and Bic and Shaw (1974). Brinch Hansen (1988) describes implementation of processes in the RC 4000 system.

Marsh *et al* (1991) discuss user-level threads and issues concerning thread libraries. Anderson *et al* (1992) discuss use of scheduler activations for communication between the kernel and a thread library. Engelschall (2000) discusses how user level threads can be implemented in Unix using standard Unix facilities. He also summarizes properties of other multithreading packages.

Kleiman (1996), Butenhof (1997), Lewis and Berg (1997) and Nichols *et al* (1996) discuss programming with POSIX threads. Lewis and Berg (2000) discuss multithreading in Java.

Bach (1986), McKusick (1996) and Vahalia (1996) discuss processes in Unix. Beck *et al* (2002) and Bovet and Cesati (2003) describe processes and threads in Linux. Stevens and Rago (2005) describe processes and threads in Unix, Linux and BSD; they also discuss daemon processes in Unix. O'Gorman (2003) discusses implementation of signals in Linux. Eykholt *et al* (1992) describe threads in SunOS, while Vahalia (1996) and Mauro and McDougall (2001) describe threads and LWPs in Solaris. Custer (1993), Richter (1999) and Russinovich and Solomon (2005) describe processes and threads in Windows. Vahalia (1996) and Tanenbaum (2001) discuss threads in Mach.

1. Anderson, T. E., B. N. Bershad, E. D. Lazowska, and H. M. Levy (1992): "Scheduler activations: effective kernel support for the user-level management of parallelism," *ACM Transactions on Computer Systems*, **10** (1), 53–79.
2. Bach, M. J. (1986): *The Design of the Unix Operating System*, Prentice-Hall, Englewood Cliffs.
3. Beck, M., H. Bohme, M. Dziadzka, U. Kunitz, R. Magnus, C. Schroter, and D. Verworner (2002): *Linux Kernel Programming, Third edition*, Pearson Education.
4. Bic, L., and A. C. Shaw (1988): *The Logical Design of Operating Systems, Second edition*, Prentice-Hall, Englewood Cliffs.
5. Brinch Hansen, P. (1970): "The nucleus of a multiprogramming system," *Communications of the ACM*, **13**, 238–241, 250.
6. Brinch Hansen, P. (1973): *Operating System Principles*, Prentice-Hall, Englewood Cliffs.
7. Bovet, D. P., and M. Cesati (2003): *Understanding the Linux Kernel*, O'Reilly, Sebastopol.
8. Butenhof, D. (1997): *Programming with POSIX threads*, Addison Wesley, Reading.
9. Custer, H. (1993): *Inside Windows/NT*, Microsoft Press, Redmond.

10. Dijkstra, E. W. (1968): "The structure of THE multiprogramming system," *Communications of the ACM*, **11**, 341–346.
11. Engelschall, R. S. (2000): "Portable Multithreading: The signal stack trick for user-space thread creation," *Proceedings of the 2000 USENIX Annual Technical Conference*, San Diego.
12. Eykholt, J. R., S. R. Kleiman, S. Barton, S. Faulkner, A. Shivalingiah, M. Smith, D. Stein, J. Voll, M. Weeks, and D. Williams (1992): "Beyond multiprocessing: multithreading the SunOS kernel," *Proceedings of the Summer 1992 USENIX Conference*, 11–18.
13. O'Gorman, J. (2003): *Linux Process Manager: The internals of Scheduling, Interrupts and Signals*, Wiley and Sons.
14. Kleiman, S., D. Shah, and B. Smaalders (1996): *Programming with Threads*, Prentice-Hall, Englewood Cliffs.
15. Lewis, B., and D. Berg (1997): *Multithreaded Programming with Pthreads*, Prentice-Hall, Englewood Cliffs.
16. Lewis, B., and D. Berg (2000): *Multithreaded Programming with Java Technology*, Sun Microsystems.
17. Mauro, J., and R. McDougall (2001): *Solaris Internals—Core Kernel Architecture*, Prentice-Hall.
18. Marsh, B. D., M. L. Scott, T. J. LeBlanc, and E. P. Markatos (1991): "First-class user-level threads," *Proceedings of the Thirteenth ACM Symposium on Operating Systems Principles*, October 1991, 110–121.
19. McKusick, M. K., K. Bostic, M. J. Karels, and J. S. Quarterman (1996): *The Design and Implementation of the 4.4 BSD Operating System*, Addison Wesley, Reading.
20. Nichols, B., D. Buttlar, and J. P. Farrell (1996): *Pthreads Programming*, O'Reilly and Associates, Sebastopol.
21. Richter, J. (1999): *Programming Applications for Microsoft Windows, Fourth edition*, Microsoft Press, Redmond.
22. Russinovich, M. E., and D. A. Solomon (2005): *Microsoft Windows Internals, Fourth edition*, Microsoft Press, Redmond.
23. Silberschatz, A., P. B. Galvin, and G. Gagne (2005): *Operating System Principles, Seventh edition*, John Wiley, New York.
24. Stevens, W. R., and S. A. Rago (2005): *Advanced Programming in the Unix Environment, Second edition*, Addison Wesley Professional.
25. Tanenbaum, A. S. (2001): *Modern Operating Systems, Second edition*, Prentice-Hall, Englewood Cliffs.
26. Vahalia, U. (1996): *Unix Internals—The New Frontiers*, Prentice-Hall, Englewood Cliffs.

chapter 4

Scheduling

The scheduling policy used in an operating system influences user service, efficient use of resources and system performance. Scheduling policies use the fundamental techniques of preemption, reordering of requests and variation of time slice to achieve their goals. We discuss how these techniques are used in classical non-preemptive and preemptive scheduling policies.

An operating system has to adapt its functioning to the availability of resources in the system; it uses a combination of three schedulers called long-term, medium-term and short-term schedulers for this purpose. An important issue in practical scheduling is fairness in user service. We discuss how operating systems incorporate fairness by dynamically varying the priority of a process. We also study scheduling in a real time environment.

Performance analysis of scheduling policies is important for tuning the performance of a scheduling policy and for comparing alternative policies. We discuss important elements of performance analysis.

4.1 PRELIMINARIES

Scheduling is the activity of selecting the next request to be serviced by a server. A scheduling policy determines the quality of service provided to users. It also influences the performance of a computer system. In Sections 2.5.3.1 and 2.6.2, we discussed the scheduling policies used in multiprogramming and time sharing systems, respectively, and saw how the techniques of preemption, assignment of priorities to processes, and time slicing are used to achieve the desired combination of user service and system performance. In this Chapter, we discuss features and properties of many other scheduling policies.

4.1.1 Scheduling Concepts and Terminology

Figure 4.1 shows a schematic of scheduling. All requests waiting to be serviced are kept in a list of pending requests. An arriving request is added to this list. Whenever scheduling is to be performed, the scheduler examines the pending requests and selects one for servicing (this is shown by the dashed arrow in Figure 4.1). This request is handed over to the server. A request leaves the server when it completes or when it is preempted by the scheduler, in which case it is put back into the list of pending requests. In either situation, scheduler performs scheduling to select the next request to be serviced. Thus, four events related to scheduling are *arrival*, *scheduling*, *preemption* and *completion*.

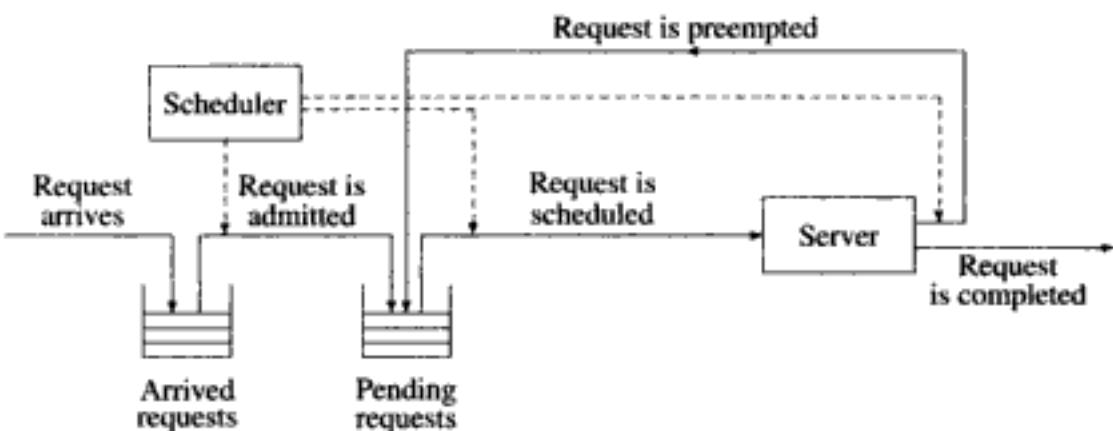


Fig. 4.1 A schematic of scheduling

A *request* is the execution of a job or a process. A user submits a request and waits for its completion. In an interactive environment, a user may interact with a process during its execution—the user makes a *subrequest* to a process and the process responds by performing an action or by computing a result.

Table 4.1 summarizes scheduling-related concepts and terms. Service time of a job or process is the total of CPU time and I/O time required by it to complete its operation. It is an inherent property of a job or process. The total time spent by a job or a process in the OS may be longer than its service time because there might be times when it is neither executing on the CPU nor performing I/O. Consequently, its completion time depends on its arrival time, service time, and the kind of service it is provided by the OS. Scheduling related concepts can be grouped into user-centric concepts and system-centric concepts. We discuss these in the following.

User-centric scheduling concepts Response time, deadline overrun, turn around time and weighted turn around are user-centric or application-centric views of a scheduler's performance. Response time is the time since submission of a subrequest to the time its processing is completed. It is an absolute measure of service

Table 4.1 Scheduling terms and concepts

Term or concept	Definition or description
<i>Request related</i>	
Arrival time	Time when a user submits a job/process.
Admission time	Time when the system starts considering a job/process for scheduling.
Completion time	Time when a job/process is completed.
Deadline	Time by which a job/process must be completed to meet the response requirement of a real time application.
Service time	The total of CPU time and I/O time required by a job/process or subrequest to complete its operation.
Preemption	Forced deallocation of CPU from a job or process.
Priority	Priority is a tie-breaking rule used to select a request when many requests await service.
<i>User service related: individual request</i>	
Deadline overrun	The amount of time by which the completion time of a job/process exceeds its deadline. Deadline overruns can be both positive or negative.
Fair share	A specified share of CPU time that should be devoted to execution of a process or a group of processes.
Response ratio	The ratio $\frac{\text{time since arrival} + \text{service time of the process}}{\text{service time of the process}}$
Response time (rt)	Time between the submission of a subrequest for processing to the time its result becomes available. This concept is applicable to interactive processes.
Turnaround time (ta)	Time between the submission of a job/process and its completion by the system. This concept is meaningful for non-interactive jobs or processes only.
Weighted turnaround (w)	Ratio of the turnaround time of a job/process to its own service time.
<i>User service related: average service</i>	
Mean response time (\bar{rt})	Average of the response times of all subrequests serviced by the system.
Mean turnaround time (\bar{ta})	Average of the turnaround times of all jobs/processes serviced by the system.
<i>Scheduling related</i>	
Schedule length	The time taken to complete a specific set of jobs/processes.
Throughput	The average number of jobs/processes or subrequests completed by a system in one unit of time.

provided to a subrequest. Turn around time is an analogous absolute measure of service provided to a request. Turn around time differs from the service time of a request because it also includes time when the request exists in the system but is neither executing on the CPU nor performing I/O operations. The mean turn around time is a measure of the average service provided to requests or processes. The weighted turn around relates the turn around time of a process to its own service time. For example, a weighted turn around of 5 indicates that the turn around received by a request is 5 times its own service time. Comparison of weighted turn arounds of different requests indicates the comparative service received by them.

Deadlines are important in real time applications. Deadline overrun is the amount of time by which a process has missed its deadline. A negative value of deadline overrun indicates that a process was completed before its deadline, whereas a positive value indicates that the deadline was missed. Deadlines are not meaningful in non-real time applications.

System-centric scheduling concepts Throughput and schedule length are system-centric performance criteria. Throughput indicates the average number of requests or subrequests completed per unit of time (see Section 2.5). It provides a good basis for comparing performance of two or more systems, or for comparing performance of the same system over different periods of time. Schedule length indicates the total amount of time taken by a server to complete a set of requests.

Throughput and schedule length are related. However, in an OS, they cannot be computed from one another. Schedule length can be computed only when a server handles a fixed set of requests and stops. An OS may start processing new requests even before previous requests are completed, so it is not possible to compute schedule length for a subset of requests processed by an OS. Nevertheless, schedule length is an important basis for comparing the performance of scheduling policies, especially when scheduling overhead cannot be ignored.

From a system administrator's viewpoint, schedule length and throughput are important indices of system performance. For a user, the turn around time of a request is an absolute index of service. Weighted turn arounds of different requests reflect on the comparative service given to them—closely spaced values of weighted turn arounds indicate fairness of service.

4.1.2 Fundamental Techniques of Scheduling

Schedulers use three fundamental techniques to provide good user service, efficiency of use, or performance of the system.

- *Priority-based scheduling:* The process in execution is the highest priority process requiring use of the CPU. It is preempted when a process with a higher priority becomes *ready*.
- *Reordering of requests:* Reordering implies servicing of requests in some order other than their arrival order. Reordering may be used by itself, or in conjunc-

tion with preemption. In the former case, it can help to improve user service. For example, servicing short requests before long ones reduces the average turn-around time of requests. Reordering can be used in conjunction with preemption to enhance user service, as in a time sharing system, or to enhance system throughput, as in a multiprogramming system.

- **Variation of time slice:** When time slicing is used, from Eq. (2.3) of Section 2.6.2, $\eta = \frac{\delta}{\delta + \sigma}$ where η is the CPU efficiency, δ is the time slice and σ is the OS overhead per scheduling decision. Better response times are obtained for smaller values of the time slice; however, CPU efficiency is lower because considerable switching overhead is incurred. To balance CPU efficiency and response times, an OS could use different values of δ for different requests—use a small value of δ for I/O-bound requests, and a large value for CPU-bound requests—or, it could vary the value of δ for a process when its behavior changes from CPU-bound to I/O-bound, or vice versa.

In Sections 4.2 and 4.3 we discuss how these techniques are used in classical non-preemptive and preemptive scheduling policies. In Section 4.4, we discuss how they are used in practice.

4.1.3 The Role of Priority

Priority is a tie-breaking rule that is employed by a scheduler when many requests await attention of the server. The priority of a request can be a function of several parameters, each parameter reflecting either an inherent attribute of the request, or an aspect concerning its service. It is called a *dynamic priority* if some of its parameters change during the operation of the request; otherwise, it called a *static priority*.

Some process reorderings could be obtained through priorities as well. For example, short processes would be serviced before long processes if priority is inversely proportional to the service time of a process and processes that have received less CPU time would be processed first if the priority is inversely proportional to the CPU time consumed by a process. However, to restrict overhead, schedulers like to avoid using complex priority functions or dynamic priorities; they employ algorithms that determine the order in which requests should be serviced.

If two or more requests have the same priority, which of them should be scheduled first? A popular scheme is to use round-robin scheduling for requests with the same priority. This way, processes with the same priority share the CPU among themselves when none of the higher priority processes is ready, which provides better user service than if one of the requests is favored over other requests with the same priority.

Priority-based scheduling has the drawback that a low priority request may never be serviced if high priority requests keep arriving. This situation is called *starvation*. It could be avoided by incrementing the priority of a request if it does not get scheduled for a certain period of time. Thus, the priority of a low priority request would

keep increasing as it waits to get scheduled until its priority exceeds the priority of all other pending requests. At this time, it would get scheduled. This technique is called *aging* of requests.

4.2 NON-PREEMPTIVE SCHEDULING POLICIES

In non-preemptive scheduling, a server always processes a scheduled request to completion. Thus preemption of a request as shown in Figure 4.1 never occurs. Scheduling is performed only when processing of the previously scheduled request gets completed. Non-preemptive scheduling is attractive due to its simplicity—it is not necessary to maintain a distinction between an unserviced request and a partially serviced one.

Since preemption is not used, the scheduler depends on reordering of requests to achieve an improvement in user service or system performance. We discuss three non-preemptive scheduling policies in this Section:

- FCFS scheduling
- Shortest request next (SRN) scheduling
- Highest response ratio next (HRN) scheduling.

We use the five processes shown in Table 4.2 to discuss the operation and performance of various scheduling policies. For simplicity we assume that a process does not perform any I/O operations.

Table 4.2 Processes for scheduling

Process	P_1	P_2	P_3	P_4	P_5
Arrival time	0	2	3	5	9
Service time	3	3	2	5	3

FCFS scheduling Requests are scheduled in the order in which they arrive in the system. The list of pending requests is organized as a queue. The scheduler always schedules the first request in the list. An example of FCFS scheduling is a batch processing system in which jobs are ordered according to their arrival times (or arbitrarily if their arrival times are identical), and results of a job are released to a user immediately on completion of a job. Example 4.1 illustrates operation of an FCFS scheduler.

Example 4.1 Top half of Figure 4.2 shows the scheduling decisions performed by the FCFS policy for the processes of Table 4.2. Left half of Table 4.3 summarizes performance of the FCFS scheduler. The *Completed* column shows id of the completed process and its turn around time (ta) and weighted turn around (w). Note that considerable variation exists in the weighted turn arounds. This variation would have been larger if processes suffering large turn around times were short, e.g., the weighted turn

around of P_5 would have been large if its execution requirement was 1.0 second or 0.5 seconds. Mean ta and w (i.e., \bar{ta} and \bar{w}) are shown below the table.

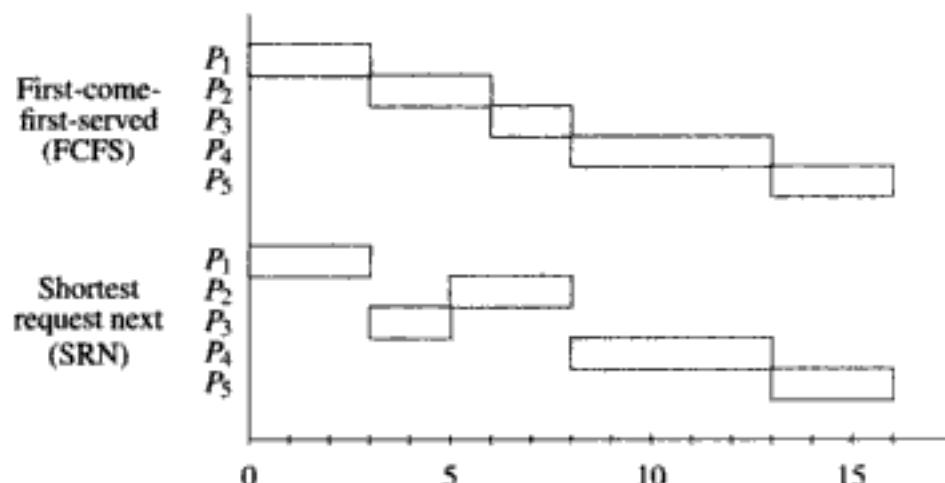


Fig. 4.2 Scheduling using FCFS and SRN policies

Table 4.3 Performance of FCFS and SRN scheduling

Time	FCFS						SRN					
	Completed			Processes in system	Sche- duled	Completed			Processes in system	Sche- duled		
	<i>id</i>	<i>ta</i>	<i>w</i>			<i>id</i>	<i>ta</i>	<i>w</i>				
0	-	-	-	{P ₁ }	P ₁	-	-	-	{P ₁ }	P ₁		
3	P ₁	3	1.00	{P ₂ , P ₃ }	P ₂	P ₁	3	1.00	{P ₂ , P ₃ }	P ₃		
5						P ₃	2	1.00	{P ₂ , P ₄ }	P ₂		
6	P ₂	4	1.33	{P ₃ , P ₄ }	P ₃	P ₄	6	2.00	{P ₄ }	P ₄		
8	P ₃	5	2.50	{P ₄ }	P ₄	P ₂	8	1.60	{P ₅ }	P ₅		
13	P ₄	8	1.60	{P ₅ }	P ₅	P ₄	8	1.60	{P ₅ }	P ₅		
16	P ₅	7	2.33	{}	-	P ₅	7	2.33	{}	-		

$$\bar{ta} = 5.40 \text{ seconds}$$

$$\bar{w} = 1.75$$

$$\bar{ta} = 5.20 \text{ seconds}$$

$$\bar{w} = 1.59$$

Example 4.1 shows that wide disparity exists in the service received by requests in FCFS scheduling. From this, one can conclude that short requests (processes P_3 and P_5) may suffer high weighted turn arounds, i.e., poorer service, compared to other requests (process P_4). In a batch processing system, jobs in the batch may be re-ordered to reduce turn around times and weighted turn arounds of individual jobs. However this would not reduce the spread of turn around times or increase the throughput.

Shortest request next (SRN) scheduling The SRN scheduler always schedules the

shortest of arrived requests. Thus, a request remains pending until all shorter requests have been serviced.

Example 4.2 The bottom half of Figure 4.2 shows the scheduling decisions performed by the SRN policy for the processes of Table 4.2. The right half of Table 4.3 summarizes performance of the SRN scheduler. The mean turn around and the mean weighted turn around are better than in FCFS scheduling because short requests tend to receive smaller turn around times and weighted turn arounds than in FCFS scheduling. This degrades the service to long requests however their weighted turn arounds do not increase much because their service times are large. The throughput is higher than in FCFS scheduling except at the end of the schedule, where it is identical.

It is difficult to implement the SRN policy in practice because service times of processes are not known a priori. Many systems expect users to provide estimates of service times of processes. However, results obtained using such data are erratic if users do not possess sufficient experience in estimating service times. Dependence on user estimates also leaves the system open to abuse or manipulation, as users might try to obtain better service by specifying low values for service times of their processes. Another practical aspect where SRN scheduling does not fare well is in the service offered to long processes. A steady stream of short processes arriving in the system can deny CPU to long processes indefinitely. This is starvation.

Highest Response Ratio Next (HRN) The HRN policy computes the response ratios of all processes in the system according to Eq. (4.1) and selects the process with the highest response ratio.

$$\text{Response ratio} = \frac{\text{Time since arrival} + \text{Service time of the process}}{\text{Service time of the process}} \quad (4.1)$$

The response ratio of a newly arrived process is 1. It keeps increasing at the rate (1 / service time) as it waits to be serviced; the response ratio of a short process increases more rapidly than that of a long process, so shorter processes are favored for scheduling. However, the response ratio of a long process eventually becomes large enough for the process to get scheduled. This feature provides an effect similar to the technique of *aging* discussed earlier in Section 4.1.3, so long processes do not starve. The next example illustrates this property.

Example 4.3 Figure 4.3 summarizes operation of the HRN scheduling policy for the five processes P_{11} – P_{15} . By the time process P_{11} completes, processes P_{12} and P_{13} have arrived. P_{12} has a larger response ratio than P_{13} , so it is scheduled next. When it completes, P_{13} has a higher response ratio than before; however, P_{14} , which arrived after P_{13} , has an even higher response ratio because it is a shorter process, so P_{14} is scheduled. When P_{14} completes, P_{13} has a higher response ratio than the shorter process P_{15} because it has spent a lot of time waiting, whereas P_{15} has just arrived. Hence P_{13} is scheduled now. Thus, a long process gets scheduled ahead of a short process because it has waited longer.

Process	P_{11}	P_{12}	P_{13}	P_{14}	P_{15}
Arrival time	0	2	3	4	8
Service time	3	3	5	2	3

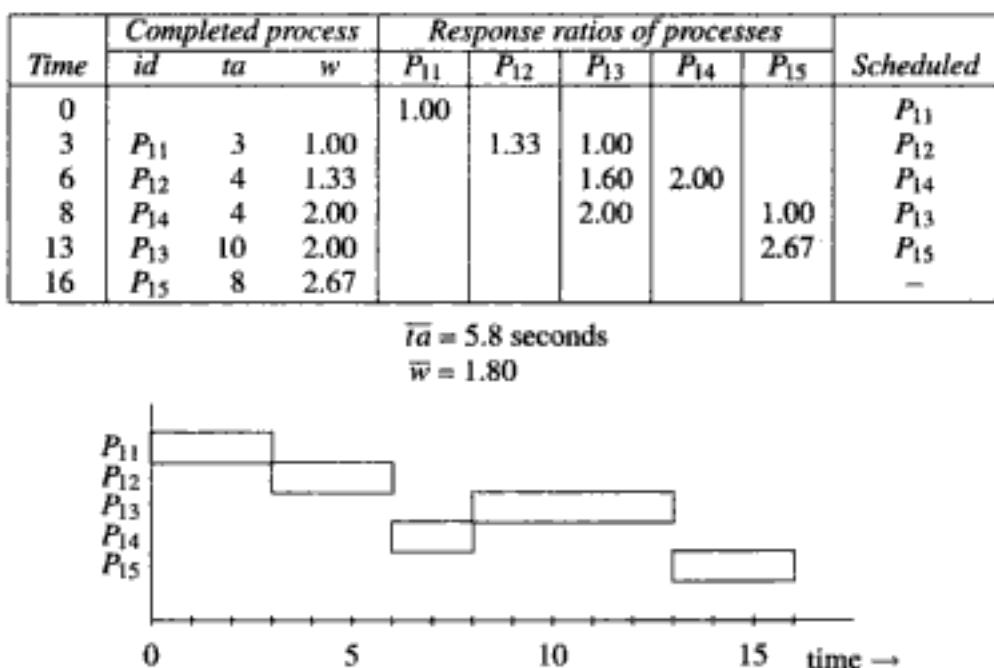


Fig. 4.3 Operation of highest response ratio (HRN) policy

4.3 PREEMPTIVE SCHEDULING POLICIES

In preemptive scheduling, the server can be switched to the processing of a new request before completing the current request. The preempted request is put back into the list of pending requests (see Figure 4.1). Its servicing would be resumed when it is scheduled again. Thus, a request may have to be scheduled many times before it completes. In Chapter 2 we saw the preemptive scheduling policies used in multiprogramming and time sharing operating systems.

We discuss four preemptive scheduling policies in this section:

- Round-robin scheduling with time slicing (RR)
- Least completed next (LCN) scheduling
- Shortest time to go (STG) scheduling
- Highest response-ratio next (HRN) scheduling.

Figure 4.4 shows the scheduling decisions made by these preemptive scheduling policies when an OS executes the processes of Table 4.2 if scheduling decisions are made every second and a process does not perform I/O operations. Table 4.4 summarizes the operation of preemptive scheduling policies. Table 4.5 summarizes the performance of these policies.

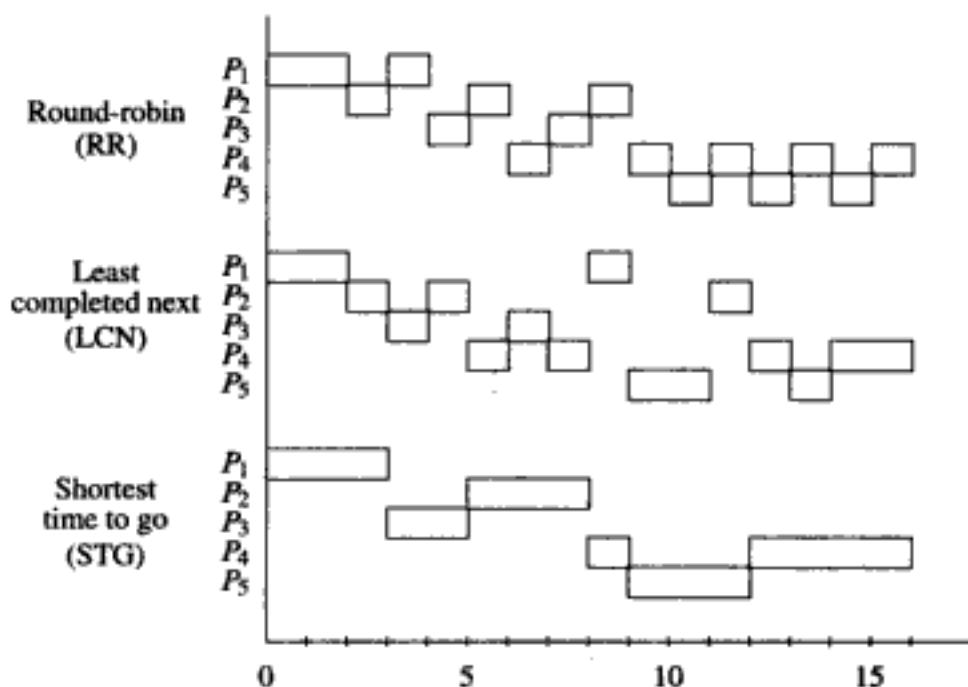


Fig. 4.4 Scheduling using preemptive scheduling policies

Round-Robin scheduling Round-robin (RR) scheduling with time slicing is aimed at providing fair service to all requests. Time slicing is used to limit the amount of CPU time a process may use when scheduled. A request is preempted if the time slice elapses. This policy maintains the weighted turn arounds of processes approximately equal to the number of active processes in the system. Variations may arise only due to the nature of processes, e.g., an I/O-bound process may lag behind CPU-bound processes in its use of the CPU. The RR policy does not fare well in terms of system performance indices like throughput since it treats all processes alike and does not give a favored treatment to short processes. Example 4.4 illustrates the performance of RR scheduling.

Example 4.4 The left part of Table 4.4 summarizes operation of the RR scheduler with $\delta = 1$ for the five processes shown in Table 4.2. The scheduler makes scheduling decisions every second. The column *Processes* shows the queue of processes in the system. The scheduler simply selects the first process for scheduling. The running process is preempted when the time slice elapses. It is now put at the end of the queue. It is assumed that a new process admitted into the system at the same instant is put into the queue before the preempted process.

The turn around times and weighted turn arounds of the processes are as shown in the leftmost part of Table 4.5. The C column shows completion times. The turn around times and weighted turn arounds are inferior compared to non-preemptive policies. This is so because the CPU time is divided among many processes due to time slicing. It can be seen that processes P_2, P_3 and P_4 , which arrive at around the same time receive approximately equal weighted turn arounds. P_3 receives the worst weighted

Table 4.4 Operation of preemptive scheduling policies

Time	Round-robin (RR)		Least completed next (LCN)		Shortest time to go (STG)	
	Processes	Scheduled	Processes	Scheduled	Processes	Scheduled
0	P_1	P_1	$P_1:0$	P_1	$P_1:3$	P_1
1	P_1	P_1	$P_1:1$	P_1	$P_1:2$	P_1
2	P_2, P_1	P_2	$P_1:2, P_2:0$	P_2	$P_1:1, P_2:3$	P_1
3	P_1, P_3, P_2	P_1	$P_1:2, P_2:1, P_3:0$	P_3	$P_2:3, P_3:2$	P_3
4	P_3, P_2	P_3	$P_1:2, P_2:1, P_3:1$	P_2	$P_2:3, P_3:1$	P_3
5	P_2, P_4, P_3	P_2	$P_1:2, P_2:2, P_3:1, P_4:0$	P_4	$P_2:3, P_4:5$	P_2
6	P_4, P_3, P_2	P_4	$P_1:2, P_2:2, P_3:1, P_4:1$	P_3	$P_2:2, P_4:5$	P_2
7	P_3, P_2, P_4	P_3	$P_1:2, P_2:2, P_4:1$	P_4	$P_2:1, P_4:5$	P_2
8	P_2, P_4	P_2	$P_1:2, P_2:2, P_4:2$	P_1	$P_4:5$	P_4
9	P_4, P_5	P_4	$P_2:2, P_4:2, P_5:0$	P_5	$P_4:4, P_5:3$	P_5
10	P_5, P_4	P_5	$P_2:2, P_4:2, P_5:1$	P_5	$P_4:4, P_5:2$	P_5
11	P_4, P_5	P_4	$P_2:2, P_4:2, P_5:2$	P_2	$P_4:4, P_5:1$	P_5
12	P_5, P_4	P_5	$P_4:2, P_5:2$	P_4	$P_4:4$	P_4
13	P_4, P_5	P_4	$P_4:3, P_5:2$	P_5	$P_4:3$	P_4
14	P_5, P_4	P_5	$P_4:3$	P_4	$P_4:2$	P_4
15	P_4	P_4	$P_4:4$	P_4	$P_4:1$	P_4
16	—	—	—	—	—	—

turn around because three processes exist in the system through most of its life. P_1 receives the best weighted turn around since no other process exists in the system during the early part of P_1 's execution. Thus weighted turn arounds depend on the load in the system.

Effectiveness of RR scheduling depends on two factors: choice of δ , the time slice, and nature of processes in the system. As discussed in Chapter 2, if a system contains n processes and each request by a process consumes exactly δ seconds, the response time (rt) for a request is

$$rt = n \times (\sigma + \delta) \quad (4.2)$$

where σ is the scheduling overhead per scheduling decision. Since all processes may not be active at all times—some will be blocked for I/O or other requests—the response time will be governed by the number of active processes rather than bn . Two other factors that can affect rt are the variation in CPU time required by different requests, and the relationship between δ and the CPU time required by a request. Eq. (4.2) assumes that each request consumes exactly δ seconds and produces a response. If a request needs more CPU time than δ , then it will have to be scheduled more than once before it can produce a response. Therefore, in certain regions of values of δ , rt for a request may increase even as δ is reduced. Example 4.5 illustrates this fact.

Table 4.5 Performance of preemptive scheduling policies (C: Completion time of process)

Process	Round-robin (RR)			Least completed next (LCN)			Shortest time to go (STG)			Highest response ratio next (HRN)		
	C	ta	w	C	ta	w	C	ta	w	C	ta	w
P ₁	4	4	1.33	9	9	3.00	3	3	1.00	5	5	1.67
P ₂	9	7	2.33	12	10	3.33	8	6	2.00	11	9	3.00
P ₃	8	5	2.50	7	4	2.00	5	2	1.00	8	5	2.50
P ₄	16	11	2.20	16	11	2.20	16	11	2.20	16	11	2.20
P ₅	15	6	2.00	14	5	1.67	12	3	1.00	15	6	2.00

$$\overline{ta} = 6.6 \text{ Seconds} \quad \overline{ta} = 7.8 \text{ Seconds} \quad \overline{ta} = 5.0 \text{ Seconds} \quad \overline{ta} = 7.2 \text{ Seconds}$$

$$\overline{w} = 2.07 \quad \overline{w} = 2.40 \quad \overline{w} = 1.44 \quad \overline{w} = 2.27$$

Example 4.5 An OS contains 10 identical processes that were initiated at the same time. Each process contains 15 identical requests, and each request consumes 20 msec of CPU time. A request is followed by an I/O operation that consumes 10 msec. The system consumes 2 msec in CPU scheduling. For $\delta \geq 20$ msec, the first request by the first process receives a response time of 22 msec and the first request by the last process receives a response time of 220 msec, so the average response time is 121 msec. A subsequent request by any process receives a response time of $10 \times (2 + 20) - 10$ msec = 210 msec since the process spends 10 msec in an I/O wait before making the next request. For $\delta = 10$ msec, a request would be preempted after 10 msec. When scheduled again, it would execute for 10 msec and produce results, so the response time for the first process is

$$= 10 \times (2 + 10) + (2 + 10) = 132 \text{ msec}$$

and that for the last process is

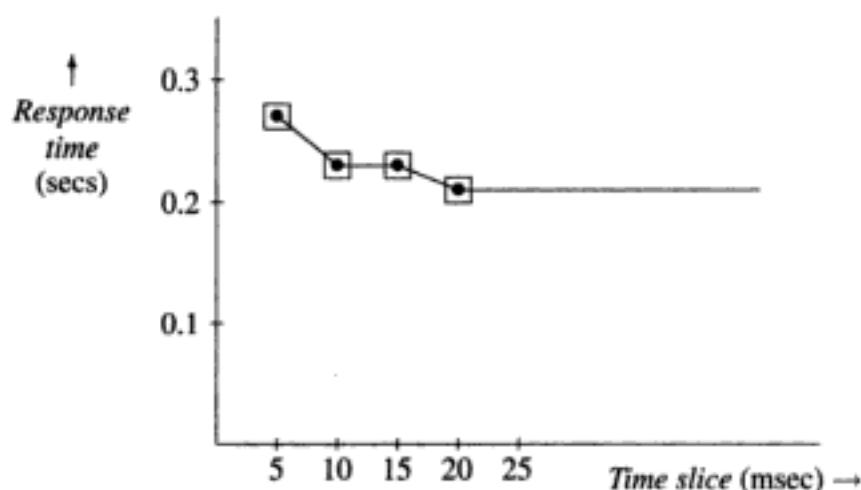
$$= 10 \times (2 + 10) + 10 \times (2 + 10) = 240 \text{ msec.}$$

A subsequent request receives a response time of $10 \times (2 + 10) + 10 \times (2 + 10) - 10 = 230$ msec. Figure 4.5 illustrates the variation of average response time to second and subsequent requests for different values of δ . Table 4.6 summarizes performance of the system for different values of δ . As expected, the overhead is higher for smaller values of δ .

Policies using resource consumption information

Policies using resource consumption information aim at eliminating some problems of preemptive policies discussed so far, e.g., poor weighted turn arounds of short processes and starvation of long processes. We discuss the following three scheduling policies:

1. Least Completed Next (LCN) scheduling
2. Shortest Time to Go (STG) scheduling
3. Response Ratio scheduling.

**Fig. 4.5** Variation of average response time with time slice**Table 4.6** Performance of RR scheduling for different values of δ

Time slice	5 msec	10 msec	15 msec	20 msec
Average rt for first request (msec)	248.5	186	208.5	121
Average rt for subsequent request (msec)	270	230	230	210
Number of scheduling decisions	600	300	300	150
Schedule length (msec)	4200	3600	3600	3300
Overhead (percent)	29	17	17	9

Least completed next (LCN) The LCN policy schedules the process that has consumed the least amount of CPU time. Thus, the nature of a process, whether CPU-bound or I/O-bound, and CPU time requirement of a process do not influence its progress in the system. All processes make approximately equal progress in terms of the CPU time consumed by them, so short processes are guaranteed to finish ahead of long processes. However, this policy has the familiar drawback of starving long processes of CPU attention. It also neglects existing processes if new processes keep arriving in the system. So even not-so-long processes tend to suffer starvation or larger turn around times.

Example 4.6 Operation of the LCN scheduling policy for the five processes shown in Table 4.2 is summarized in the middle part of Table 4.4. The scheduling information used by the policy consists of a pair (process id, CPU time consumed). We use the notation $P_i : t_j$, where P_i is a process id and t_j is the CPU time consumed by it, to denote this information in the column *Processes*. The scheduler analyses this information and selects the process that has consumed the least amount of CPU time. In case of a tie, it selects a process that has not been serviced for a longer period of time.

The turn around times and weighted turn arounds of the processes are shown in the second part of Table 4.5. Processing of P_1, P_2 and P_4 is delayed because new processes

arrive in the system and obtain CPU service before these processes can make further progress. Process P_4 catches up in terms of service during the latter half of its life in the system because no new processes arrive. LCN provides poorer turn around times and weighted turn arounds than those provided by RR and STG policies (see Examples 4.4 and 4.7) because it favors newly arriving processes over existing processes in the system, e.g., it favors P_3 over P_1, P_2 ; and P_5 over P_4 .

Shortest time to go (STG) The STG policy is a preemptive version of the SRN policy. A process is scheduled when its remaining processing requirements are the smallest in the system. The STG policy favors a process that is nearing completion, irrespective of the CPU time already consumed by it. Thus, a long process nearing completion may be favored over short processes entering the system. The long process would finish first, thus improving its weighted turn around, however this may affect turn around times and weighted turn arounds of several short processes in the system. On the other hand, long processes have problems receiving service in early phases of their operation, so favored service to long processes nearing completion may not avoid starvation of long processes.

Example 4.7 The right part of Table 4.4 summarizes performance of the STG scheduling policy for the five processes shown in Table 4.2. The scheduling information used by the policy consists of a pair (process id, CPU time needed for completion). We use the notation $P_i : t_j$, where P_i is a process id and t_j is the CPU time needed by it for completion, to denote this information in the column *Processes*. The scheduler uses this information for selecting the process that requires the least amount of CPU time for completion. In case of a tie, it selects a process that has not been serviced for a longer period of time.

The third part of Table 4.5 shows the turn around times and weighted turn arounds of the processes. Note that processes P_1, P_3 and P_5 obtain good turn around times and weighted turn arounds. In fact the turn around times and weighted turn arounds of all processes other than P_4 are better than those provided by the LCN policy (see Example 4.6). This is so because their service times are shorter than that of P_4 . Hence superior service received by them is at the cost of neglecting P_4 , whose execution does not even begin until P_5 is complete. This neglect of long processes can be seen more starkly if another process P_6 with CPU time requirement of 6 seconds were to arrive at $t = 1$ second. Its execution would begin only after P_4 completes; its turn around time and weighted turn around would be 21 seconds and 3.5, respectively.

4.4 SCHEDULING IN PRACTICE

4.4.1 Long, Medium and Short-term Scheduling

An operating system has to provide a suitable combination of user-centric and system-centric features. It also has to adapt to the nature and number of user requests that are expected to arise in its environment, and to the availability of resources. A single scheduler and a single scheduling policy cannot address all its concerns. Therefore, an OS uses an arrangement consisting of three schedulers called the *long-term*

scheduler, the *medium-term scheduler* and the *short-term scheduler* to address different user-centric and system-centric issues. Table 4.7 summarizes features of these schedulers.

Table 4.7 Long, medium and short-term scheduling

Long-term scheduling	Decides when to admit an arrived process for scheduling based on its nature, whether CPU-bound or I/O-bound, and availability of resources like kernel data structures, user terminals and disk space for swapping.
Medium-term scheduling	Moves processes between the memory and the disk to optimize use of the memory. Maintains a sufficient number of <i>ready</i> processes in the memory.
Short-term scheduling	Decides which <i>ready</i> process to execute next and for how long.

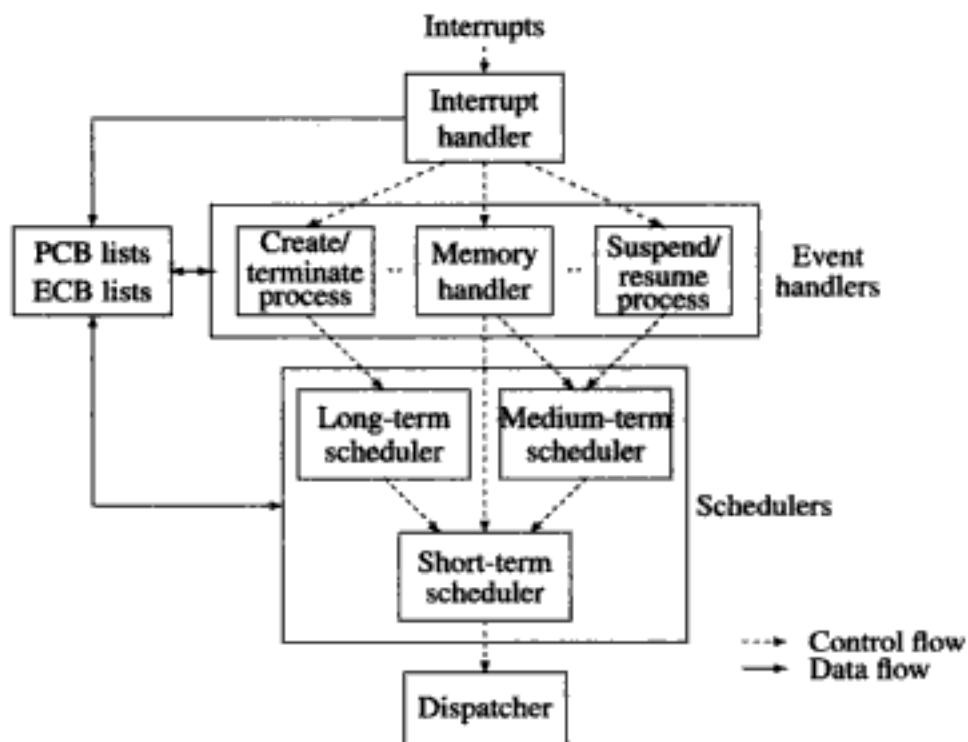


Fig. 4.6 Event handling and scheduling

Figure 4.6 shows an overview of scheduling and related actions. As discussed in Sections 2.1.2 and 3.3.2, the kernel operates in an interrupt-driven manner. Every event that requires the kernel's attention causes an interrupt. The interrupt handler performs a context save function and invokes an event handler. The event handler processes the event and changes the state of the process affected by the event. It then

invokes the long-term, medium-term or short-term scheduler as appropriate. For example, the event handler that creates a new process invokes the long-term scheduler, and the medium-term scheduler is invoked when a process is to be suspended or resumed. Eventually the short-term scheduler gains control and selects a process for execution.

Long-term scheduling The long-term scheduler decides when to start considering a request for medium-term and short-term scheduling. This is called *admission* of a request. The long-term scheduler may defer admission of a request for two reasons. It may not be able to allocate sufficient resources like kernel data structures or I/O devices to a request when it arrives, or it may find that admission of a request would affect system performance in some way. For example, if the system currently contains a large number of CPU-bound requests, it might defer a new CPU-bound request, but might admit a new I/O-bound request straightaway.

Long-term scheduling is not relevant if every arriving request can be admitted for processing straightaway, e.g., in a time sharing OS that can create virtual resources when necessary. However, long-term scheduling was used in 1960's and 1970's for job scheduling because computing equipment was expensive and computer systems had limited resources. It continues to be important in operating systems that have limited resources. Its use is unavoidable if requests have deadlines, or a set of requests are repeated with a known periodicity.

Medium-term scheduling Medium-term scheduling maps the large number of requests that have been admitted to the system into the relatively smaller number of requests that can fit into the memory of the system at any time. Thus it focuses on suspending or reactivating processes by performing swap-out or swap-in actions such that the short-term scheduler would find a sufficient number of *ready* processes. The medium-term scheduler decides when to swap-out a process and when to swap it back into memory, changes the state of the process appropriately and enters its PCB in the appropriate list of PCBs. The actual swapping-in and swapping-out is performed by the memory manager.

The decision to suspend a process is relatively easy. It can be made when a user requests suspension, when the kernel runs out of free memory, or when it finds that some process currently in memory is not likely to be allocated the CPU for a long time. In time-sharing systems, processes in *blocked* or *ready* states are candidates for suspension (see Figure 3.7). A decision to reactivate a process is more involved. The medium-term scheduler has to guess when a process is likely to be dispatched next, and it has to swap-in the process ahead of this time. It can base this guess on the position occupied by a process in a scheduling list. Swap-in and swap-out decisions can be made periodically, or when a related event occurs.

Short-term scheduling Short-term scheduling is concerned with effective use of the CPU. It selects one process from a list of *ready* processes, decides for how long

the process should be executed and arranges to produce a timer interrupt when the time elapses. It now hands over the selected process to the dispatching mechanism.

In summary, the long-term scheduler selects processes that should be considered for scheduling by the medium-term scheduler, the medium-term scheduler selects processes that should be considered for scheduling by the short-term scheduler, and the short-term scheduler selects processes that should be executed on the CPU. Each scheduler applies its own criteria concerning use of resources and quality of service to select processes for the next stage of scheduling. We discuss the nature and relevance of long, medium and short-term scheduling activities in different OSs in the following Sections.

Scheduling in batch processing, multiprogramming and time sharing

In a batch processing system a batch of jobs is submitted for processing, however only one job is under processing at any time. Functioning of the long-term scheduler is trivial because all resources in the system are devoted to one job at any time and jobs are processed in FCFS order—it merely admits the next job of the batch whenever the previous one ends. The short-term scheduler immediately schedules the admitted job. Medium-term scheduling is absent since swapping is not performed. When the job completes, the long-term scheduler admits the next job in the batch.

In a multiprogramming system, admission of a job may have to be delayed due to reasons of system performance and scarcity of resources. The long-term scheduler admits a job when all resources required by it can be allocated and admitting it does not affect the mix of CPU-bound and I/O-bound jobs in the system. Medium-term scheduling is absent since the system does not perform swapping.

Figure 4.7 illustrates the long, medium and short-term scheduling in a time sharing operating system. The long-term scheduler admits a process when kernel resources like control blocks and other resources like I/O devices can be allocated to it. It allocates swap space for the process on a disk, copies the code of the process into the swap space, and adds the process to the list of swapped-out processes.

The medium-term scheduler controls swapping of processes and moves processes between the *swapped-out*, *blocked* and *ready* lists. Whenever the CPU is free, the short-term scheduler selects one process from the *ready* list for execution. The dispatching mechanism initiates or resumes execution of the selected process on the CPU. A process may shuttle between the medium and short-term schedulers many times due to swapping.

From this discussion, it is seen that long-term scheduling is mostly absent in modern operating systems, and medium-term scheduling is typically confined to swapping of processes. We discuss implementation of short-term scheduling in the next few Sections.

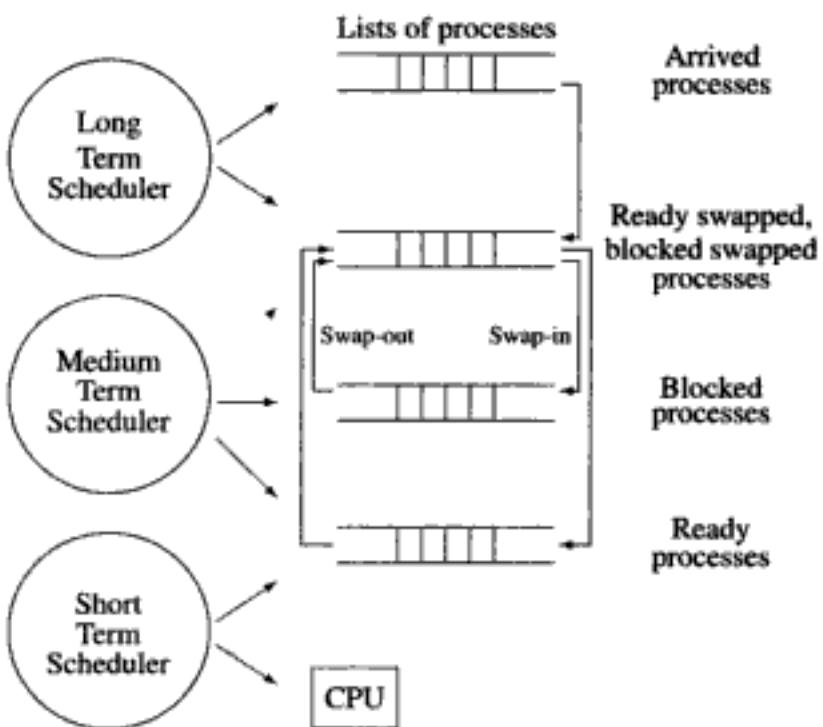


Fig. 4.7 Long, medium and short-term scheduling in a time sharing system

4.4.2 Process Scheduling

The short-term scheduler is often called the *process scheduler*. It uses a list of *ready* processes and decides which process to execute and for how long.

Figure 4.8 shows a schematic of the process scheduler. The list of *ready* processes is maintained as a list of their PCBs. Other PCB lists represent *blocked* and suspended processes. Organization and use of PCB lists depend on the scheduling policy. When the process scheduler receives control, it selects one process for execution. It now invokes the process dispatching mechanism, which loads contents of two PCB fields—the PSW and CPU registers fields—into the CPU to resume execution of the selected process. Thus, the dispatching mechanism interfaces with the policy module on one side and the hardware on the other side.

The context save mechanism is a part of the interrupt handler. It is invoked when an interrupt occurs to save the PSW and CPU registers of the interrupted process. The priority computation and reordering mechanism recomputes the priority of requests and reorders the PCB lists to reflect the new priorities. This mechanism is either invoked explicitly by the policy module when appropriate or it is invoked periodically. Its exact actions depend on the scheduling policy in use.

In the following Sections, we discuss principles and implementation of the process scheduling policies used in practice.

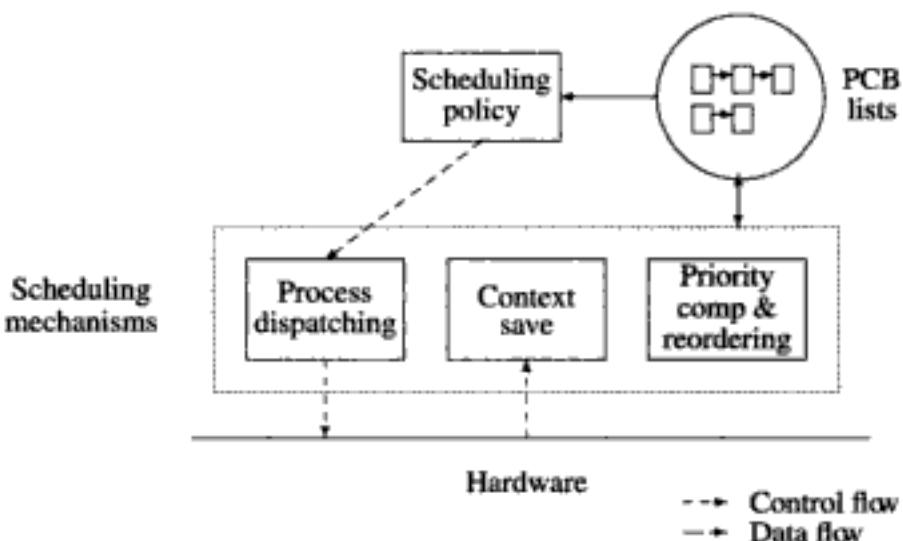


Fig. 4.8 Mechanism and policy modules of process scheduler

4.4.3 Priority-based Scheduling

Priority-based scheduling offers good response times to high priority processes and good throughput. In Section 2.5 we discussed priority-based scheduling used in multiprogramming systems. Its main features are as follows:

1. A mix of CPU-bound and I/O-bound processes exists in the system.
2. An I/O-bound process has a higher priority than a CPU-bound process.
3. Process priorities are static, i.e., they do not change with time.
4. Process scheduling is preemptive; a low priority *running* process is preempted if a higher priority process becomes *ready* (see Example 2.8). In effect, a low priority process cannot be *running* if a higher priority process exists in *ready* state.

We use these features to develop a schematic for priority-based process scheduling.

The scheduler can maintain separate lists of *ready* and *blocked* processes and always select the highest priority process from the *ready* list. However, process priorities are static and scheduling is preemptive. Hence a simpler arrangement can be designed as follows: The scheduler can maintain a single list of PCBs in which PCBs are arranged in the order of reducing priorities. It can scan this list and simply select the first *ready* process it finds. This is the highest priority *ready* process in the system. Table 4.8 summarizes priority-based scheduling.

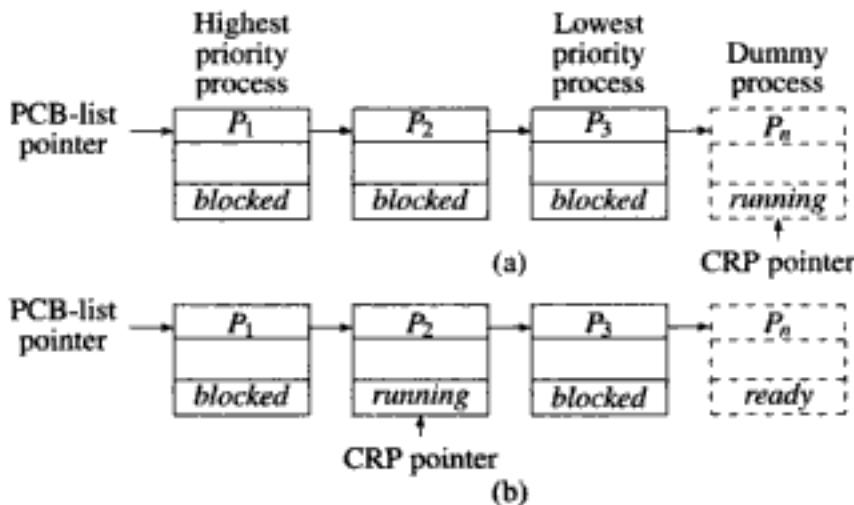
In addition to the PCB list, the scheduler maintains a pointer called *currently running process pointer* (CRP pointer). This pointer points to the PCB of the process that is in the *running* state. When an interrupt occurs, the context save mechanism saves the PSW and CPU registers into appropriate fields of this PCB.

What should the scheduler do if no *ready* processes exist in the system? It should

Table 4.8 Simple priority-based scheduling

1. A single list of PCBs is maintained in the system.
2. PCBs in the list are organized in the order of decreasing priorities.
3. The PCB of a newly created process is entered in the list in accordance with its priority.
4. When a process terminates, its PCB is removed from the list.
5. The scheduler scans the PCB list from the beginning and schedules the first *ready* process it finds.

simply ‘freeze’ the CPU so that the CPU does not execute any instructions, but remains in an interruptible state so that occurrence of an event can be processed by the event handler. If the architecture lacks a ‘freeze’ state for the CPU, the scheduler can achieve an equivalent effect quite simply by defining a dummy process that contains an infinite loop. (This is a system process as against other processes that are user processes.) This process is always in the *ready* state. It is assigned the lowest priority so that it gets scheduled only when no *ready* processes exist in the system. Once scheduled, this process executes until some higher priority process becomes *ready*; and gets preempted when this happens.

**Fig. 4.9** Priority-based scheduling

Example 4.8 Figure 4.9(a) illustrates the situation when all user processes are *blocked*. The only PCB showing a process in the *ready* state is the one for the dummy process. The scheduler selects this process and dispatches it. Figure 4.9(b) shows the situation after process P_2 becomes *ready*. The PCB of P_2 is the first PCB in the PCB list that shows a process in the *ready* state, so P_2 is scheduled and the CRP pointer is set to point at it. (The state of the dummy process would have been changed to *ready* by the interrupt handler before scheduling was performed.)

4.4.4 Round-Robin Scheduling

Round-robin scheduling can be implemented by organizing the list of *ready* processes as a queue. The scheduler always selects the first process in the queue. If the time slice elapses during the execution of a process, the process is put at the end of the queue. A process starting an I/O operation is removed from the *ready* queue. It is added at the end of the queue when the I/O operation completes. A process in the *ready* queue steadily progresses to the head of the queue and gets scheduled.

Round-robin scheduling is best handled by maintaining two lists of PCBs. One list would contain PCBs of *ready* processes, while the other list would contain PCBs of *blocked* and *swapped out* processes. The list of *ready* processes would be organized as a queue. Since *blocked* and *swapped out* processes are ignored for the purpose of scheduling, their PCB list need not be maintained in any specific order. When the state of a process changes, the scheduler shifts the PCB of the process from one PCB list to another. Table 4.9 summarizes round-robin scheduling.

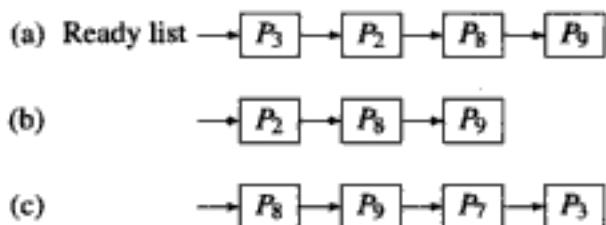
Table 4.9 Round-robin scheduling

1. The scheduler maintains a queue of *ready* processes and a list of *blocked* and *swapped-out* processes.
2. The PCB of a newly created process is added to the end of the *ready* queue. The PCB of a terminating process is removed from the scheduling data structures.
3. The scheduler always selects the PCB at the head of the *ready* queue.
4. When a *running* process finishes its time slice, it is moved to the end of the *ready* queue.
5. The event handler performs following actions:
 - (a) When a process makes an I/O request, or is swapped-out, its PCB is removed from the *ready* queue to the *blocked/swapped-out* list.
 - (b) When the I/O operation awaited by a process finishes, or the process is swapped-in, its PCB is removed from the *blocked/swapped out* list to the end of the *ready* queue.

Example 4.9 Figure 4.10 shows the *ready* queue of a time sharing scheduler at different times during the operation of the system. Figure 4.10(a) shows the *ready* queue when process P_3 is executing. If P_3 initiates an I/O operation, it is shifted to the *blocked* queue. The *ready* queue now looks as shown in Figure 4.10(b). When the I/O operation completes, P_3 is added to the end of the *ready* queue (see Figure 4.10(c)). In the meanwhile process P_2 has completed and a new process, process P_7 , has been created.

4.4.5 Multilevel Scheduling

Efficiency and good user service project conflicting requirements on the scheduler, so the scheduler must trade off these features against each other. Multiprogramming and time sharing represent extreme positions in this tradeoff. Multiprogramming

**Fig. 4.10** Ready lists in a time sharing system

provides high levels of efficiency but cannot guarantee good user service, while time sharing provides good response times, but cannot guarantee high efficiency due to high scheduling overhead. Multilevel scheduling provides a hybrid solution to the problem of providing efficiency and good user service simultaneously.

A multilevel scheduler maintains a number of ready queues. Processes in different queues have different priorities and receive different time slices. High priority processes receive small time slices, so they receive good response times. Low priority processes are expected to keep the CPU busy, thus providing good efficiency, so they receive large time slices. Table 4.10 summarizes features of a multilevel scheduler.

Table 4.10 Multilevel scheduling

1. The scheduler uses many *ready* lists.
2. Each *ready* list has a pair of attributes, viz. (*time slice, scheduling priority*), associated with it. The time slice for a list is inversely proportional to the priority of the list. Each process in a ready list receives the time slice and the scheduling priority associated with the list.
3. Processes in a *ready* list are considered for scheduling only when all higher priority *ready* lists are empty.
4. Round-robin scheduling is performed within each list.

To exploit the features of multilevel scheduling, the scheduler differentiates between highly interactive processes, moderately interactive processes, and non-interactive processes. The scheduler puts the highly interactive processes in the highest priority queue. A small time slice is adequate for these processes, hence they receive very good response times (see Eq. 2.2). Moderately interactive processes could also be put into the same queue, however that would degrade the response times to highly interactive processes and cause high scheduling overhead due to excessive switching between processes, so the scheduler puts moderately interactive processes in *ready* queues with medium priorities where they receive larger time slices. Non-interactive processes are put in the lowest priority *ready* queue. These processes receive the largest time slice. This feature reduces the scheduling overhead in the

system. If more than three *ready* queues are used, other processes occupy intermediate scheduling levels depending on how interactive they are.

Example 4.10 Figure 4.11 illustrates an efficient arrangement of *ready* queues in a multilevel scheduler. A separate queue of *ready* processes is maintained for each priority value. The header of a queue contains a pointer to the PCB of the first process in the queue. It also contains a pointer to the header of the queue for the next lower priority. The scheduler scans the headers in the order of decreasing priority and selects the first process in the first non-empty queue it can find. This way, the scheduling overhead depends on the number of distinct priorities, rather than on the number of *ready* processes. The PCB of a process contains an additional field *scheduling level*. Whenever a blocked process becomes ready the event handler uses this field to identify the *ready* queue to which it should be added.

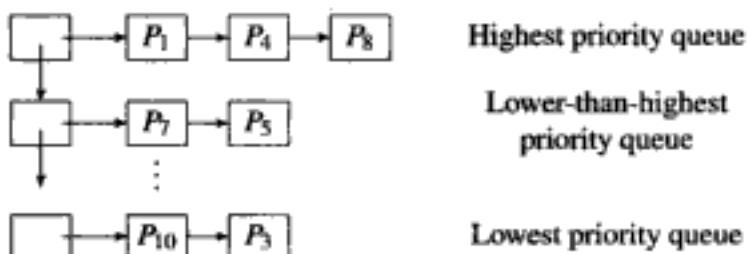


Fig. 4.11 Ready queues in a multi-level scheduler

The multilevel scheduling model inherits some drawbacks of the multiprogramming model. It requires processes to be classified according to their CPU-bound and I/O-bound nature. Further, the classification is static, i.e., a process is classified only once in its lifetime, so a wrong classification would degrade both user service and system performance. The scheduling model also cannot handle a change in the computational or I/O behavior of a process during its lifetime. As seen in Example 4.10, processes in the lowest priority level may starve.

Multilevel adaptive scheduling (also called Multilevel feedback scheduling) A multilevel adaptive scheduler tries to keep a process at the ‘correct’ scheduling level so that it receives an appropriate combination of priority and time slice. The scheduler observes the recent CPU and I/O usage by the process to determine the correct scheduling level for it and moves the process to this level. This way the scheduler adapts to changes in the computational and I/O behavior of processes. Thus, a process can be I/O-bound during one phase of its life and CPU-bound during another phase, and still receive an appropriate priority and time slice. This feature eliminates the problems associated with a priori assignment of static priorities and safeguards both response times and CPU efficiency.

CTSS The *compatible time sharing system* for the IBM 7094 was one of the first time sharing systems. It used multilevel adaptive scheduling with an eight-level pri-

ority structure. The priority levels were numbered 0 through 7. Level numbered n had a time slice of 0.5×2^n CPU seconds associated with it. At initiation, each user process was placed at level 2 or 3 depending on its memory requirement. The promotion/demotion policy was as follows: If a process completely used up the time slice at its current scheduling level (i.e., it did not initiate an I/O operation), it was demoted to the next higher numbered level; otherwise, it remained at the same level. A process was promoted to the next lower numbered level if it spent more than a minute in *ready* state in its current level without obtaining any CPU service. Further, any process performing I/O on the user terminal was promoted to level 2. Subsequently, it would migrate to the 'correct' place for it in the priority structure through possible demotions.

Example 4.11 Figure 4.12 shows promotions and demotions in the CTSS-type multi-level adaptive scheduling. Figure 4.12(a) shows process P_3 in execution, being at the top of the highest priority *ready* queue. Processes P_5 , P_8 and P_7 exist in lower priority queues. Process P_3 completely uses up its time slice, hence it is demoted to level 2. A little later process P_7 is promoted to level 2 because it has spent more than a minute in level 3. Figure 4.12(b) shows the resulting *ready* queues.

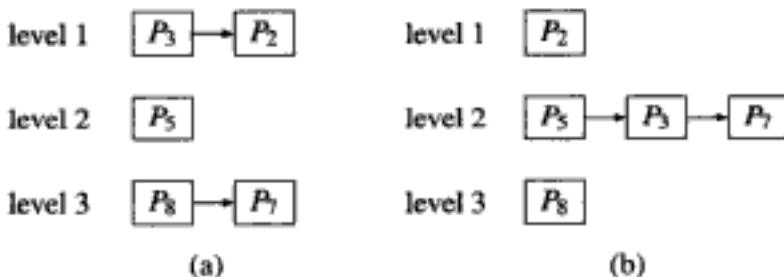


Fig. 4.12 Example of multi-level adaptive scheduling

4.4.6 Fair Share Scheduling

A common criticism of all scheduling policies discussed so far is that they are oblivious to the relationships between processes and users. They assume that each process belongs to a different user and try to provide equitable service to all processes. If applications initiated by users create different number of processes, an application employing more processes is likely to receive more CPU attention than an application employing fewer processes.

The notion of a *fair share* addresses this issue. A fair share is the fraction of CPU time that should be devoted to execution of a group of processes that belong to the same user or the same application. The scheduler considers a process from a group for scheduling only if the group has not utilized its fair share of CPU time, and sets the value of the time slice for the process such that the group would not exceed its fair share of CPU time at the end of the time slice. This policy ensures an equitable use of the CPU by processes belonging to different users or different applications.

Note that the actual share of CPU time received by a group of processes may differ from the fair share of the group due to lack of activity in its processes or in processes of other groups. For example, consider two groups G_1 and G_2 of processes, each having a 50 percent share of CPU time. If all processes in G_1 are sleeping, processes of G_2 should be given 100 percent of the available CPU time so that CPU time is not wasted. What should the scheduler do when processes of G_1 become active after a prolonged period of sleep? Should they be given only 50 percent of CPU time after they wake up, because that is their fair share of CPU time, or should the scheduler give them almost all the available CPU time until their actual CPU consumption since inception becomes 50 percent? Lottery scheduling, which we describe in the following, and the scheduling policy used in the Unix operating system (see Section 4.6) differ in the way they handle this situation.

Lottery scheduling is a novel technique proposed for sharing a resource in a probabilistically fair manner. Lottery "tickets" are distributed to all processes (or applications) sharing a resource in such a manner that a process gets as many tickets as its fair share of the resource. For example, a process would be given five tickets out of a total of 100 tickets if its fair share of the resource is five percent. When the resource is to be allocated, a lottery is held to randomly select a winning ticket from the tickets held by processes that actively seek the resource. The process holding the winning ticket is then allocated the resource. The actual share of the resources allocated to the process depends on contention for the resource. Lottery scheduling can be used for fair share CPU scheduling as follows: Tickets can be issued to applications (or users) on the basis of their fair share of CPU time. An application can share its tickets among its processes in any manner it desires. To allocate a CPU time slice, the scheduler holds a lottery in which only tickets of *ready* processes participate. When the time slice is a few milliseconds, this scheduling method provides fairness even over fractions of a second if all groups of processes are active.

4.4.7 Power Management

In Section 4.4.3, we saw how the kernel puts the CPU into an infinite loop that does nothing when no *ready* processes exist. This solution wastes power in executing useless instructions. In power starved systems such as embedded and mobile systems, it is essential to prevent this wastage of power.

To address this requirement, computers provide a special mode in the CPU. When in this mode, the CPU does not execute instructions, which conserves power; however, the CPU can accept interrupts and resume normal operation when an interrupt arises. We will use the term *sleep mode* of the CPU generically for such modes.

Some computers provide several sleep modes. In the 'light' sleep mode, the CPU simply stops executing instructions. In a 'heavy' sleep mode, the CPU not only stops executing instructions, but also takes other steps that reduce its power consumption, e.g., slowing the clock and disconnecting the CPU from the system bus. Ideally, the kernel should put the CPU into the deepest sleep mode possible when the system does

not have processes in the *ready* state. However, a CPU takes a longer time to ‘wake up’ from a heavy sleep mode than it would from a light sleep mode, so the kernel has to make a trade-off here. It starts by putting the CPU in the light sleep mode. If no processes become *ready* for some time, it puts the CPU into a heavier sleep mode, and so on. This way, it conserves only as much power as possible without sacrificing response times too much.

Operating systems like Unix and Windows have generalized power management to include all devices. Typically, a device is put into a lower power consuming state if it has been dormant at its present power consuming state for some time. Users are also provided with utilities through which they can configure the power management scheme used by the OS.

4.5 REAL TIME SCHEDULING

Real time applications often impose some special scheduling constraints, in addition to the familiar need to meet deadlines. First, the processes within a real time application are interacting processes and may have priorities among them that are determined by the nature of the application; second, the processes may be periodic. Example 4.12 illustrates these constraints.

Example 4.12 Consider the real time data logging application of Example 3.1. The *copy_sample* and *disk.write* processes interact to use the buffer. Data samples arrive at 100 microseconds intervals; it is important that a sample should be copied out before the next sample arrives. To achieve this, *copy_sample* is given a higher priority than other processes of the application. This process must also run with a periodicity of 100 microseconds.

None of the scheduling policies discussed so far take deadlines or such constraints into account, so a real time operating system needs special scheduling policies.

4.5.1 Process Precedences and Feasible Schedules

Processes of a real time application interact among themselves to ensure that they perform their actions in a desired order. We make the simplifying assumption that such interaction takes place only at the start or end of a process. It causes dependences between processes, which must be taken into account while determining deadlines and while scheduling. We use a *process precedence graph* (PPG) to depict such dependences between processes.

Process P_i is said to *precede* process P_j if execution of P_i must be completed before P_j can begin its execution. The notation $P_i \rightarrow P_j$ shall indicate that process P_i directly precedes process P_j . The precedence relation is transitive, i.e., $P_i \rightarrow P_j$ and $P_j \rightarrow P_k$ implies that P_i precedes P_k . The notation $P_i \xrightarrow{*} P_k$ is used to indicate that process P_i directly or indirectly precedes P_k .

A *process precedence graph* is a directed graph $G \equiv (N, E)$ such that $P_i \in N$ represents a process, and an edge $(P_i, P_j) \in E$ implies $P_i \rightarrow P_j$. Thus, a path $P_i \dots P_k$ in PPG implies $P_i \xrightarrow{*} P_k$. A process P_k is a descendant of P_i if $P_i \xrightarrow{*} P_k$.

In Section 2.7, we defined a *hard real time system* as one that can meet the response time requirement of a real time application in a guaranteed manner, even when fault tolerance actions are required. This condition implies that the time required by the OS to complete operation of all processes in the application does not exceed the response requirement of the application. On the other hand, a *soft real time system* meets the response requirement of an application only in a probabilistic manner, and not necessarily at all times. The notion of a *feasible schedule* helps to differentiate between these situations.

Definition 4.1 (Feasible schedule) A feasible schedule for processes of an application is a sequence of scheduling decisions that enables the processes to operate in accordance with their precedences and meet the response requirement of the application.

Real time scheduling focuses on finding and implementing a feasible schedule for an application, if one exists. Consider an application for updating airline departure information on displays at 15 second intervals. It consists of the following independent processes, where process P_5 handles an exceptional situation that seldom occurs.

Process	P_1	P_2	P_3	P_4	P_5
Service time	3	3	2	4	5

A feasible schedule does not exist for completing all five processes in 15 seconds, so a deadline overrun would occur. However, several schedules are possible when process P_5 is not active. The scheduler in a soft real time system can find any one of them and use it.

Table 4.11 summarizes three main approaches to real time scheduling. We discuss the features and properties of these scheduling approaches in the following.

Static scheduling As the name indicates, a schedule is prepared before the system is put into operation. The schedule considers process precedences, periodicities, resource constraints and possibilities of overlapping I/O operations in one process with computations in another process. This schedule is represented in the form of a table whose rows indicate when execution of different processes should begin. No scheduling decisions are made during operation of the system. The real time OS simply consults the table and starts execution of processes as indicated in it.

The size of the scheduling table will depend on the periodicity of processes. If processes are aperiodic, or if all processes have the same periodicity, the scheduling table will have only as many rows as the number of processes in the application. This schedule is used repeatedly during operation of the system. If periodicities of processes are different, the length of the schedule that needs to be represented in the

Table 4.11 Real time scheduling

Static scheduling	A schedule is prepared <i>before</i> execution of the application begins. Process interactions, periodicities, resource constraints and deadlines are considered while forming the schedule.
Priority-based scheduling	The real time application is analysed to assign appropriate priorities to processes in it. Conventional priority-based scheduling is used during operation of the application.
Dynamic scheduling	Scheduling is performed when a request to create a process is executed. Process creation succeeds only if response requirements of the process can be satisfied in a guaranteed manner.

scheduling table will be the least common multiple of periodicities of all processes in the application.

Static scheduling leads to negligible scheduling overhead during system operation. However, it is inflexible and cannot handle issues like fault tolerance.

Priority-based scheduling A system analyst uses two considerations while assigning priorities to processes: criticality of processes as discussed in Example 4.12, and periodicities of process. A process with smaller periodicity needs to execute more often than a process with a larger periodicity. Hence the priority of a process should depend on the inverse of its periodicity.

During system operation, process priorities are used to perform scheduling decisions. Priority-based scheduling provides more flexibility than static scheduling; however, it incurs scheduling overhead during system operation. Priority based scheduling can handle some kinds of failures naturally, because high priority processes will continue to get favored treatment so long as the needed resources are available.

Dynamic scheduling In systems using the dynamic scheduling approach, scheduling is performed during the system's operation. Typically, a scheduling decision is performed when a process arrives. Multi-media systems like video on demand use a dynamic scheduling approach. A request to initiate a process contains information such as the process's resource requirement, service time and a deadline or a specification of service quality. On receiving a process initiation request, the scheduler checks to see whether it is possible to assign the resources needed by the process and ensure its deadline or expected quality of service. It accepts the process only if these checks succeed.

Another approach to dynamic scheduling is to optimistically admit processes for execution. In this approach, there is no guarantee that the deadline or service quality desired by a process can be met. Soft real time systems often follow this approach.

4.5.2 Deadline Scheduling

Two kinds of deadlines can be specified for each process: a starting deadline, or the latest instant of time by which execution of the process must start; and a completion deadline, or the time by which execution of the process must complete.

Deadline estimation An in-depth analysis of a real time application and its response requirements is carried out during its development. Deadlines for individual processes can be determined by considering process precedences and working backwards from the response requirement of the application. Accordingly, the deadline of a process P_i is

$$D_i = D_{application} - \sum_{k \in \text{descendant}(i)} x_k \quad (4.3)$$

where $D_{application}$ is the deadline of the application, x_k is the service time of process P_k , and $\text{descendant}(i)$ is the set of descendants of P_i in the PPG, i.e., the set of all processes that lie on some path between P_i and the exit node of the PPG. Thus, the deadline for a process P_i is such that if it is met, all processes that directly or indirectly depend on P_i can also finish by the overall deadline of the application. This method is illustrated in Example 4.13.

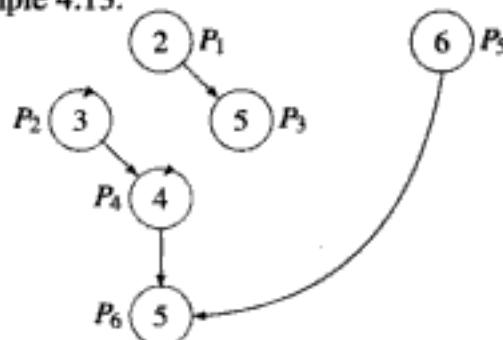


Fig. 4.13 PPG for a simple real time system

Example 4.13 Figure 4.13 shows the PPG of a real time application containing 6 processes. Each circle is a node of the graph and represents a process. The number in a circle indicates the service time of a process. An edge in the PPG shows a precedence constraint. Thus, process P_2 can be initiated only after process P_1 completes, and process P_4 can be initiated only after processes P_2 and P_3 complete, etc. We assume that each process is CPU-bound and is executed in a non-preemptive manner. The total of the service times of the processes is 25 seconds. If the application has to produce a response in 25 seconds, the deadlines of the processes would be as follows:

Process	P_1	P_2	P_3	P_4	P_5	P_6
Deadline	8	16	16	20	20	25

A practical method of estimating deadlines will have to incorporate several other constraints as well. For example, processes may perform I/O. If an I/O operation

of one process can be overlapped with execution of some independent process, the deadline of its predecessors (and ancestors) in the PPG can be relaxed by the amount of I/O overlap. For example, processes P_2 and P_3 in Figure 4.13 are independent of one another. If the service time of P_2 includes 1 second of I/O time, the deadline of P_1 can be made 9 seconds instead of 8 seconds if the I/O operation of P_2 can overlap with P_3 's processing. However, overlapped execution of processes must consider resource availability as well. Hence determination of deadlines is far more complex than described here.

Earliest deadline first (EDF) scheduling As its name suggests, this policy always selects the process with the earliest deadline. Consider a set of real time processes that do not perform I/O operations. If seq is the sequence in which processes are serviced by a deadline scheduling policy and $pos(P_i)$ is the position of process P_i in seq , a deadline overrun does not occur for process P_i only if the sum of its own service time and service times of all processes that precede it in seq does not exceed its own deadline, i.e.

$$\sum_{k: pos(P_k) \leq pos(P_i)} x_k \leq D_i \quad (4.4)$$

where x_k is the service time of process P_k , and D_i is the deadline of process P_i . If this condition is not satisfied, a deadline overrun will result for process P_i .

Table 4.12 Operation of Earliest Deadline First scheduling

Time	Process completed	Deadline overrun	Processes in system	Process scheduled
0	—	0	$P_1 : 8, P_2 : 16, P_3 : 16, P_4 : 20, P_5 : 20, P_6 : 25$	P_1
2	P_1	0	$P_2 : 16, P_3 : 16, P_4 : 20, P_5 : 20, P_6 : 25$	P_2
5	P_2	0	$P_3 : 16, P_4 : 20, P_5 : 20, P_6 : 25$	P_3
10	P_3	0	$P_4 : 20, P_5 : 20, P_6 : 25$	P_4
14	P_4	0	$P_5 : 20, P_6 : 25$	P_5
20	P_5	0	$P_6 : 25$	P_6
25	P_2	0	—	—

When a feasible schedule exists, it can be shown that Condition 4.4 holds for all processes, i.e., a deadline overrun will not occur for any process. Table 4.12 illustrates operation of the EDF policy for the deadlines of Example 4.13. The notation $P_4 : 20$ in the column *processes in system* indicates that process P_4 has the deadline 20. Processes P_2, P_3 and P_5, P_6 have identical deadlines, so three schedules other than the one shown in Table 4.12 are possible using EDF scheduling; however, none of them would incur deadline overruns.

The primary advantages of EDF scheduling are its simplicity and non-preemptive nature, which reduces the scheduling overhead. EDF scheduling is a good policy for

static scheduling because existence of a feasible schedule can be determined a priori, and absence of scheduling actions during operation implies that 100 percent CPU utilization can be achieved if processes do not perform I/O operations or if their I/O operations can be overlapped with computations of other processes. It is also a good dynamic scheduling policy for use in soft real time systems where process information may not be available in advance, because occasional deadline overruns are acceptable.

However, EDF scheduling does not consider service times of processes while preparing a schedule, hence it may not perform well when a feasible schedule does not exist. The next example illustrates this aspect of EDF scheduling.

Example 4.14 Consider the PPG of Figure 4.13 with the edge (P_5, P_6) removed. It contains two independent applications, one contains the processes P_1, P_4 and P_6 , while the other contains P_3 alone. If all processes are to complete by 19 seconds, a feasible schedule does not exist. Now deadlines of the processes determined using Eq. (4.3) are as follows:

Process	P_1	P_2	P_3	P_4	P_5	P_6
Deadline	2	10	10	14	19	19

EDF scheduling may schedule the processes either in the sequence $P_1, P_2, P_3, P_4, P_5, P_6$, which is the same as in Table 4.12, or in the sequence $P_1, P_2, P_3, P_4, P_6, P_5$. Processes P_5 and P_6 miss their deadlines in the first sequence, whereas only process P_5 misses its deadline in the second sequence. However, we have no control over which sequence will be chosen by an implementation of EDF scheduling. Thus the number of processes that may suffer deadline overruns in a soft real time system may not be optimal.

4.5.3 Rate Monotonic Scheduling

When processes in an application are periodic but have different periodicities, the existence of a feasible schedule can be determined in an interesting way. Consider three independent processes that do not perform I/O operations:

Process	P_1	P_2	P_3
Time period (msec)	10	15	30
Service time (msec)	3	5	9

Process P_1 repeats every 10 msec and needs 3 msec of CPU time. So the fraction of the CPU's time that it uses is $3/10$, i.e., 0.30. The fractions of CPU time used by P_2 and P_3 are analogously $5/15$ and $9/30$, i.e., 0.33 and 0.30. They add up to 0.93, so if the CPU overhead of OS operation is 0, it is feasible to service these three processes when they operate with these periodicities. We denote the service time of process P_i as x_i , and use the notation tp_i for the time period of P_i . In general, a set

of periodic processes $P_1..P_n$ can be serviced by a hard real time system that has a negligible overhead if

$$\sum_{i=1 \dots n} \frac{x_i}{tp_i} \leq 1.$$

We still have to schedule these processes so that they can all execute with the proper periodicities. The *rate monotonic* (RM) scheduling policy does it as follows: It determines the *rate* at which a process has to repeat, i.e., the number of repetitions per second, and assigns the rate itself as the priority of the process. This way, a process with a higher periodicity has a higher priority, which would enable it to operate more frequently. It now employs a priority-based scheduling technique to perform scheduling.

In the above example, priorities of processes P_1, P_2 and P_3 would be $1/0.010$, $1/0.015$ and $1/0.025$, i.e., 100, 67 and 45, respectively. Figure 4.14 shows how these processes would operate. Process P_1 would be scheduled first. It would execute once and become dormant after 3 msec, because $x_1 = 3$ msec. Now P_2 would be scheduled and would complete after 5 msec. P_3 would be scheduled now, but it would be preempted after 2 msec because P_1 becomes *ready* for the second time, and so on. As shown in Figure 4.14, process P_3 would complete at 28 msec. By this time, P_1 has executed three times and P_2 has executed two times.

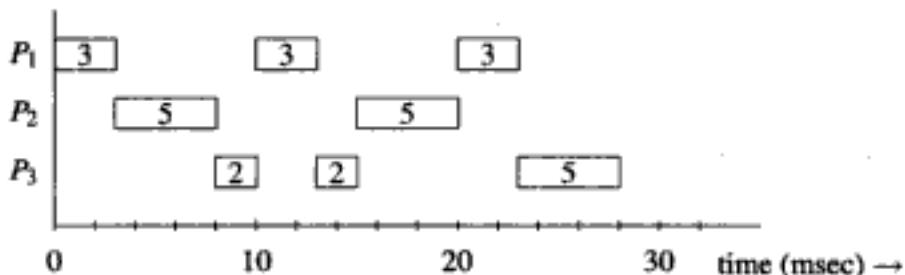


Fig. 4.14 Operation of real time processes using rate monotonic scheduling

Rate monotonic scheduling is not guaranteed to find a feasible schedule in all situations. For example, if process P_3 had a time period of 27 msec, its priority would be different; however, relative priorities of the processes would be unchanged. So now P_3 would suffer a deadline overrun of 1 msec. A feasible schedule would have been obtained if P_3 had been scheduled at 20 msec and P_1 at 25 msec, however this is not possible under RM scheduling because it assigns static priorities. Liu and Layland (1973) have shown that deadline overruns do not arise in RM scheduling if the fraction of CPU time used by real time processes does not exceed $m(2^{1/m} - 1)$, where m is the number of processes. Recall that EDF scheduling would avoid deadline overruns so long as a feasible schedule exists for the processes under consideration.

4.6 SCHEDULING IN UNIX

Process states and state transitions used in Unix have been described in Section 3.5.1. A Unix process is in one of three modes at any moment:

1. Kernel non-interruptible mode
2. Kernel interruptible mode
3. User mode.

Kernel non-interruptible mode processes enjoy the highest priorities, while user mode processes have the lowest priorities. The logic behind this priority structure is as follows: A process in the kernel mode has many OS resources allocated to it. If it is allowed to run at a high priority, it will release these resources sooner. Processes in the kernel mode that do not hold many OS resources are put in the kernel interruptible mode.

Event addresses A process that is blocked on an event is said to *sleep* on it. Unix uses an interesting arrangement to activate processes sleeping on an event. Instead of ECBs (see Section 3.3.6), it uses the notion of an *event address*. A set of addresses is reserved in the kernel for this purpose. Every event is mapped into one of these addresses. When a process wishes to sleep on an event, address of the event is computed using a special procedure. The state of the process is changed to *blocked* and the address of the event is put in its process structure. This address serves as the description of the event awaited by the process. When an event occurs, the kernel computes its event address and activates all processes sleeping on it.

This arrangement has one drawback—it incurs unnecessary overhead in some situations. Several processes may sleep on the same event. The kernel activates all of them when the event occurs. The processes must themselves decide whether all of them should resume their execution or only some of them should. For example, when several processes sleeping due to data access synchronization are activated, only one process should gain access to the data and other activated processes should go back to sleep. The method of mapping events into addresses adds to this problem. A hashing scheme is used for mapping, hence two or more events may map into the same event address. Now occurrence of any one of these events would activate all processes sleeping on all these events. Only some processes sleeping on the correct event should resume their execution. All other processes should go back to sleep.

Process priorities Unix is a pure time sharing operating system, so it uses the round-robin scheduling policy. However, in a departure from pure round-robin, it dynamically varies the priorities of processes and performs round-robin only for processes with identical priority. Thus, in essence, it uses multilevel adaptive scheduling. The reasons for dynamic variation of process priorities are explained in the following.

A user process executes in the user mode when it executes its own code. It enters the kernel mode when it starts executing the kernel code due to an interrupt

or a system call. A process executing a system call may block for a resource or an event. Such a process may hold some kernel resources; when it becomes active again it should be scheduled as soon as possible so that it can release kernel resources and return to the user mode. To facilitate this, all processes in the kernel mode are assigned higher priorities than all processes in the user mode. When a process gets blocked in the kernel mode, the cause of blocking is used to determine what priority it should have when activated again.

Unix processes are allocated numerical priorities, where a larger numerical value implies a lower effective priority. In Unix 4.3 BSD, the priorities are in the range 0 to 127. Processes in the user mode have priorities between 50 and 127, while those in the kernel mode have priorities between 0 and 49. When a process gets blocked in a system call, its priority is changed to a value in the range 0–49 depending on the cause of blocking. When it becomes active again, it executes the remainder of the system call with this priority. When it exits the kernel mode, its priority reverts to its previous value, which was in the range 50–127.

The second reason for varying process priorities is that the round-robin policy does not provide fair CPU attention to all processes. An I/O-bound process does not fully utilize a time slice hence its CPU usage always lags behind the CPU usage of CPU-bound processes. The Unix scheduler provides a higher priority to processes that have not received much CPU attention in the recent past. It recomputes process priorities every second according to the following formula:

$$\begin{aligned}\text{Process priority} &= \text{base priority for user processes} \\ &\quad + f(\text{CPU time used});\end{aligned}$$

where all user processes have the same base priority.

The scheduler maintains the CPU time used by each process in its process table entry. The real time clock raises an interrupt 60 times a second, i.e., once every 16.67 msec. The clock handler increments the count in the CPU usage field of the running process. The CPU usage field is 0 to start with, hence a process starts with the base priority for user processes. As it receives CPU attention, its effective priority reduces. If two processes have equal priorities, they are scheduled in a round-robin manner.

If the total CPU usage of a process is used in computing process priority, the scheduling policy would resemble the LCN policy, which does not perform well in practice (see Section 4.1.1). Therefore, Unix uses only the recent CPU usage rather than the total CPU usage since the start of a process. To implement this policy, the scheduler applies a ‘decay’ to the influence of the CPU time used by a process. Every second, before recomputing process priorities, values in the CPU usage fields of all processes are divided by 2 and stored back. The new value in the CPU usage field of a process is used as $f(\text{CPU time used})$. This way influence of the CPU time used by a process reduces as the process waits for CPU.

An interesting feature in Unix is that a user can control the priority of a process. This is achieved through the system call `nice(<priority value>)`; which sets the *nice value* of a user process. The effective priority of the process is now computed as follows:

$$\begin{aligned}\text{Process priority} &= \text{base priority for user processes} \\ &\quad + f(\text{CPU time used}) + \text{nice value};\end{aligned}\quad (4.5)$$

A user is required to use a zero or positive value in the `nice` call. Thus, a user cannot increase the effective priority of a process, but can lower it. (This is typically done when a process is known to have entered a CPU-bound phase.)

Example 4.15 Table 4.13 summarizes operation of a Unix-like scheduling policy for the processes in Table 4.2. It is assumed that process P_3 is an I/O bound process, which initiates an I/O operation lasting 0.5 seconds after executing on the CPU for 0.1 seconds, and none of the other processes perform I/O. The T field indicates the CPU time consumed by a process and the P field contains its priority. The scheduler updates the T field of a process 60 times a second and recomputes process priorities once every second. The time slice is 1 second, and the base priority of user processes is 60. The first line of Table 4.13 shows that at $t = 0$, only P_1 exists in the system. Its T field contains 0, hence its priority is 60. Two lines are shown for $t = 1.0$. The first line shows the T fields of processes at $t = 1$, while the second line shows the P and T fields after the priority computation actions at $t = 1$. At the end of the time slice, contents of the T field of P_1 are 60. The decaying action of dividing the CPU time by 2 reduces it to 30, hence the priority of P_1 becomes 90. At $t = 2$, the effective priority of P_1 is smaller than that of P_2 because their T fields contain 45 and 0, respectively, hence P_2 is scheduled. Similarly P_3 is scheduled at $t = 2$ seconds.

Process P_3 uses the CPU for only 0.1 seconds before starting an I/O operation, so it has a higher priority than P_2 when scheduling is performed at $t = 4$ seconds and it gets scheduled ahead of process P_2 . It again gets scheduled at $t = 6$ seconds. These scheduling actions correct the bias against I/O-bound processes exhibited by the RR scheduler.

4.6.1 Fair Share Scheduling in Unix

Unix schedulers achieve fair share scheduling by adding one more term to Eq. (4.5) as follows:

$$\begin{aligned}\text{Process priority} &= \text{base priority for user processes} \\ &\quad + f(\text{CPU time used by process}) \\ &\quad + f(\text{CPU time used by process group}) + \text{nice value};\end{aligned}$$

where f is the same function used in Eq. (4.5). Now the OS keeps track of the CPU time utilized by each process and by each group of processes. The priority of a process depends on the recent CPU time used by it and by other processes in the

Table 4.13 Operation of a Unix-like scheduling policy when processes perform I/O

Time	P ₁		P ₂		P ₃		P ₄		P ₅		Sche- duled
	P	T	P	T	P	T	P	T	P	T	
0.0	60	0									
1.0		60									P ₁
	90	30									P ₁
2.0	90	0									P ₂
	105	45	60	0							P ₂
3.0	45	60			0						P ₃
	82	22	90	30	60	0					P ₃
3.1	82	22	90	30	60	6					P ₁
4.0	76	30			6						
	98	38	75	15	63	3					P ₃
4.1	98	38	75	15	63	9					P ₂
5.0	38	69			9		0				
	79	19	94	34	64	4	60	0			P ₄
6.0	19	34			4		60	60			P ₃
	69	9	77	17	62	2	90	30			

group. Since the effective priority is the inverse of the priority value computed by the above equation, the priority of a process reduces even when other processes of its group receive CPU time. Example 4.16 illustrates operation of a fair share scheduler.

Example 4.16 Table 4.14 depicts scheduling of the processes of Table 4.2 by a Unix-like fair share scheduler. Fields *P*, *T* and *G* contain process priority, CPU time consumed by a process and CPU time consumed by a group of processes, respectively. Two process groups exist. The first group contains processes *P*₁, *P*₂, *P*₃ and *P*₅, while the second group contains process *P*₄ all by itself. As expected, process *P*₄ receives a favored treatment when compared to other processes. In fact, in the period *t* = 5 to *t* = 15, it receives every alternate time slice. Processes *P*₂, *P*₃ and *P*₅ suffer because they belong to the same process group. These facts are reflected in the turn around times and weighted turn arounds of the processes, which are as shown in Table 4.15.

4.7 SCHEDULING IN LINUX

Linux supports both real time and non-real time applications. Accordingly, it has two classes of processes. The real time processes have static priorities between 0 and 100, where 0 is the highest priority. Real time processes can be scheduled in two ways: FIFO or round-robin within each priority level. The kernel associates a flag with each process to indicate how it should be scheduled.

Non-real time processes have lower priorities than all real time processes; their priorities are dynamic and have numerical values between 20 and 19, where 20 is the highest priority. Effectively, the kernel has (100 + 40) priority levels. To start

Table 4.14 Operation of fair share scheduling

Time	P_1			P_2			P_3			P_4			P_5			Scheduled
	P	C	G	P	C	G	P	C	G	P	C	G	P	C	G	
0	60	0	0			0			0			0			0	P_1
1	120	30	30			30			30			0			30	P_1
2	150	45	45	105	0	45			45			0			45	P_2
3	134	22	52	142	30	52	112	0	52			0			52	P_3
4	127	11	56	131	15	56	146	30	56			0			56	P_1
5		58		125	7	58	133	15	58	60	0	0			58	P_4
6		29		92	3	29	96	7	29	120	30	30			29	P_2
7		44		135	31	44	107	3	44	90	15	15			44	P_4
8		22		97	15	22	83	1	22	134	37	37	82	0	22	P_5
9		41		108	7	41	101	0	41	96	18	18	131	30	41	P_4
10		20		83	3	20	80	0	20	138	39	39	95	15	20	P_3
11		40		101	1	40			40	98	19	19	107	7	40	P_4
12		20		80	0	20			20	138	39	39	83	3	20	P_2
13		40			40				40	98	19	19	101	1	40	P_4
14		20			20				20			39	80	0	20	P_5
15		40			40				40			19	130	30	40	P_5
16																

Table 4.15 Performance of fair share scheduling

Process	P_1	P_2	P_3	P_4	P_5
Completion time	5	13	11	14	16
Turn around time	5	11	8	9	7
Weighted turn around	1.67	3.67	4.00	1.80	2.33

Mean turn around time (\bar{ta}) = 8.0 seconds

Mean weighted turn around (\bar{w}) = 2.69

with, each non-real time process has the priority 0. The priority can be varied by the process itself through the *nice* or *setpriority* system calls. However, special privileges are needed to increase the priority through the *nice* system call, so processes typically use this call to lower their priorities when they wish to be nice to other processes. In addition to such priority variation, the kernel varies the priority of a process to reflect its I/O-bound or CPU-bound nature. To implement this, the kernel maintains information about how much CPU time the process has used recently and for how long it was in the *blocked* state, and adds a bonus between 5 and -5 to the nice value of the process. Thus, a highly interactive process would have an effective priority of nice -5, while a CPU-bound process would have an effective priority of nice+5.

Due to the multilevel priority structure, the linux kernel organizes its scheduling

data as shown in Figure 4.11 of Section 4.4.5. Thus, each priority level has a list of ready processes associated with it. To limit the scheduling overhead, Linux uses a scheduler schematic analogous to Figure 3.10. Thus, scheduling is not performed after every event handling action. It is performed when the currently executing process has to block due to a system call, or when the `need_resched` flag has been set by an event handling action. This is done while handling expiry of the time slice, or while handling an event that activates a higher priority process than the currently executing one.

Non-real time processes are scheduled using the notion of a time slice; however, the Linux notion of time slice is different from the notion described in earlier chapters. The Linux time slice is actually a time quantum that a process can use over a period of time in accordance with its priority. Linux uses time slices in the range 10 msec to 200 msec. Contrary to the description in earlier chapters, a high priority process has a larger time slice. This does not affect response times in Linux because, as described above, high priority processes are interactive in nature.

The Linux scheduler uses two lists of processes, an *active list* and an *exhausted list*. Both lists are ordered by priorities of processes and use the data structure described earlier. The scheduler schedules a process from the active list, which uses time from its time slice. When its time slice is exhausted, it is put into the exhausted list. Schedulers in Linux kernel 2.5 and earlier kernels executed a priority recomputation loop when the active list became empty. The loop computed a new time slice for each process based on its dynamic priority. At the end of the loop, all processes were transferred to the active list and normal scheduling operation was resumed.

The Linux 2.6 kernel uses a new scheduler that incurs less overhead and scales better with the number of processes and CPUs. The scheduler spreads the priority recomputation overhead throughout the scheduler's operation, rather than lumping it in the recomputation loop. It achieves this by recomputing the priority of a process when the process exhausts its time slice and gets moved to the exhausted list. When the active list becomes empty, the scheduler merely interchanges the active and exhausted lists.

The scheduler scalability is ensured in two ways. The scheduler has a bit flag to indicate whether the list of processes for a priority level is empty. When invoked, the scheduler tests the flags of the process lists in the order of reducing priority, and selects the first process in the first occupied process list it finds. This procedure incurs a scheduling overhead that does not depend on the number of ready processes; it depends only on the number of scheduling levels, hence it is bound by a constant. This scheduling is called $O(1)$, i.e., order 1, scheduling. Schedulers in older Linux kernels used a synchronization lock on the active list of processes to avoid race conditions when many CPUs were supported. The Linux 2.6 kernel maintains active lists on a per-CPU basis, which eliminates the synchronization lock and associated delays. This arrangement also ensures that a process is executed by the same CPU every time it is scheduled; it helps to ensure better cache hit ratios.

4.8 SCHEDULING IN WINDOWS

Windows scheduling aims at providing good response times to real time and interactive threads. Scheduling is priority-driven and preemptive. Scheduling within a priority level is performed using a round-robin policy with time slicing. Priorities of non-real time threads are dynamically varied to favor interactive threads. This aspect is analogous to multilevel adaptive scheduling (see Section 4.4.5).

Threads are divided into two classes—real time threads and other threads. Real time threads are given priorities in the range 16–31. These priorities remain unchanged during operation of the thread. Other threads have priorities in the range 1–15. Their priorities can vary during their lifetime, hence this class of threads is also called the variable priority class. The effective priority of a thread in this class at any moment is a combination of three factors—the base priority of the process to which the thread belongs, the base priority of the thread and the dynamic component assigned by the scheduler. A higher numerical value of priority implies higher effective priority.

The base priority of processes and threads are specified at their creation. The base priority of a thread is in the range –2 to 2. The initial priority of a thread is the sum of the base priority of the process in which it is created and its own base priority. The scheduler varies the priority of a thread dynamically. The effective priority of a thread can vary between (base priority of process –2) and 15. Thus, it can reach the highest priority value for the variable priority class, however it cannot dip more than 2 levels below the base priority of the process to which it belongs. Priority variation is implemented as follows: If a thread uses up its time slice during execution, its priority is reduced by 1 unit. When a thread that is blocked on an event wakes up, it is given a priority increase based on the nature of the event. If the thread was blocked on input from the keyboard, its priority is boosted by 6. This policy favors interactive threads over other threads.

The system guards against starvation by temporarily raising the priority of a ready thread that has not received CPU for more than 3 seconds. The priority of such a thread is raised to 15 and it is given a CPU burst that is twice its normal burst. After this burst, its priority and CPU burst revert back to their old values.

4.9 PERFORMANCE ANALYSIS OF SCHEDULING POLICIES

Performance analysis of scheduling policies can be used for two purposes—to compare performance of alternative scheduling policies and to determine “good” values of key system parameters like time slice and number of active users. Performance is sensitive to the workload of requests directed at the scheduler, hence it is essential to conduct performance analysis in the environment in which a scheduler is to be used. We consider different approaches to performance analysis and discuss their advantages and drawbacks.

A common difficulty in performance analysis concerns accurate characterization

of the typical workload. There are several reasons for this difficulty. As mentioned in the context of SRN policy, user estimates of service times are not reliable either because users lack the experience to provide good estimates of service time or because knowledgeable users may provide misleading estimates to obtain favored service from the system. Some users may even resort to changes in their requests to obtain better service—if a user knows that SRN policy is being used, he may split a long running job into several jobs with short service times. All these factors distort the workload, so great care has to be exercised while developing a characterization of typical workload.

Performance of a scheduling policy in a specific computational environment can be studied by implementing it in an OS and subjecting the OS to a workload of typical requests found in the environment. However, this approach is not feasible due to complexity, cost and delays involved in implementing the required scheduling policy in an OS.

Two practical approaches to performance analysis of scheduling policies are simulation and mathematical modeling. A *simulator* mimics decisions of a scheduler and relevant actions of an OS and determines completion times of requests in a typical workload. This is achieved by coding the scheduling policy and relevant OS functions as a program—the simulator program—and using a typical workload as its input. The workload is a recording of the real workload directed at the OS during a sample period. Analysis may be repeated with many workloads to eliminate the effect of variations across workloads. To eliminate workload distortion, care must be taken to record the workloads without the knowledge of users!

A *mathematical model* consists of two components—a model of the server and a model of the workload being processed. The model provides a set of mathematical expressions for important performance characteristics like service times of requests and overhead. These expressions provide insights into the influence of various parameters on system performance. The workload model differs from workloads used in simulations in that it is not a recording of actual workload in any specific time period. It is a mathematical abstraction called a distribution that is a good approximation for the actual workload observed during *any* period.

Queuing theory Widespread use of mathematical models to analyze performance of various systems led to development of a separate branch of mathematics known as *queuing theory*. Performance analysis using queuing theory is called queuing analysis. The earliest well known application of mathematical modeling was by Erlang (1909) in evaluating the performance of a telephone exchange with the number of trunk lines as the controlling parameter.

The fundamental queuing theory model of a system is identical with the simple scheduler model discussed at the start of this Chapter (see Figure 4.1). This is known as the single-server model. Requests arriving into the system are entered into a queue. The server selects requests from this queue for the purpose of scheduling. A request

leaves the system on completion. If the server is preemptible, a preempted request is put back into the scheduling queue and has to await rescheduling in order to resume execution. This is illustrated by the dashed arrow in Figure 4.1. Queuing analysis is used to develop mathematical expressions for server efficiency, mean queue length, and mean wait time.

A request arriving at time A_i with service time X_i is completed at time C_i . The elapsed time ($C_i - A_i$) depends on two factors—arrival times and service times of requests that are either in execution or in the scheduling queue at some time during the interval ($C_i - A_i$), and the scheduling policy used by the server. It is reasonable to assume that arrival and service times of requests entering the system are not known in advance, i.e., these characteristics of requests are nondeterministic in nature.

The scheduling environment is characterized by two parameters—the *arrival pattern* of requests and their *service pattern*. These parameters govern the arrival times and service times of requests, respectively. Although characteristics of individual requests are unknown, they are customarily assumed to conform to certain statistical distributions. We give a brief introduction to statistical distributions and their use in mathematical modeling using the following notation:

- α : Mean arrival rate (requests per second)
- ω : Mean execution rate (requests per second)
- ρ : α/ω

ρ is called the utilization factor of the server. When $\rho > 1$, the work being directed at the system exceeds its capacity. In this case, the number of requests in the system increases indefinitely. Performance evaluation of such a system is of little practical relevance since turn around times can be arbitrarily large. When $\rho < 1$, the system capacity exceeds the total work directed at it. However, this is true only as a long-term average; it may not hold in an arbitrary interval of time. Hence the server may be idle once in a while, and a few requests may exist in the queue at certain times.

Most practical systems satisfy $\rho < 1$. Even when we consider a slow server, ρ does not exceed 1 because most practical systems are self-regulatory in nature—the number of users is finite and the arrival rate of requests slackens when the queue length is large because most user requests are locked up in the queue!

A system reaches a *steady state* when all transients in the system induced due to its abrupt initiation at time $t = 0$ die down. In the steady state, values of mean queue lengths, mean wait times, and mean turn-around times reflect performance of the scheduling policy. For obtaining these values, we start by assuming certain distributions for arrival and servicing of requests in the system.

Arrival times The time between arrival of two consecutive requests is called inter-arrival time. Since α is the arrival rate, the mean inter-arrival time is $1/\alpha$. A distribution that has this mean inter-arrival rate and which fits empirical data reasonably well can be used as the workload characterization for queuing analysis. Arrival of

requests in the system can be regarded as random events totally independent of each other. Two assumptions leading to a Poisson distribution of arrivals are now made. First, the number of arrivals in an interval t to $t + dt$ depends only on the value of dt and not on past history of the system during the interval $(0, t)$. Second, for small values of dt , probability of more than one arrival in the interval t to $(t + dt)$ is negligible. The first assumption is known as the *memoryless property* of the arrival times distribution. An exponential distribution function giving the probability of an arrival in the interval 0 to t for any t , has the form:

$$F(t) = 1 - e^{-\alpha t}$$

This distribution has the mean inter-arrival time $1/\alpha$ since $\int_0^\infty t.dF(t) = 1/\alpha$. It is found that the exponential distribution fits the inter-arrival times in empirical data reasonably well. (However, a hyper-exponential distribution with the same mean of $1/\alpha$ is found to be a better approximation for the experimental data (Fife, 1965)).

Service times Function $S(t)$ gives the probability that service time of a request is less than or equal to t .

$$S(t) = 1 - e^{-\alpha t}$$

As in the case of arrival times, we make two assumptions that lead to a Poisson distribution of service times. Hence the probability that a request that has already consumed t units of service time will terminate in the next dt seconds depends only on the value of dt and not on t . In preemptive scheduling, this applies every time a request is scheduled to run after an interruption.

The memoryless property of service times implies that a scheduling algorithm cannot make any predictions based on past history of a request in the system. Thus, any preemptive scheduling policy that requires knowledge of future behavior of requests must depend on estimates of service times supplied by a programmer. The scheduling performance will then critically depend on user behavior and may be manipulated by users. In a practical situation, a system must strive to achieve the opposite effect—that is, system performance should be immune to user specification (or misspecification) of a request's service time. This feature seems to point towards round-robin scheduling with time slicing as a practical scheduling policy.

Performance analysis The relation between L , the mean queue length and W , the mean wait time for a request before its servicing begins is given by Little's formula, viz.

$$L = \alpha \times W \quad (4.6)$$

This relation follows from the fact that while a request waits in the queue, $\alpha \times W$ new requests join the queue.

When a new request arrives, it is added to the request queue. In non-preemptive scheduling, the new request would be considered only after the server completes the

request it is servicing. Let W_0 be the expected time to complete the current request. Naturally, W_0 is independent of a scheduling policy. $W_0 = \frac{\alpha}{2} \cdot \int_0^\infty t^2 dF(t)$, and has the value $\frac{\alpha}{\omega^2}$ for an exponential distribution $F(t) = 1 - e^{-\alpha t}$. W , the mean wait time for a request when a specific scheduling policy is used, is computed using W_0 and features of the scheduling policy. We outline how the mean wait times for FCFS and SRN policies are derived. Derivations for HRN and round-robin are more complex and can be found in Brinch Hansen (1973). Table 4.16 summarizes the mean wait time for a request whose service time is t when different scheduling policies are used.

Table 4.16 Summary of performance analysis

Scheduling policy	Mean wait time for a request with service time = t
FCFS	$\frac{W_0}{1-\rho}$
SRN	$\frac{W_0}{1-\rho_t}$, where $\rho_t = \int_0^t \alpha \cdot x \cdot dS(x)$
HRN (Non-preemptive)	for small t : $W_0 + \frac{\rho^2}{1-\rho} \times \frac{t}{2}$ for large t : $\frac{W_0}{(1-\rho)(1-\rho+\frac{2W_0}{t})}$
Round-robin	$\frac{n}{\alpha(1-P_0)} - \frac{1}{\alpha}$, where $P_0 = \frac{1}{\sum_{j=0}^n \frac{\alpha^j}{(n-j)!} \times (\alpha)^j}$ (P_0 is the probability that no terminal awaits a response)

W , the waiting time for some request r' , is the amount of time r' spends in the queue before its service begins. Hence in FCFS scheduling

$$W = W_0 + \sum_i (X_i)$$

where request i is ahead of request r' in the scheduling queue. The system is in the steady state, so we can replace the \sum_i term by $n \times \frac{1}{\omega}$, where n is the number of requests ahead of r' and $\frac{1}{\omega}$ is the mean service time. Since n is the mean queue length, $n = \alpha \times W$ from Little's formula. Hence

$$\begin{aligned} W &= W_0 + \alpha \times W \times \frac{1}{\omega} \\ &= W_0 + \rho \times W. \end{aligned}$$

Therefore, $W = \frac{W_0}{1-\rho}$. Thus, the mean wait time in FCFS scheduling rises sharply for high values of ρ .

In SRN scheduling, requests whose service times $< X_{r'}$, where $X_{r'}$ is the service time of r' , are serviced before request r' . Hence the waiting time for request r' is

$$\begin{aligned} W &= W_0 + \sum_i (X_i), \text{ where } X_i < X_{r'} \\ &= \frac{W_0}{1-\rho_{r'}}, \text{ where } \rho_{r'} = \int_0^{r'} \alpha \cdot x \cdot dS(x). \end{aligned}$$

Capacity planning Performance analysis can be used for capacity planning. For example, the formulae shown in Table 4.16 can be used to determine values of important parameters like sizes of process queues used by the kernel.

As an example, consider an OS in which the mean arrival rate of requests is 5 requests per second, and the mean response time for requests is 3 seconds. The mean queue length is computed by Little's formula (Eq. (4.6)) as $5 \times 3 = 15$. Note that queues will exceed this length from time to time. Example 4.17 provides a basis for deciding the capacity of the *ready* queue.

Example 4.17 An OS kernel permits upto n entries in the queue of *ready* requests. If the queue is full when a new request arrives, the request is rejected and leaves the OS. P_i , the probability that the ready queue contains i processes at any time, can be shown to be:

$$P_i = \frac{\rho^i \times (1 - \rho)}{1 - \rho^{n+1}} \quad (4.7)$$

For $\rho = 0.5$ and $n = 3$, $P_0 = \frac{8}{15}$, $P_1 = \frac{4}{15}$, $P_2 = \frac{2}{15}$, and $P_3 = \frac{1}{15}$. Hence 6.7 percent of requests are lost. A higher value of n should be used if fewer requests are to be lost.

EXERCISE 4

1. Give examples of conflicts between user-centric and system-centric views of scheduling.
2. Study performance of the non-preemptive and preemptive scheduling policies on processes described in Table 4.2 if their arrival times are 0, 1, 3, 7 and 10 seconds, respectively. Draw charts analogous to Figs. 4.2 and 4.4 to show operation of these policies.
3. Comment on correctness of the following statements:
 - (a) "If a system using the SRN scheduling policy completes execution of requests in the sequence r_1, r_2, \dots, r_n , weighted turn around of $r_i >$ weighted turn around of r_j if $i > j$."
 - (b) "LCN scheduling provides better turn around times for I/O-bound requests than provided by RR scheduling."
4. Prove that SRN scheduling provides the minimum average turn-around time for a set of requests that arrive at the same time instant.
5. If the highest response ratio next (HRN) scheduling policy is applied preemptively, compare and contrast it with the following policies:
 - (a) Shortest Time to Go (STG) policy.
 - (b) Least completed Next (LCN) policy.
 - (c) Round-robin (RR) policy with time slicing.

(Hint: Think of the service provided to short and long processes.)
6. An OS implements the HRN policy as follows: Every t seconds, response ratios of all processes are computed. This is followed by scheduling, which selects the process with the highest response ratio. Comment on the performance and overhead of the scheduling policy for large and small values of t .

7. A program contains a single loop that executes 50 times. The loop contains a computation that lasts 50 msec followed by an I/O operation that consumes 200 msec. This program is executed in a time-sharing system with 9 other identical programs. All programs start their execution at the same time. The scheduling overhead of the OS is 3 msec. Compute the response time in the first and subsequent iterations if
 - (a) The time slice is 50 msec.
 - (b) The time slice is 20 msec.
8. Describe actions of the kernel when a time slice elapses.
9. Is it possible to predict the average response time experienced by a process in scheduling level i of CTSS if n_i processes exist at level i ?
10. Houston Automatic SPooling system (HASP) was a scheduling sub-system used in the IBM/360. HASP assigns high priority to I/O-bound processes and low priority to CPU-bound processes. A process is classified into CPU-bound or I/O-bound based on its recent behavior vis-a-vis the time slice—a process is a CPU-bound process if it uses up its entire time-slice when scheduled; otherwise, it is an I/O-bound process. To obtain good throughput, HASP requires that a fixed percentage of processes in the scheduling queue must be I/O-bound processes. Periodically, HASP adjusts the time-slice to satisfy this requirement—the time slice is reduced if more processes are considered I/O-bound than desired, and it is increased if lesser number of processes are I/O-bound.
Explain the purpose of adjusting the time slice. Describe operation of HASP if most processes in the system are (a) CPU-bound, and (b) I/O-bound.
11. Comment on similarities and differences between
 - (a) LCN and Unix scheduling
 - (b) HASP and multilevel adaptive scheduling.
12. Determine the starting deadlines for processes of Example 4.13.
13. Comment on validity of the following statement:
“The Unix scheduling policy favors interactive processes over non-interactive processes.”
14. g_i is a group of processes in a system using fair share scheduling. When a process P_j from g_i is selected for scheduling, we say that “ P_j is a selection from g_i ”. Show that if processes do not perform I/O operations, two consecutive selections from g_i cannot be for the same process.
15. An OS using a preemptive scheduling policy uses dynamically changing priorities. The priority of a process changes at different rates depending on its state as follows

- α : Rate of change of priority when a process is *running*
 β : Rate of change of priority when a process is *ready*
 γ : Rate of change of priority when a process is performing I/O

Note that the rate of change of priority can be positive, negative or zero. A process has priority 0 when it is created. A process with a larger numerical value of priority is considered to have a higher priority for scheduling.

Comment on the resulting scheduling policies if

- (a) $\alpha > 0, \beta = 0, \gamma = 0$.
- (b) $\alpha = 0, \beta > 0, \gamma = 0$.

- (c) $\alpha = \beta = 0, \gamma > 0$.
 (d) $\alpha < 0, \beta = 0, \gamma = 0$.

Will behavior of the scheduling policies change if the priority of a process is set to 0 every time it is scheduled?

16. Give a yes/no answer to the following questions:
- An I/O-bound process is executed twice, once in a system using round-robin scheduling and again in a system using multilevel adaptive scheduling. The number of times it is scheduled by the round-robin scheduler and by the multi-level scheduler is identical.
 - If processes do not perform I/O, round-robin scheduling resembles shortest time to go (STG) scheduling.
 - If processes do not perform I/O, the Unix scheduling policy degenerates to the conventional round-robin scheduling policy.
17. Explain how starvation is avoided in the Unix and Windows systems.
18. A system uses the FCFS scheduling policy. Identical computational requests arrive in the system at the rate of 20 requests per second. It is desired that the mean wait time in the system should not exceed 2.0 seconds. Compute the size of each request in CPU seconds.
19. Identical requests, each requiring 0.05 CPU seconds, arrive in an OS at the rate of 10 requests per second. The kernel uses a fixed sized *ready* queue. A new request is entered in the *ready* queue if the queue is not already full, otherwise the request is discarded. What should be the size of the ready queue if less than 1 percent of requests should be discarded.
20. The mean arrival rate of requests in a system using FCFS scheduling is 5 requests per second. The mean wait time for a request is 3 seconds. Find the mean execution rate.
21. We define 'small request' as a request whose service time is less than 5 percent of $\frac{1}{\alpha}$. Compute the turn-around time for a small request in a system using the non-preemptive HRN scheduling policy when $\alpha = 5$ and $\omega = 8$.

BIBLIOGRAPHY

Corbató *et al* (1962) discuss use of multilevel feedback queues in the CTSS operating system. Coffman and Denning (1973) report studies related to multilevel scheduling. The fair share scheduler is described in Kay and Lauder (1988), and lottery scheduling is described in Waldspurger and Weihs (1994). Real time scheduling is discussed in Liu and Layland (1973), Zhao(1989), Khanna *et al* (1992) and Liu (2000). Power conservation is a crucial new element in scheduling. Power can be conserved by running the CPU at lower speeds. Zhu *et al* (2004) discuss speculative scheduling algorithms that save power by varying the CPU speed and reducing the number of speed changes while ensuring that an application meets its time constraints.

Bach (1986), McKusick *et al* (1996) and Vahalia (1996) discuss scheduling in Unix, Bovet and Cesati (2003), O'Gorman (2003) and Love (2005) discuss scheduling in Linux, Mauro and McDougall (2001) discuss scheduling in Solaris, while Russinovich and Solomon (2005) discuss scheduling in Windows.

Queueing theory is used in scheduling and performance evaluation. The earliest application of queuing theory was by Erlang in 1909, who applied it in the performance evaluation

of a telephone exchange. Trivedi (1982) is an excellent source on all aspects of queuing theory. Earlier works include Kleinrock (1975, 1976) and Coffman (1976). Brinch Hansen (1973) introduces the necessary queuing theory background for the discussion of scheduling. Hellerman and Conroy (1975) describe use of queuing theory in performance evaluation.

1. Bach, M. J. (1986): *The Design of the Unix Operating System*, Prentice-Hall, Englewood Cliffs.
2. Bovet, D. P., and M. Cesati (2003): *Understanding the Linux Kernel*, O'Reilly, Sebastopol.
3. Brinch Hansen, P. (1973): *Operating System Principles*, Prentice-Hall, Englewood Cliffs.
4. Bruno, J., E. G. Coffman, and R. Sethi (1974): "Scheduling independent tasks to reduce mean finishing time," *Communications of the ACM*, **17** (7), 382–387.
5. Coffman, E. G. (Ed.) (1976): *Computer and Job Shop Scheduling*, Wiley, New York.
6. Coffman, E. G., and P. J. Denning (1973): *Operating Systems Theory*, Prentice-Hall, Englewood Cliffs.
7. Conway, R. W., W. L. Maxwell, and L. W. Miller (1967): *Theory of Scheduling*, Addison-Wesley, Reading.
8. Corbato, F. J., M. Merwin-Daggett, and R. C. Daley (1962): "An experimental time-sharing system," *Proceedings of the AFIPS Fall Joint Computer Conference*, 335–344.
9. Hellerman, H., and T. F. Conroy (1975): *Computer System Performance*, McGraw-Hill Kogakusha, Tokyo.
10. Kay, J., and P. Lauder (1988): "A fair share scheduler," *Communications of the ACM*, **31** (1), 44–55.
11. Khanna, S., M. Sebree, and J. Zolnowsky (1992): "Real time scheduling in SunOS 5.0," *Proceedings of the Winter 1992 USENIX Conference*, San Francisco, January 1992, 375–390.
12. Kleinrock, L. (1975–1976): *Queuing Systems*, Vols. I and II, Wiley, New York.
13. Love, R. (2005): *Linux Kernel Development, Second edition*, Novell Press.
14. Liu, C. L., and J. W. Layland (1973): "Scheduling algorithms for multiprogramming in a hard real-time environment," *Communications of the ACM*, **20**, 1, 46–61.
15. Liu, J. W. S. (2000): *Real-Time Systems*, Pearson education.
16. Mauro, J., and R. McDougall (2001): *Solaris Internals—Core Kernel Architecture*, Prentice-Hall.
17. McKusick, M. K., K. Bostic, M. J. Karels, and J. S. Quarterman (1996): *The Design and Implementation of the 4.4BSD Operating System*, Addison Wesley, Reading.
18. O'Gorman, J. (2003): *Linux Process Manager: The internals of Scheduling, Interrupts and Signals*, Wiley and Sons.
19. Russinovich, M. E., and D. A. Solomon (2005): *Microsoft Windows Internals, Fourth edition*, Microsoft Press, Redmond.
20. Trivedi, K. S. (1982): *Probability and Statistics with Reliability—Queuing and Computer Science Applications*, Prentice-Hall, Englewood Cliffs.
21. Vahalia, U. (1996): *Unix Internals: The New Frontiers*, Prentice Hall, Englewood Cliffs.

22. Waldspurger, C. A., and W. E. Weihl (1994): "Lottery scheduling," *Proceedings of the First USENIX Symposium on Operating System Design and Implementation (OSDI)*, 1–11.
23. Zhao, W. (1989) : Special issue on real-time operating systems, *Operating System Review*, **23**, 7.
24. Zhu, D., D. Mosse, and R. Melhem (2004): "Power-aware scheduling for AND/OR graphs in real-time systems," *IEEE Transactions on Parallel and Distributed Systems*, **15** (9), 849–864.

Memory Management

As seen in Chapter 2, the memory hierarchy is comprised of the cache, the memory management unit (MMU), random access memory (RAM), which is simply called *memory* in this chapter; and a disk. We discuss management of memory by the OS in two parts—this chapter discusses techniques for efficient use of memory, whereas the next chapter discusses management of *virtual memory*, which is part of the memory hierarchy consisting of the memory and the disk.

We begin by discussing fundamentals of static and dynamic memory allocation. We then introduce the model used for memory allocation to a process and show how it accommodates both static and dynamic components.

To ensure efficient use of memory, the kernel reuses the memory allocated to a process when the process terminates. *Memory fragmentation* is a problem that makes some areas of memory unusable, leading to inefficient use of memory. We discuss practical techniques used to reduce memory fragmentation, particularly *noncontiguous memory allocation* using *paging* and *segmentation*. The kernel uses part of the memory for its own activities—for control blocks such as the PCBs and ECBs. The sizes of these data are known *a priori*, so the kernel uses special techniques that exploit this knowledge to achieve fast allocation/deallocation of control blocks and efficient use of memory. The last section discusses linking, relocation, and properties of program forms. This section is useful for readers who are uninitiated in these topics.

5.1 MANAGING THE MEMORY HIERARCHY

As discussed earlier in Chapter 2, the memory hierarchy is an arrangement of several memory units with varying speeds and sizes that creates an illusion of a fast and large memory at a low cost. The CPU refers to the fastest memory, the *cache*, when it needs to access an instruction or data. If the required instruction or data is not available in the cache, it is fetched from the next lower level in the memory hierarchy, which

could be a slower cache or the random access memory (RAM), simply called *memory* in this book. If the required instruction or data is also not available in the next lower level memory, it is fetched there from a still lower level, and so on. Performance of the memory hierarchy depends on the *hit ratios* in various levels of the hierarchy, where the hit ratio in a level indicates what fraction of instructions or data bytes that were looked for in that level were actually present in it. Eq. (2.1) of Chapter 2 indicates how the effective memory access time depends on a hit ratio.

Figure 5.1 illustrates the memory hierarchy and describes its operation. The memory hierarchy is comprised of cache memories like the L1 and L2 caches, the memory management unit (MMU), memory and a disk. The L1 and L2 caches are managed in the hardware, so the kernel and user programs must employ special techniques to achieve high hit ratios in the caches. For example, the kernel removes the address space of a preempted or blocked process from a cache to ensure memory protection. This action leads to a poor cache hit ratio for a process when its operation is initiated or resumed after a break. The kernel counters this effect by switching between threads of the same process whenever possible; and through *affinity scheduling* in a multiprocessor system (see Section 13.5). In Section 5.10, we shall discuss special design techniques employed by the kernel to ensure good cache performance while accessing kernel data structures.

The kernel allocates memory to user processes. The primary performance concern in this function is accommodating more user processes in memory, so that both system performance and user service would improve. The kernel meets this concern through efficient reuse of memory when a process completes, which reduces the amount of unused memory in the system at any moment. During operation, a process creates data structures *within* the memory already allocated to it by the kernel. This function is actually performed by the run-time library of the programming language in which the code of the process is written. The run-time library employs techniques that efficiently reuse memory when a process creates and destroys data structures during its operation. Thus some of the concerns and techniques employed by the kernel and the run-time libraries are similar.

As a sequel to the kernel's focus on accommodating a large number of processes in memory, the kernel may decide on keeping only a part of each process's address space in memory. This is achieved using the part of the memory hierarchy called *virtual memory* that comprises of memory and a disk (see the dashed box in Figure 5.1). The parts of a process's address space that are not in memory are loaded from the disk when needed during operation of the process. In this arrangement, the hit ratio of a process in memory determines its performance. Hence the kernel employs a set of techniques to ensure a high hit ratio for processes. The disk in the virtual memory is managed entirely by the kernel; the kernel stores different parts of each process's address space on the disk in such a manner that they can be accessed efficiently. This contributes to good execution performance of processes in a virtual memory.

We discuss management of the memory hierarchy by an operating system in two

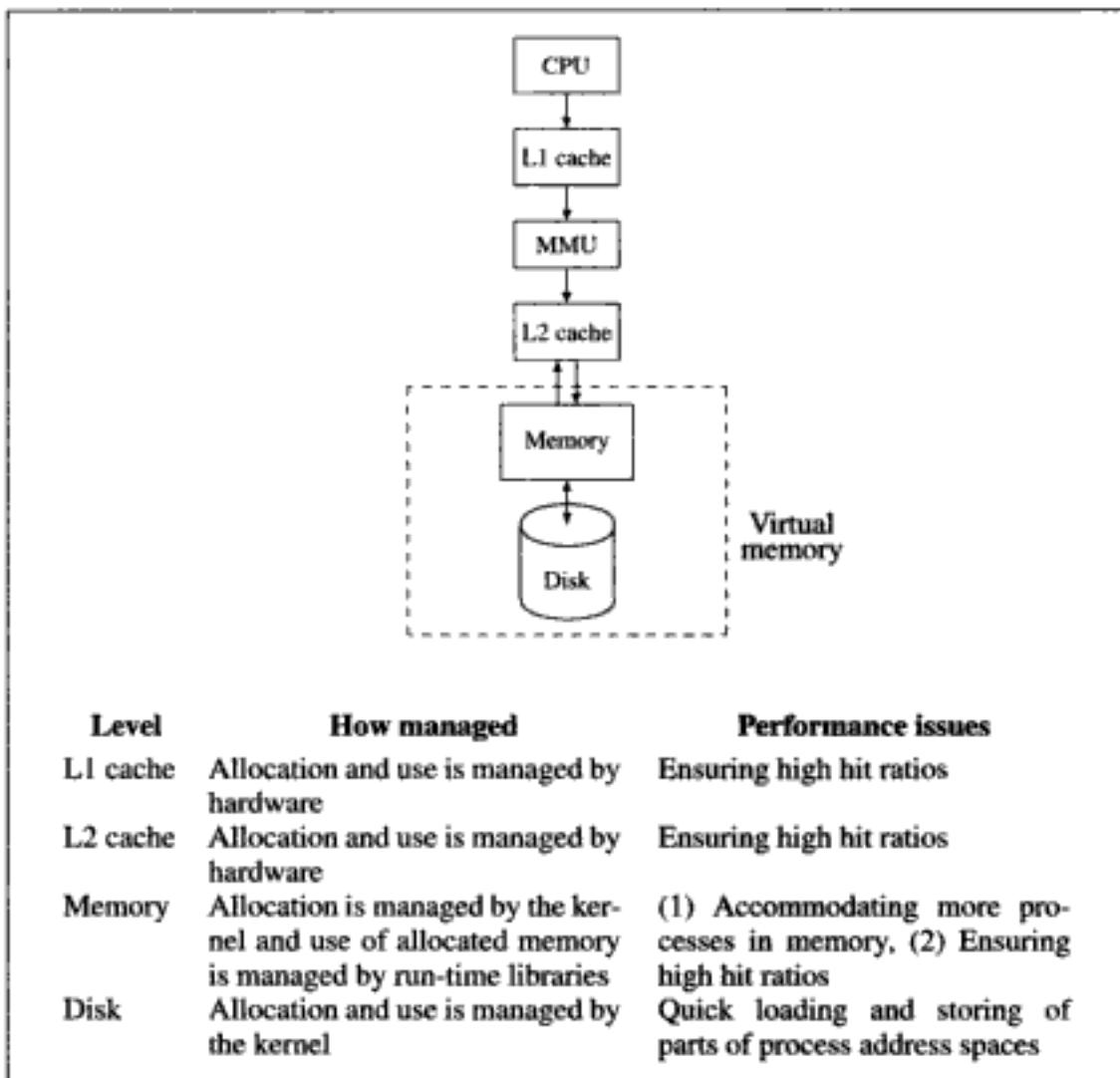


Fig. 5.1 Managing the memory hierarchy

parts. This chapter focuses on the management of memory, and focuses on techniques employed for efficient use of memory and for speedy allocation and deallocation of memory. Later we discuss how presence of the memory management unit (MMU) simplifies both these functions. Chapter 6 discusses management of the virtual memory, particularly the techniques employed by the kernel to ensure high hit ratios in memory and limit the memory committed to each process.

5.2 STATIC AND DYNAMIC MEMORY ALLOCATION

Memory allocation to a process involves specification of memory addresses to its instructions and data. Memory allocation is an aspect of a more general action known as *binding*. Two other aspects related to the execution of a program, viz. linking and

loading, are also aspects of binding. We begin this chapter with a quick overview of two principal methods of performing binding, their fundamental properties, and their implications for memory allocation, linking and loading of programs.

An entity has a set of attributes. Each attribute has a value. For example, a variable in a program has attributes like type, dimensionality, scope and memory address. *Binding* is the act of specifying the value of an attribute. Memory allocation is the act of specifying the memory address attribute of an entity.

Bindings should be performed for the attributes of an entity before the entity is used during operation of a process or a software system. *Static* and *dynamic* binding are two fundamental methods of performing binding.

Definition 5.1 (Static and dynamic binding) Static binding is a binding performed before the operation of a program (or a software system) begins, while dynamic binding is a binding performed during its operation.

More information concerning an entity may be available during execution of a program than before its execution begins. Using this information, it may be possible to perform better quality binding. For example, dynamic binding may be able to achieve more efficient use of resources such as memory.

Static and dynamic memory allocation Static memory allocation can be performed by a compiler, linker or loader while readying a program for execution. Dynamic memory allocation is performed in a lazy manner. That is, memory is allocated to an entity just before it is used for the first time during the execution of a program.

Static memory allocation is possible only if data sizes are known before execution of a program is initiated. If sizes are not known, they have to be guessed, which can lead to wastage of memory and lack of flexibility. For example, consider an array whose size is not known during compilation. Memory is wasted if its guessed size is larger than its actual size, whereas a program cannot execute correctly if its actual size is larger than the guessed size. Dynamic allocation can avoid both these problems since the actual size of the array would be known at the time of the allocation.

Static memory allocation does not require any memory allocation actions during program execution. By contrast, dynamic memory allocation incurs the overhead of memory allocation actions performed during execution of a program. Some of these actions are even repeated several times during execution of a program. Table 5.1 summarizes key features of static and dynamic memory allocation.

Operating systems also exploit features of static and dynamic memory allocation to obtain the best mix of execution efficiency and flexibility. Where flexibility is either not important or not needed, the OS makes memory allocation decisions statically. This approach provides execution efficiency. Where flexibility is needed and its cost is justifiable, the OS performs memory allocation decisions dynamically. For example, it uses static memory allocation for some attributes of kernel data structures because it knows characteristics of kernel data (see Section 5.10), whereas it may per-

Table 5.1 Features of static and dynamic memory allocation

Kind of allocation	Key features
Static allocation	<ul style="list-style-type: none"> • Allocation is performed <i>before</i> start of execution of a program. • Size of data should be known before start of execution, else allocated memory may be wasted or a process may run out of memory during its operation. • No allocation actions during program execution.
Dynamic allocation	<ul style="list-style-type: none"> • Allocation is performed <i>during</i> execution of a program. • Allocation exactly equals data size; no wastage of memory. • Allocation overhead during program execution.

form dynamic memory allocation for user programs because it does not know much about them. It may also perform dynamic memory allocation to different parts of a program to reduce memory wastage and improve system performance.

Static and dynamic linking/loading The distinction between the terms linking and loading has become blurred. However, we use the terms as follows: A *linker* links modules together to form an executable program. A *loader* loads a program or a part of a program in memory for execution.

A static linker links all modules of a program before its execution begins. If several programs use the same module from a library, each program would have a private copy of the module and several copies of the module might exist in the memory at the same time. Dynamic linking of a module is performed when a reference to it is executed. It provides a wide range of possibilities concerning use, sharing and updating of library modules. Modules that are not invoked during execution of a program need not be linked to it at all. If the module referenced by a program has already been linked to another program that is also in execution, the same copy can be linked to this program as well. Dynamic linking also provides an interesting advantage when a library of modules is updated—any program that invokes a new module automatically starts using the new version of the module! Dynamically linked libraries (DLLs) use some of these features to advantage.

Dynamic linking requires dynamic loading. However, dynamic loading can be used with static linking as well. Its advantage is that modules that are not invoked need not be loaded at all. Another way to conserve memory is to overwrite a module existing in memory with a new module. This idea is used in *virtual memory*, which is discussed in the next Chapter.

5.3 MEMORY ALLOCATION TO A PROCESS

5.3.1 Stacks and Heaps

The compiler of a programming language generates code for a program and allocates its static data. It creates an object module for the program (see Section 5.11) that contains the code, static data, and information about size of the program. The linker links the program with library functions and the run-time support of the programming language, prepares a ready-to-execute form of the program, and stores it in a file. The program size information is recorded in the directory entry of the file.

The run-time support allocates two kinds of data during execution of the program. The first kind of data includes variables whose scope is associated with functions, procedures or blocks in a program. These data are allocated when a function, procedure or block is entered and are deallocated when it is exited. Due to the last-in-first-out nature of their allocation/deallocation, these data are allocated on the stack. The second kind of data is data dynamically created by a program using language features like the `new` statement of Pascal, C++ or Java, or the `malloc`, `calloc` statements of C. We refer to such data as *program controlled dynamic data* (PCD data). The PCD data is allocated using a data structure called a *heap*.

Stack In a *stack*, allocations and deallocations are performed in a last-in-first-out (LIFO) manner in response to *push* and *pop* operations, respectively. We assume each entry in the stack to be of some standard size, say, l bytes. Only the last entry of the stack is accessible at any time. A contiguous area of memory is reserved for the stack. A pointer called the *stack base* (SB) points to the first entry of the stack, while a pointer called the *top of stack* (TOS) points to the last entry allocated in the stack. We will use the convention that a stack grows towards the lower end of memory; we depict it as upwards growth in the figures.

During execution of a program, a stack is used to support function calls. The group of stack entries that pertain to one function call is called a *stack frame*; it is also called an *activation record* in compiler terminology. A stack frame is pushed on the stack when a function is called. To start with, the stack frame contains either addresses or values of the function's parameters, and the *return address*, i.e., the address of the instruction to which control should be returned after completing the function's execution. During execution of the function, the run-time support of the programming language in which the program is coded creates local data of the function within the stack frame. At the end of the function's execution, the entire stack frame is popped off the stack and the return address is used to pass control back to the calling program.

Two provisions are made to facilitate use of stack frames: The first entry in a stack frame is a pointer to the previous stack frame on the stack. This entry facilitates popping off of a stack frame. An additional pointer called the *frame base* (FB) is used to point to the start of the topmost stack frame in the stack. It helps in accessing various stack entries in the stack frame. Example 5.1 illustrates how the stack is used

to implement function calls.

Example 5.1 Figure 5.2 shows the stack during execution of a program containing nested function calls. Figure 5.2(a) shows the stack after main, the primary function of the program, has made a function call `sample(x,y,i)`. A stack frame was pushed on the stack when the call was made. The first entry in the stack frame contains a pointer to the previous stack frame in the stack. The second entry is `ret.ad(main)`, which is the return address into function `main`. The next three entries pertain to the parameters `x`, `y` and `i`, while the entries following them pertain to the local data of function `sample`. The frame base pointer points to the first entry in the stack frame. The TOS pointer points to the last local data in the stack frame. The code for function `sample` accesses the return address, information about the parameters, and its local data using displacements from the frame base (FB): Assuming each stack entry to be 4 bytes, the return address is at a displacement of 4 from the frame base, the first parameter is at a displacement of 8 from the frame base, etc.

Figure 5.2(b) shows the stack after function `sample` has made a function call `calc(a,b,sum)`. A new stack frame has been pushed on the stack, the value of the frame base has been saved in the first entry of this stack frame, the frame base has been set to point at the start of the new stack frame, and the top of stack pointer now points at last entry in the new stack frame. At the completion of the function, the TOS pointer would be set to point at the stack entry preceding the entry pointed to by FB, and FB would be loaded with the address contained in the stack entry to which it was pointing. These actions would effectively pop off the stack frame of `calc` and set FB to point at the start of the stack frame for `sample`. The resulting stack would be identical to the stack before function `sample` called `calc`.

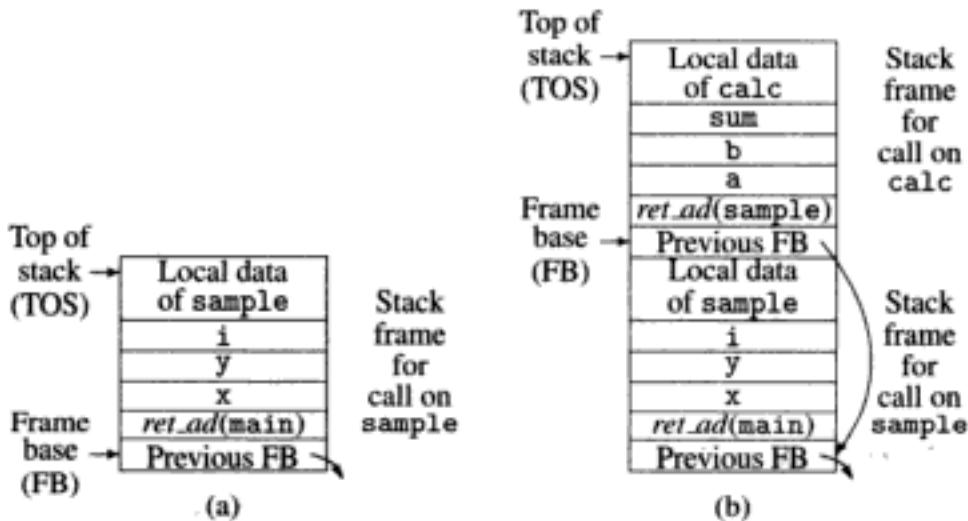


Fig. 5.2 Stack after (a) main calls sample, (b) sample calls calc

Heap A *heap* permits allocation and deallocation of memory in a random order. An allocation request by a process returns with a pointer to the allocated memory area in the heap, and the process accesses the allocated memory area through this pointer. A deallocation request must present a pointer to the memory area to be

deallocated. The next example illustrates use of a heap to manage the PCD data of a process. As illustrated there, holes develop in the memory allocation as data structures are created and freed. The heap allocator has to reuse such free memory areas while meeting future demands for memory.

Example 5.2 Figure 5.3 shows the status of a heap after executing the following C program:

```
float *floatptr1, *floatptr2;
int *intptr;
floatptr1 = (float *) calloc(5, sizeof(float));
floatptr2 = (float *) calloc(4, sizeof(float));
intptr = (int *) calloc(10, sizeof(int));
free(floatptr2);
```

The `calloc` routine is used to make a request for memory. The first call requests sufficient memory to accommodate 5 floating point numbers. The heap allocator allocates a memory area and returns a pointer to it. This pointer is stored in `floatptr1`. The first few bytes of each allocated memory area are assumed to contain a *length* field. This field is used during deallocation when the routine `free` is called with a pointer to an allocated memory area. Figure 5.3(a) shows the heap after all `calloc` calls have been processed. Figure 5.3(b) shows the heap after the `free` call. `free` has freed the memory area pointed to by `floatptr2`. This action has created a 'hole' in the allocation.

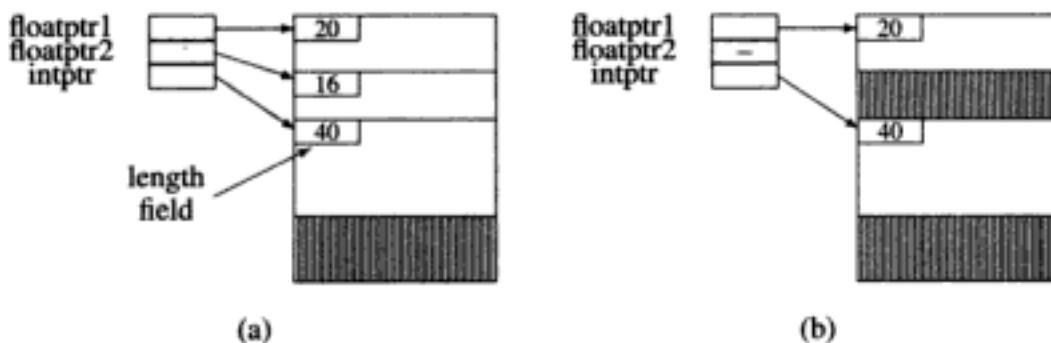


Fig. 5.3 (a) A heap, (b) A 'hole' in the allocation when memory is deallocated

5.3.2 The Memory Allocation Model

The memory allocated to a process contains the following:

- Code and static data of the program to be executed
- Stack
- Program controlled dynamic data (PCD data).

The compiler of a programming language generates the code for a program and allocates its static data. The linker links it with the code of library modules required

by it to prepare a ready-to-execute version of the program, and the loader loads this version in memory (see Section 5.11 for a review of linking and loading). The *stack* contains some program data and data that is created and used specifically to control execution of the program, e.g., parameters of procedures or functions that have been called but have not been exited from, and return addresses to be used while exiting from them.

Two kinds of data are allocated dynamically by the run-time support of a programming language. The first kind consists of variables whose scopes are associated with functions, procedures or blocks in a program. These data are allocated when a function, procedure or block is entered and are deallocated when it exits. Due to the LIFO nature of their allocation/deallocation, these data are allocated on the stack. The second kind of data concerns data dynamically created by a program using language features like the `new` statement of Pascal, C++ or Java, or the `malloc`, `calloc` statements of C. We refer to such data as *program controlled dynamic data* (PCD data). The PCD data is allocated using a data structure called a *heap* (see Section 5.3.1).

Sizes of the code and static data components in a program are known at the time of compilation. The compiler puts this information in the compiled form of a program (this form is called an *object module*—see Section 5.11). When the executable form of a program is stored in a file, the directory entry of the file contains the size information.

Sizes of the stack and the PCD data vary during execution of a program. So how does the kernel know how much memory to allocate for a program's execution? In general, it does not know. It can guess sizes of these dynamic components and allocate them a certain amount of memory. However, this amounts to static allocation, hence it lacks flexibility. As discussed in Section 5.2, the allocated memory may be wasted or a program may run out of space during execution.

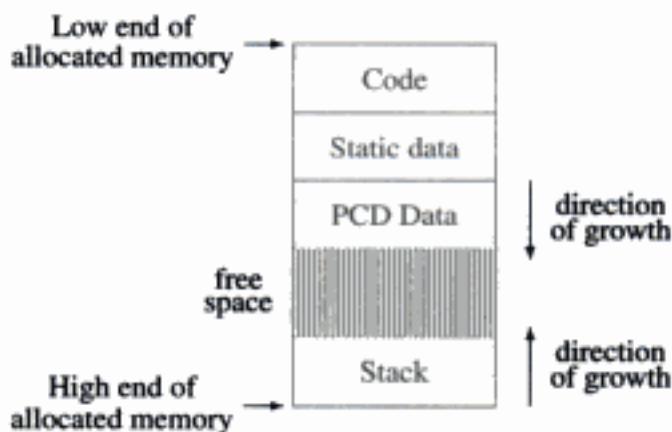


Fig. 5.4 Memory allocation model for a process

To avoid facing these problems individually for these two components, operating systems use the memory allocation model depicted in Figure 5.4. The code and static data components in the program are allocated memory areas that exactly match their sizes. The PCD data and the stack share a single large area of memory but grow in opposite directions when memory is allocated to new entities. The PCD data is allocated starting at the low end of this area while the stack is allocated starting at the high end of the area. The memory between these two components is free. It can be used to create new entities in either component. In this model the stack and PCD data components do not have individual size restrictions.

When the execution of a program is initiated, the kernel allocates a memory area to it and loads its code and static data. The remaining memory is free; it is used for PCD data and the stack as described earlier. New entries are created in the stack as procedure and function calls are made, while new entries are created in the PCD area when the program requests memory for program controlled data. These are aspects of dynamic memory allocation, hence as described in Table 5.1 they attract execution-time penalties due to allocation and deallocation actions.

During execution, a program creates or destroys PCD data by calling appropriate routines of the run-time library of the programming language in which it is coded. The library routines perform allocations/deallocations in the PCD data area allocated to the program's execution. Thus, the kernel is not involved in this kind of memory management. In fact it is oblivious to it.

5.3.3 Loading and Execution of a Program

A program is typically coded (or compiled) such that it can execute only in a specific area of memory. This memory area may not be available when the program is to be executed, so the program either has to be executed in the memory area for which it is prepared, or its code has to be changed so that it can execute correctly from some other memory area. The action performed in the latter case is called *relocation* of a program. It is performed by a *relocating loader*. Use of the relocating loader is accompanied by some processing and memory costs described later Section 5.11. Some computer architectures use an ingenuous scheme to facilitate execution of a program from *any* area of memory without incurring these costs.

Relocation register A *relocation register* is a special register in the CPU that helps in relocation of a program. It contains an integer number. The CPU adds this number to every address generated during execution of a program. The result is another memory address, which is used to make a memory reference. Thus,

$$\begin{aligned}\text{Effective memory address} &= \text{Address used in an instruction} \\ &\quad + \text{contents of relocation register.}\end{aligned}$$

Example 5.3 illustrates how the relocation register can be used to achieve relocation of a program.

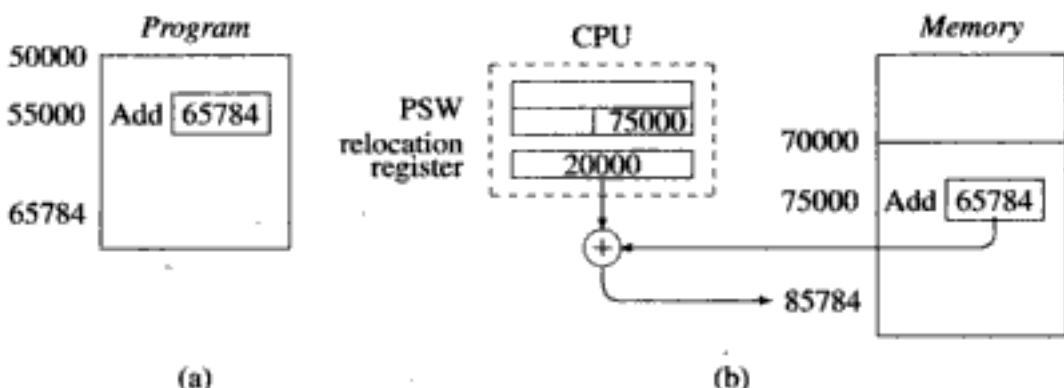


Fig. 5.5 Program relocation using a relocation register: (a) program, (b) execution

Example 5.3 A program has been prepared for execution in a memory area starting on the address 50000. We call this the assumed start address. Let the memory area allocated to the program's execution have the start address of 70000 (see Figure 5.5). The relocation of the program is achieved simply by loading an appropriate value in the relocation register, which is computed as follows:

The value to be loaded in relocation register
= start address of allocated memory assumed start address
= 70000 50000 = 20000.

Consider execution of the Add instruction in the program. It exists in the memory byte with the address 55000 in the program, and it accesses the memory byte with address 65784. This instruction would be loaded in the location with address 75000. During its execution, the memory byte with the address $65784 + 20000 = 85784$ would be accessed.

5.3.4 Memory Protection

To ensure that processes do not interfere with each other's code or data, every memory address used by a process should be checked to see whether it lies within the memory area(s) allocated to the process. For obvious reasons, this function cannot be performed by software, so its implementation requires support from a machine's architecture. This support comes in two forms—memory bound registers and memory protection keys associated with areas of memory.

As discussed in Section 2.1.1, memory protection using memory bound registers is implemented using the *lower bound register* (LBR) and the *upper bound register* (UBR) in the CPU. These registers contain the start and end addresses, respectively, of the memory area allocated to a process (see Figure 5.6). These registers are stored in the memory protection information (MPI) field of the PSW. The kernel loads appropriate values in LBR and UBR while scheduling a process for execution. A user process should not be able to tamper with the values in these registers, hence instructions for loading and saving these registers are made privileged instructions.

The memory protection hardware compares every memory address used by a

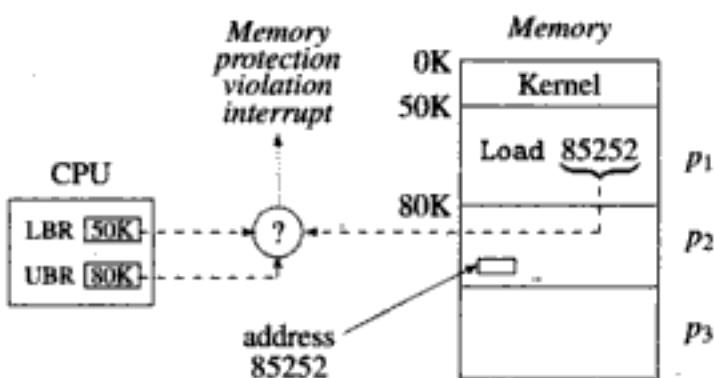


Fig. 5.6 Memory protection using bound registers

process with the contents of the memory bound registers. A *memory protection violation* interrupt is generated if the address is smaller than the address in LBR or larger than the address in UBR. On processing this interrupt, the kernel terminates the erring process.

When a relocation register is used (see Section 5.3.3), the effective memory address should be compared with the addresses contained in LBR and UBR. The memory protection check becomes simpler if every program has the assumed start address of '0000'. Now both the relocation register and LBR will contain the same address. So the lower bound register can be eliminated. Further, it is unnecessary to check whether an effective address is smaller than the address contained in the relocation register.

5.4 REUSE OF MEMORY

Two fundamental concerns in the design of a memory allocator are the speed of memory allocation and efficient use of memory. The latter concern has two facets

- *Memory overhead of the allocator:* Memory overhead includes the memory used by the memory allocator for its own operation, and the memory used by a requesting process to keep track of allocated memory.
- *Reuse of memory released by processes:* It should be possible to reuse the memory released by a process while making fresh allocations of memory.

In a stack, memory overhead consists of the SB and TOS pointers. Reuse of memory is automatic since memory released when a record is popped off the stack is used when a new record is created in the stack. In a heap, memory overhead consists of pointers to allocated areas. Reuse of memory is not automatic; the memory allocator must try to reuse a free area while making fresh allocations. Efficient use of memory becomes an important issue here because the size of a fresh request for memory seldom matches the size of any released memory areas, hence there is always a possibility that some memory may be wasted when a memory area is reused.

The memory allocator maintains a *free list* to keep information concerning all free memory areas in the system. This list is constructed using parts of the free areas. This way, its existence does not impose a memory overhead. Figure 5.7(a) shows a singly linked free list in a heap containing five areas marked a–e in active use and three free areas x–z. The first few bytes in each free area hold the size of the area. The next few bytes contain a pointer to the next free area in the list. Figure 5.7(b) shows a doubly linked free list. This organization would facilitate addition/deletion of new memory areas to/from the list. In both cases, entries can be arranged in the increasing order by size so that the 'best' area to satisfy a memory request can be readily identified.

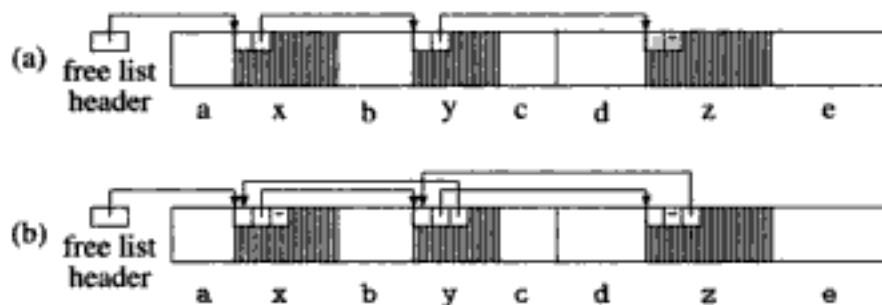


Fig. 5.7 Free area management (a) singly linked free list, (b) doubly linked free list

Performing fresh allocations using a free list

Three techniques can be used to perform a fresh allocation from a free list:

- First fit technique
- Best fit technique
- Next fit technique.

To service a request for n bytes of memory, the first fit technique uses the first free area it can find whose size is $\geq n$ bytes. n bytes are allocated to the request, and the remaining part of the area is put back into the free list. This technique suffers from the problem that a free area may be split several times, hence free memory areas may become successively smaller in size. The best fit technique uses the smallest free area with size $\geq n$. It avoids needless splitting of large areas; however, it tends to generate small free areas due to the splits. Hence in the long run it, too, may suffer from the problem of numerous unusable small free areas. It also incurs higher allocation cost because it either has to process the entire free list at every allocation, or maintain the free list in ascending order by size. The next fit technique is a compromise between these two techniques. It remembers the entry from which the last allocation was made. While making a new allocation, it searches the free list starting from the next entry and performs allocation using the first free area of size $\geq n$ bytes that it can find. This way, it avoids splitting the same free area repeatedly as in the

first fit technique and also avoids the allocation overhead of the best fit technique. Example 5.4 illustrates these allocation techniques.

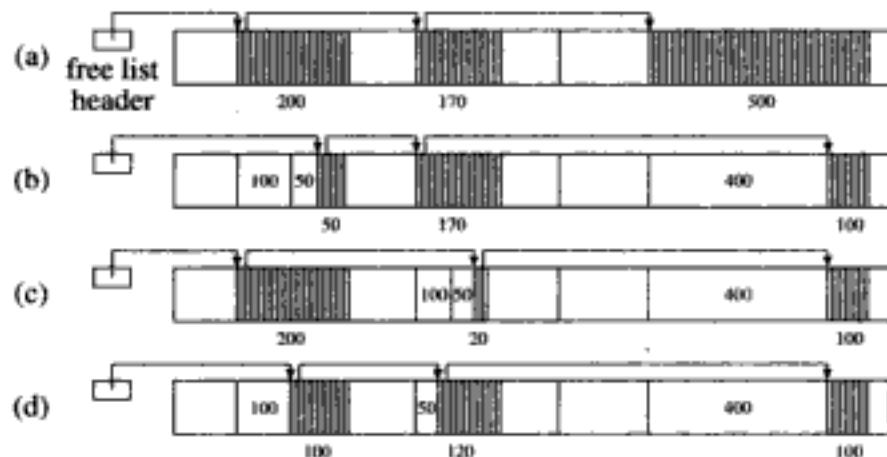


Fig. 5.8 (a) Free list, (b)–(d) allocation using first, best and next fit

Example 5.4 As shown in Figure 5.8(a), a free list contains three free areas of size 200, 170 and 500 bytes, respectively. Processes make allocation requests for 100, 50 and 400 bytes. The first fit technique will allocate 100 and 50 bytes from the first free area leaving a free area of 50 bytes. It allocates 400 bytes from the third free area. The best fit technique will allocate 100 and 50 bytes from the second free area leaving a free area of 20 bytes. The next fit technique allocates 100, 50 and 400 bytes from the three free areas.

Knuth (1973) presents experimental data and concludes that first fit is superior to best fit in practice. Both first fit and next fit perform better than best fit. However, next fit tends to split *all* free areas if the system has been in operation long enough, whereas first fit may not split the last few free areas. This feature facilitates allocation of large memory areas when the first fit technique is used.

Memory fragmentation

Definition 5.2 (Memory fragmentation) Memory fragmentation is the existence of unusable areas in the memory of a computer system.

Table 5.2 describes two aspects of memory fragmentation. *External fragmentation* occurs if a memory area remains unused because it cannot be allocated. *Internal fragmentation* occurs if a process is allocated more memory than it needs. The balance of memory remains unused. In Figure 5.8(c), best fit allocation creates a free area of 20 bytes, which is too small to be allocated. This is an example of external fragmentation. We would have internal fragmentation if an allocator were to allocate 70 bytes of memory when a process requests 50 bytes.

Table 5.2 Forms of memory fragmentation

Form of fragmentation	Description
External fragmentation	Some area of memory is too small to be allocated.
Internal fragmentation	More memory is allocated than requested by a process. Hence some allocated memory remains unused.

Memory fragmentation results in poor utilization of memory. In this Section, and in the remainder of this Chapter, we discuss several techniques to avoid or minimize memory fragmentation.

Merging free areas

External fragmentation can be countered by merging free areas to form larger free areas. Merging can be attempted every time an area is added to the free list. A simple method would be to search the free list to check whether any area adjoining the new area is already in the free list. If so, it can be deleted from the free list, merged with the new area and the new area can be added to the list. However, this method is expensive because it involves a search of the free list every time a new area is to be added to it. We describe two generic techniques that perform merging more efficiently. Section 5.4.1 discusses a memory allocator that uses a special merging technique.

Boundary tags A *tag* is a status descriptor for a memory area. It consists of an ordered pair (allocation status, size). Two tags containing identical information are used per memory area. These are stored at the start and end of the area, i.e., in the first and last few bytes of the area. Thus every allocated or free area of memory contains tags near its boundaries. If an area is free, the free list pointer follows the tag at its starting boundary. Figure 5.9 shows this arrangement.

When an area becomes free, we check the boundary tags of its neighboring areas. These tags are easy to find because they immediately precede and follow boundaries of the newly freed area. If any of the neighbors is free, it is merged with the newly freed area. Figure 5.10 shows the three possibilities in merging. If only the left neighbor is free, the newly freed area is merged with it. Boundary tags are now set for the new area. The left neighbor already existed in the free list (see Figure 5.10(b)), hence it is enough to simply change its size field. If only the right neighbor is free, the newly freed area is merged with it and boundary tags are set for the merged area. Now the free list has to be modified to remove the entry for the right neighbor and add an entry for the merged area (see Figure 5.10(c)). If both neighbors are free, the newly freed area is merged with both of them to form a single free area. The size field of the left neighbor's entry in the free list is modified to reflect the merging. The right neighbor also had an entry in the free list. The free list is modified to simply

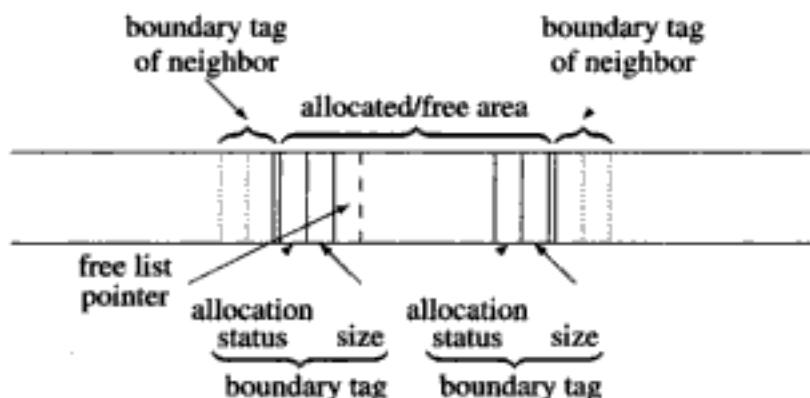


Fig. 5.9 Boundary tags and free area pointer

delete this entry (see Figure 5.10(d)). Maintaining the free list as a doubly linked list would enable this operation to be performed efficiently.

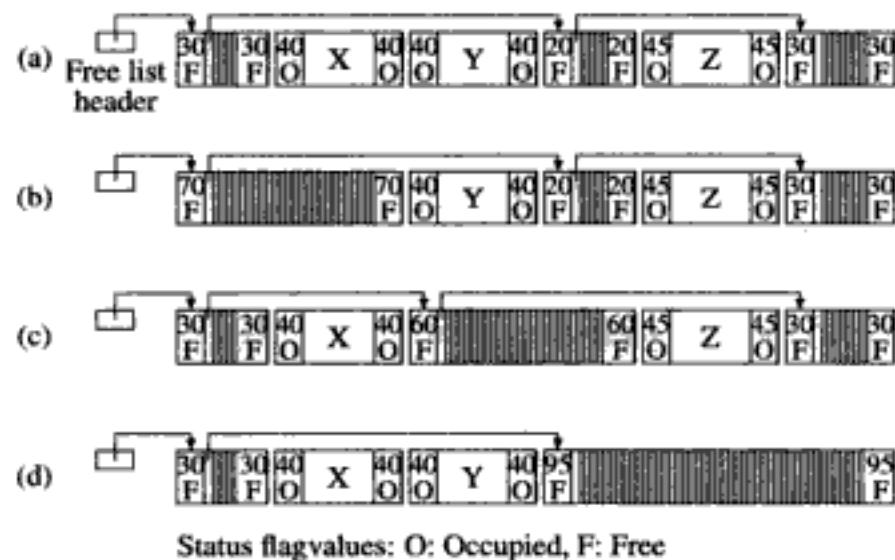


Fig. 5.10 Merging using boundary tags (a) Free list, (b)–(d) freeing of areas X, Y and Z

A relation called the *fivey percent rule* holds when we use this method of merging. When an area of memory is freed, the total number of free areas in the system increases by 1, decreases by 1 or remains the same depending on whether the area being freed has zero, two or one free areas as neighbors. These areas are shown as areas of type C, A and B, respectively, in the following:



When an allocation is made the number of free areas reduces by 1 if the requested size matches the size of some free area; otherwise, it remains unchanged since

remaining free area would be returned to the free list. The latter case is far more probable than the former. Assuming a large memory so that the situation at both ends of memory can be ignored, and assuming that each area is equally likely to be released, we have

$$\text{Number of allocated areas, } n = \#A + \#B + \#C$$

$$\text{Number of free areas, } m = \frac{1}{2}(2 \times \#A + \#B)$$

where $\#A$ is the number of free areas of type A, etc. In the steady state $\#A = \#C$, so $m = \frac{1}{2}n$. This relation is called the fifty percent rule.

The fifty percent rule helps in estimating of the size of the free list and the effort involved in an allocation method like the best fit method that requires the entire free list to be processed. It also gives us a method of estimating the free area in memory at any time. If s_f is the average size of free areas of memory, the total free memory is $s_f \times \frac{n}{2}$.

Memory compaction In this approach memory bindings are changed such that all free areas can be merged to form a single free area. As the name suggests, this is achieved by ‘packing’ all allocated areas towards one end of the memory. Figure 5.11 illustrates use of compaction to merge free areas.

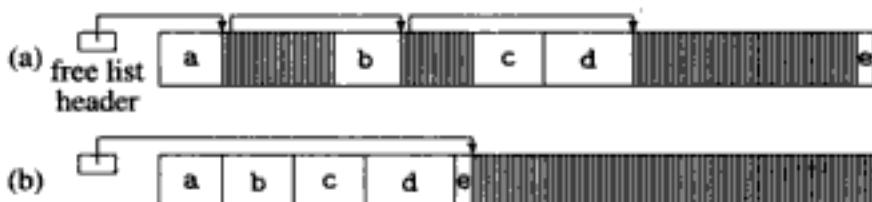


Fig. 5.11 Memory compaction

Compaction is more involved than suggested by this discussion because it involves movement of code and data in memory. If area b in Figure 5.11 contains a process, it needs to be relocated to execute correctly from the new memory area. This would involve modification of operand addresses in its instructions. If the computer system provides a relocation register, relocation can be achieved by simply changing the address in the relocation register (see Section 5.3.3). Otherwise, relocation would have to be performed by software means. This is not practical because all addresses used by a process, including addresses of heap allocated data and addresses contained in CPU registers, would need to be modified. Hence compaction is performed only if a computer system provides a relocation register or an equivalent arrangement.

5.4.1 Buddy System and Powers-of-Two Allocators

The buddy system and powers-of-two allocators perform allocation of memory in blocks of a few standard sizes—the size of each memory block is a power of two. This restriction reduces the effort involved in allocation and merging of blocks, and

results in fast allocation and deallocation. However, it leads to internal fragmentation unless a process requests an area of memory whose size is a power of two.

Buddy system allocator A buddy system splits and recombines memory blocks in a pre-determined manner during allocation and deallocation. Blocks created by splitting a block are called *buddy blocks*. Free buddy blocks are merged to form the block that was split to create them. This operation is called *coalescing*. Under this system, adjoining free blocks that are not buddies are not coalesced. The binary buddy system, which we describe here, splits a block into two equal sized buddies. Thus each block b has a single buddy block that either precedes b in memory, or follows b in memory. Memory block sizes are 2^n for different values of $n \geq t$, where t is some threshold value. This restriction ensures that memory blocks are not meaninglessly small in size.

The buddy system allocator associates a 1 bit tag with each block to indicate whether the block is *allocated* or *free*. The tag of a block may be located in the block itself, or it may be stored separately. The allocator maintains many lists of free blocks; each free list is maintained as a doubly linked list and consists of free blocks of identical size, i.e., blocks of size 2^k for some $k \geq t$. Operation of the allocator starts with a single free memory block of size 2^z , for some $z > t$. It is entered in the free list for blocks of size 2^z . The following actions are performed when a process requests a memory block of size m . The system finds the smallest power of 2 that is $\geq m$. Let this be 2^l . If the list of blocks with size 2^l is not empty, it allocates the first block from the list to the process and changes the tag of the block from *free* to *allocated*. If the list is empty, it checks the list for blocks of size 2^{l+1} . It takes one block off this list, and splits it into two halves of size 2^l . These blocks become buddies. It puts one of these blocks into the free list for blocks of size 2^l and uses the other block to satisfy the request. If a block of size 2^{l+1} is not available, it looks into the list for blocks of size 2^{l+2} . After splitting, one of the blocks would be put into the free list for blocks of size 2^{l+1} and the other block would be split further for allocation as described before. If a block of size 2^{l+2} is not available, it looks into the list of blocks of size 2^{l+3} , and so on. Thus, several splits may have to be performed before a request can be satisfied.

When a process frees a memory block of size 2^l , the buddy system changes the tag of the block to *free* and checks the tag of its buddy block to see whether the buddy block is also free. If so, it merges these two blocks into a single block of size 2^{l+1} . It now repeats the coalescing check transitively, i.e., it checks whether the buddy of this new block of size 2^{l+1} is free, and so on. It enters a block in a free list only when it finds that its buddy block is not free.

Example 5.5 Figure 5.12 illustrates operation of a binary buddy system. Parts (a) and (b) of the figure show the status of the system before and after the block marked with the down-arrow symbol ' \downarrow ' is released by a process. For simplicity, in each part we show two views of the system. The top half shows the free lists while the bottom half shows the layout of memory. For ease of reference, corresponding blocks in the

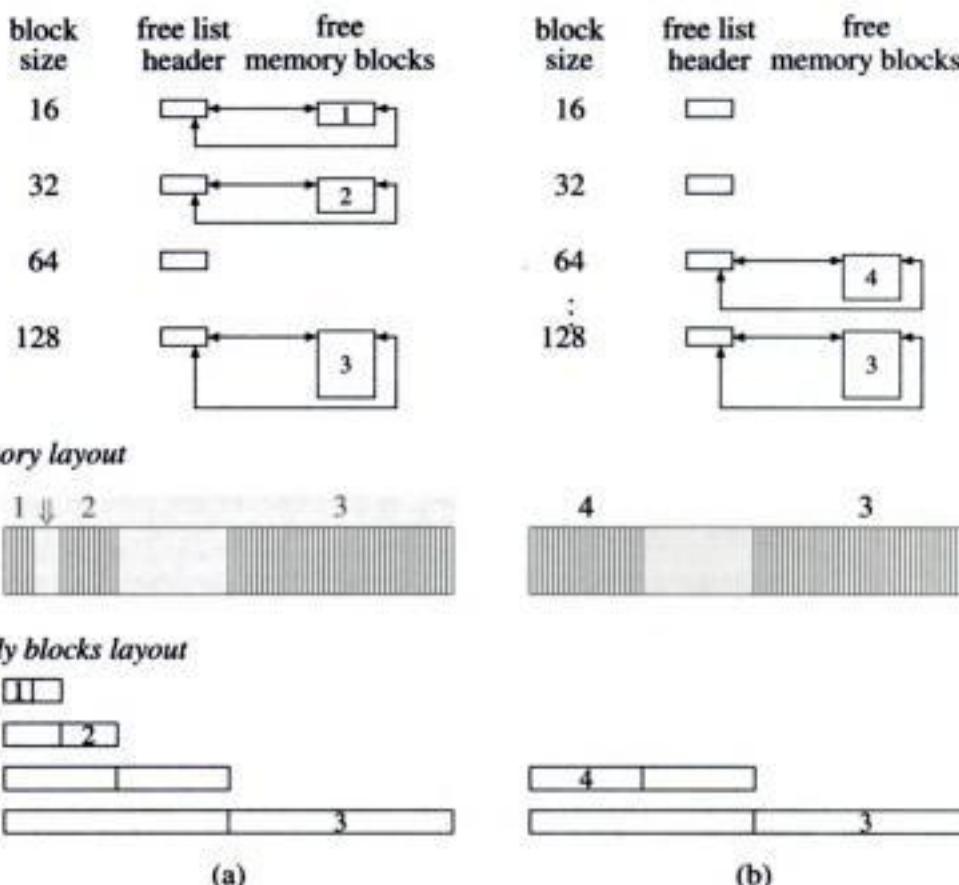


Fig. 5.12 Buddy system operation when a block is released

two halves carry identical numbers. The block being released has a size of 16 bytes. Its buddy is the free block numbered 1 in Figure 5.12(a), and so the buddy system allocator merges these two blocks to form a new block. The buddy of this new block is block 2, and so the new block and block 2 are merged to form a free block of size 64 bytes. This free block is numbered 4 in Figure 5.12(b). It is now entered in the appropriate free list.

The check for a buddy's tag can be performed efficiently because block sizes are powers of 2. Let the block being freed have a size of 16 bytes. Since 16 is 2^4 , its address is of the form ...y0000, where four 0s follow y, and y is 0 or 1. Its buddy block has the address ...z0000 where z = 1 - y. This address can be obtained simply by performing an exclusive or operation with a number...10000, i.e., with 2^4 . For example, if the address of a block is 101010000, its buddy's address is 101000000. In general, address of the buddy of a block of size 2^n bytes can be found by performing exclusive or with 2^n .

Powers-of-two allocator As in the buddy system, sizes of memory blocks are powers of 2, and separate free lists are maintained for blocks of different sizes. However, similarity with the buddy system ends here. Each block contains a header

element that is used for two purposes. It contains a status flag to indicate whether the block is currently allocated or free. If a block is free, another field in the header contains size of the block. If a block is allocated, the other field in the header contains address of the free list to which it should be added when it becomes free (see Figure 5.13).

status	size/address
<i>free or allocated</i>	<i>size of block or address of free list</i>

Fig. 5.13 Header element in the powers-of-two allocator

When a request is made for n bytes, the allocator finds the smallest free block that is large enough to hold n bytes. It first checks the free list containing blocks whose size is the smallest value of x such that $2^x \geq n$. If this free list is empty, it checks the list containing blocks that are the next higher power of 2 in size, and so on. An entire block is allocated to a request, i.e., no splitting of blocks takes place. Also, no effort is made to coalesce adjoining blocks to form larger blocks; when released, a block is simply returned to its free list.

System operation starts by forming blocks of desired size and entering them into the appropriate free lists. New blocks can be created dynamically whenever the allocator runs out of blocks of a given size, or when no block can be allocated to a request.

Comparison of memory allocators Memory allocators can be compared on the basis of speed of allocation and efficient use of memory. The buddy and powers-of-two allocators are superior to the first-fit, best-fit or next-fit allocators in terms of allocation speed because they avoid searches in free lists. The powers-of-two allocator is faster than the buddy allocator because it does not need to perform splitting and merging.

Memory utilization can be compared by computing a memory utilization factor as follows

$$\text{Memory utilization factor} = \frac{\text{memory in use}}{\text{total memory committed}}$$

where *memory in use* is the amount of memory in use by requesting processes, and *total memory committed* includes allocated memory, free memory existing with the memory allocator and the memory occupied by its own data structures. The largest value of the utilization factor depicts the best case performance of a system and the smallest value depicts the worst case performance. It can be seen that the buddy and power-of-two allocators do not fare well in terms of the utilization factor because they allocate blocks whose sizes are powers of 2. Internal fragmentation results unless memory requests match block sizes. These allocators also use up additional memory to store the list headers. The buddy system allocator also requires memory

to store tags.

A Powers-of-two allocator fails to satisfy a request if a sufficiently large free block does not exist. Since it does not merge free blocks into larger blocks, this can happen even when free contiguous blocks of smaller size could have been combined to satisfy the request. In a buddy system this would happen only if adjoining free blocks are not buddies. This is rare in practice. In fact, Knuth (1973) reports that in simulation studies a buddy allocator was able to achieve 95 percent memory utilization before failing to satisfy a request.

Powers-of-two allocators suffer from another drawback. While the size of a block is a power of two, the header element cannot be used by a process to which the block is allocated. Thus the useful portion of a block is somewhat smaller than a power of 2. If a memory request is for an area that is exactly a power of 2 in size, this method uses up at least twice that amount of memory. Allocators using the first-fit or best-fit techniques provide better memory utilization since no part of an allocated block is wasted. However, first-fit, best-fit and next-fit allocators suffer from external fragmentation since free blocks may be too small to satisfy any requests.

5.5 CONTIGUOUS MEMORY ALLOCATION

Contiguous memory allocation is the classical memory allocation model in which each process is allocated a single contiguous area in memory. In early computer systems the memory allocation decision was made *statically*, i.e., *before* the execution of a process began. The OS determined the memory required to execute a process and allocated sufficient memory to it.

Practical issues in contiguous memory allocation are

1. Memory protection
2. Static and dynamic relocation of a program to execute from the memory area allocated to it
3. Measures to avoid memory fragmentation.

Of these, memory protection and static relocation have been discussed earlier in Sections 5.3.4 and 5.3.3. In this Section we focus on practical techniques of contiguous memory allocation that tackle the memory fragmentation problem.

5.5.1 Handling Memory Fragmentation

There are two facets to memory fragmentation

- *External fragmentation* arises when free memory areas existing in a system are too small to be allocated to processes. The kernel is aware of external fragmentation, and may be able to take some actions to relieve it.
- *Internal fragmentation* exists when the memory allocated to a process is not fully utilized by it. Internal fragmentation arises if the kernel allocates a memory area of a standard size to a process irrespective of its request.

We discuss two techniques used to overcome the problem of external fragmentation—memory compaction and reuse of memory fragments.

Memory compaction The kernel can periodically perform memory compaction to eliminate external fragmentation. This action produces a single unused area in memory. This area may be large enough to accommodate one or more new processes even in cases where individual free areas before compaction were too small for that purpose. Example 5.6 illustrates use of memory compaction.

Example 5.6 Processes A, B, C and D exist in memory (see Figure 5.14(a)). Two free memory areas exist when B terminates, however none of them is large enough to accommodate another process (see Figure 5.14(b)). The kernel performs compaction to create a single free area and initiates process E in this area (see Figure 5.14(c)).

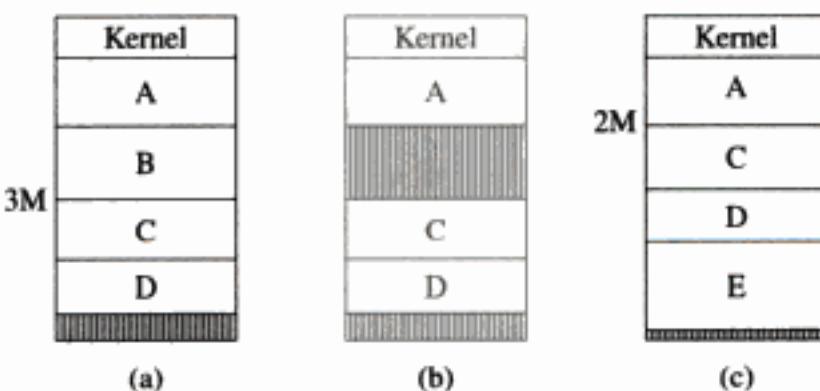


Fig. 5.14 Memory compaction

Compaction involves movement of processes in the memory during their execution. This action involves *dynamic relocation* of a program. It can be achieved quite simply in computer systems using a relocation register (see Section 5.3.3). Consider process C of Figure 5.14(a). If the assumed start address of its program was 0, relocation register should contain the address 3 M bytes when C is in execution. During compaction, C is moved to the memory area with start address 2 M bytes. To implement this dynamic relocation, relocation register would be loaded with the address 2 M whenever program C is scheduled in Figure 5.14(c). Dynamic relocation is not practical if the computer system does not contain a relocation register. In such cases, the kernel must resort to reuse of free memory areas.

Reuse of memory areas Reuse of free memory areas avoids the overhead of program relocation, and also does not require special hardware provisions like a relocation register. However, it may cause delays in initiating execution of programs. In Example 5.6, memory allocation by reuse would delay initiation of process E until the memory contains a free area that is large enough to accommodate it.

5.5.2 Swapping

Motivation and the basic mechanism of swapping was described in Section 2.6.3. A time sharing system swaps out a process that is not in the *running* state to make space for another process. While swapping out the process, its code and data space is written to a *swapping area* on the disk. The swapped out process is brought back into memory before it is due to resume its execution.

A basic issue while swapping in a process is this: Should the process be allocated the same memory area that it occupied before it was swapped out? If so, its swapping-in depends on swapping-out of some other process that may have been allocated that memory area in the mean time. Ability to place the swapped in process elsewhere in memory would be useful, however it requires relocation of the process to execute from a new memory area. As mentioned in Section 5.5.1, only computer systems that provide a relocation register can achieve this easily.

5.6 NONCONTIGUOUS MEMORY ALLOCATION

In the noncontiguous memory allocation model, several non-adjacent memory areas are allocated to a process. None of these areas is large enough to hold the complete process, hence a part of the process (including its data and stack) is loaded in each of these areas. We call each such part a *component* of a process.

The noncontiguous memory allocation model provides two important advantages. A memory area that is not large enough to hold a complete process can still be used, so less external fragmentation results. (In fact, as discussed later, no external fragmentation exists when paging is used.) This feature reduces external fragmentation and improves memory utilization; consequently, it may not be necessary to perform either merging of free memory areas or compaction. Example 5.7 illustrates this model.

Example 5.7 Four unallocated memory areas of 50K, 30K, 80K and 60K bytes exist in the memory as shown in Figure 5.15(a). A process P of size 140K bytes is to be initiated. It is split into three components that we will call P-1, P-2 and P-3 and loaded into three of the free areas as follows (see Figure 5.15(b)):

<i>process component</i>	<i>size</i>	<i>memory start address</i>
P-1	50K	100K
P-2	30K	300K
P-3	60K	450K

For simplicity, we assume 1K to be 1000 bytes. Two free memory areas of size 20K and 60K bytes now exist following component P-3 and process D, respectively. These areas could be utilized to initiate another process of size \leq 80K bytes.

It is not enough to load a process into noncontiguous areas of memory—it has to be able to execute correctly. The program of process P of Example 5.7 would have been compiled and linked assuming the classical contiguous memory allocation

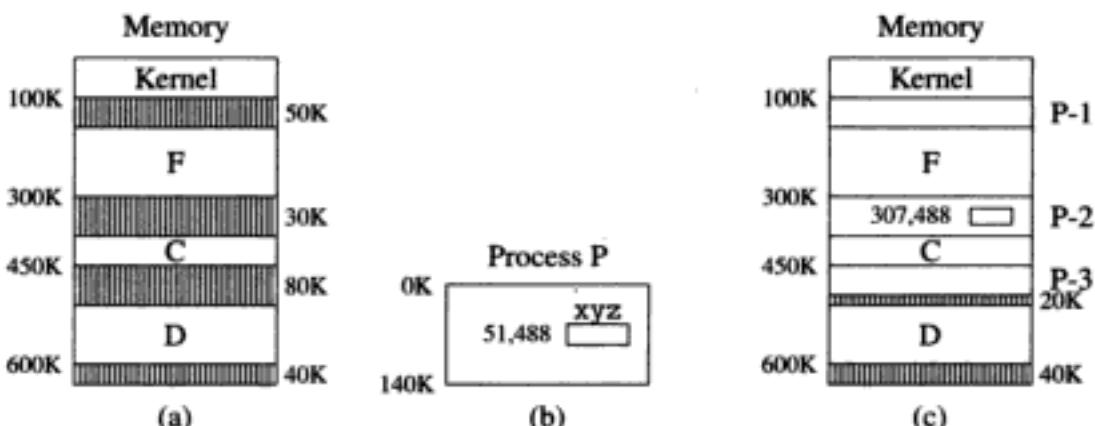


Fig. 5.15 Noncontiguous loading of process P

model (see Figure 5.4). So how can it execute when noncontiguous allocation is performed? We discuss this issue using the notion of logical and physical organization.

Logical and physical organization In Section 1.1, we mentioned that the abstract view of an entity is called the *logical view* and the arrangement and relationship between components of the entity is called the *logical organization*. The real view of an entity is called the *physical view* and the arrangement depicted in it is called the *physical organization*. Figure 5.15 illustrates the logical and physical views of process P of Example 5.7. In the logical view, P is a contiguous entity with the start of address 0K. In the physical view, P is actually loaded in 3 non-adjacent areas with the start addresses 100K, 300K and 450K bytes, respectively.

Example 5.8 In Example 5.7, the logical address space of P extends from 0 to 140K, while the physical address space extends from 0 to 640K. Data area xyz in the program of process P has the address 51,488 (see Figure 5.15(b)). This is the logical address of xyz. The process component P-1 in Figure 5.15 has a size of 50K bytes, i.e., 51,200 bytes, so xyz is situated in component P-2 at an offset of 288 bytes. Since P-2 is loaded in the memory area with the start address 300K bytes, i.e., 307,200 bytes, the physical address of xyz is 307,488 (see Figure 5.15(c)).

The schematic diagram of Figure 5.16 shows how the CPU obtains the physical address that corresponds to a logical address. The kernel stores information about the memory areas allocated to process P in a table and makes it available to the *memory management unit* (MMU). In Example 5.7, this information would consist of the sizes and memory start addresses of P-1, P-2 and P-3. The CPU sends the logical address of each data or instruction used in the process to the MMU, and the MMU uses the memory allocation information stored in the table to compute the corresponding physical address. This address is called the *effective memory address* of the data or instruction. The procedure of computing the effective memory address from a logical address is called *address translation*.

Logical addresses, physical addresses and address translation A logical address is

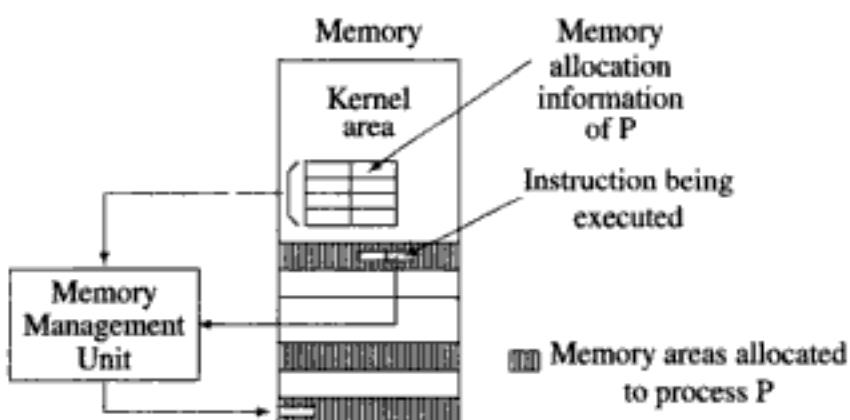


Fig. 5.16 A schematic of address translation in noncontiguous memory allocation

the address of an instruction or data byte as used in a process. (This address may be obtained using index, base or segment registers.) The set of logical addresses used by a process constitute the *logical address space* of the process. A *physical address* is the effective memory address of an instruction or data byte. The set of physical addresses in the system constitute the *physical address space* of the system.

A logical address is considered to consist of two parts—the id of the process component containing the address, and the id of the byte within the component. We represent each address by a pair of the form

$$(comp_i, byte_i).$$

The MMU computes the address where a byte $(comp_i, byte_i)$ exists in memory using the formula

$$\begin{aligned} \text{Effective memory address of } (comp_i, byte_i) \\ = & \text{ start address of memory area allocated to } comp_i \\ & + \text{ offset of } byte_i \text{ within } comp_i \end{aligned} \quad (5.1)$$

To facilitate address translation by the MMU, the kernel provides the memory allocation information concerning process P to the MMU when it schedules P. In Example 5.7, this information would consist of the sizes and memory start addresses of P-1, P-2 and P-3.

Implementation of noncontiguous memory allocation Two approaches are used to implement noncontiguous memory allocation:

- Paging
- Segmentation.

In paging, each process consists of fixed sized components called *pages*. The size of a page is specified in the architecture of the computer system. The memory

can accommodate an integral number of pages in it. Memory allocation is performed with a page as a unit—a memory area whose size is the same as the page size is allocated to it. Since all pages are of the same size, no unallocated memory area can be smaller than a page in size. Consequently, external fragmentation does not arise in the system.

In segmentation, a programmer identifies components called *segments* in a process. A segment is a logical entity in a program, e.g., a function, a data structure or an object. This arrangement facilitates sharing of code, data and program modules. Segments can have different sizes, so memory management is cumbersome and external fragmentation can exist.

We discuss paging and segmentation in Sections 5.7 and 5.8, respectively.

Memory protection All memory areas allocated to a process need to be protected against interference by other processes. This can be achieved using the memory protection schemes described in Section 5.3.4. The scheme based on memory protection keys can be used with a minor modification—whenever a component of a process is loaded into an area of memory, the protection key of all blocks in that area of memory would be set to the APK of the process. While executing the process, the CPU would detect a memory protection violation if the APK of the process does not match the protection key of a memory byte accessed by it.

Use of the bound register scheme for memory protection requires an elaborate arrangement. The scheme has to be used individually for each memory area allocated to a process, so it is implemented by the MMU. While performing address translation for a logical address ($comp_i, byte_i$), the MMU checks whether $comp_i$ exists in the program of the process and whether $byte_i$ exists in $comp_i$. A protection violation interrupt is raised if any of these checks fail. The bounds check can be simplified in paging because a logical address has just enough bits to accommodate the offset of the last byte in a page. Thus, it is not necessary to check whether $byte_i$ exists in $comp_i$.

The bounds check schematic of memory protection is attractive because it permits processes to possess different kinds of access privileges to a shared component. For example, one process may possess a read privilege to a data segment while another process possesses a read/write privilege to it. This feature is particularly meaningful in segmentation because each segment is a logical entity in a program.

5.6.1 Comparison of Contiguous and Noncontiguous Allocation

Table 5.3 summarizes key features of contiguous and noncontiguous memory allocation. In contiguous allocation, a single area of memory is allocated to a process using a first-fit, best-fit or next-fit approach. In noncontiguous allocation, memory allocation is performed for each part of a process. In paging, all process parts are of the same size, so memory can be partitioned into areas of equal size and a pool-based allocation strategy can be used. This approach reduces allocation overhead. It also

eliminates external fragmentation of memory; however, internal fragmentation may exist in the last page of a process. Systems using segmentation share some drawbacks of contiguous allocation. Segments are not of the same size, hence external fragmentation cannot be avoided.

Swapping is more effective in noncontiguous memory allocation because address translation enables a swapped-in process to execute from any part of memory. Protection is also more effective because processes can possess different kinds of access privileges to a shared component.

Table 5.3 Comparison of contiguous and noncontiguous memory allocation

Feature	Contiguous allocation	Noncontiguous allocation
Overhead	No overhead during execution of a program.	Address translation is performed during program execution.
Allocation	Allocates a single area of memory	Allocates several memory areas—one memory area to each component of a process.
Reuse of memory	Internal fragmentation exists in partitioned allocation. External fragmentation exists in first-fit/best-fit/next-fit allocation.	In paging : No external fragmentation, but internal fragmentation exists. In segmentation : External fragmentation exists, but no internal fragmentation.
Swapping	Unless the computer system provides a relocation register, a swapped-in process must be placed in its originally allocated area.	Swapped-in process can be placed anywhere in memory.

5.7 PAGING

In a system using paging, a process is considered to be a contiguous entity containing instructions and data. In the logical view, a process consists of a linear arrangement of pages. Each page has s bytes in it, where s is a power of 2. The value of s is specified in the architecture of the computer system. Processes use numeric logical addresses. The computer hardware decomposes a logical address into the pair (p_i, b_i) , where p_i is the page number and b_i is an offset in p_i , $0 \leq b_i < s$. The physical view consists of non-adjacent areas of memory allocated to pages of the process.

Figure 5.17 illustrates the arrangement used to execute processes P and R in a system using paging. The size of P is 5500 bytes. The page size is 1K bytes, hence P has 6 pages. These pages are numbered from 0 to 5 and bytes in a page are numbered from 0 to 1023. The last page contains only 380 bytes. If a data item `sample` has the address 5248, the computer hardware would view its address as the pair (5, 128). Process R has 3 pages, numbered from 0 to 2.

The kernel partitions the memory into areas called *page frames*. Each page frame

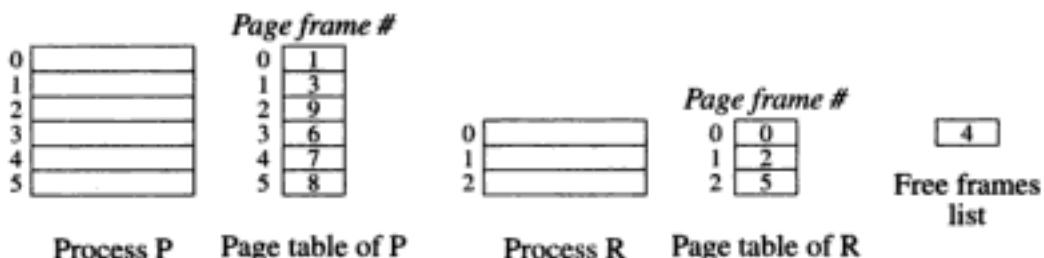


Fig. 5.17 Processes in paging

is the same size as a page, i.e., 1K bytes. In Figure 5.17, the computer has a memory of 10K bytes, hence page frames are numbered from 0 to 9. At any moment, some page frames are allocated to program pages and others are free. The kernel maintains a list called the *free frames list* to note the ids of free page frames. At the moment, only page frame 4 is free.

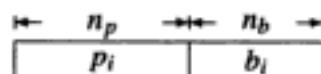
While loading process P for execution, the kernel consults the frame table and allocates a free page frame to each page of the process. To facilitate address translation, a *page table* (PT) is constructed for the process. Each entry of the page table indicates the page frame allocated to a page of the process. This table is indexed using a page number. During execution of the process, the MMU consults the page table to perform address translation.

In Figure 5.17, the frame table indicates that six page frames are occupied by process P, three page frames are occupied by a process R, and one page frame is free. The page table of P indicates frame numbers allocated to pages of P. The logical address (5, 128) would be translated into the physical address 8320 using the entry for page 5 in the page table.

Address translation We use the following notation:

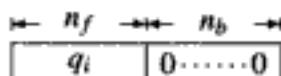
- s : Size of a page
- l_l : Length of a logical address (i.e., number of bits in it)
- l_h : Length of a physical address
- n_b : Number of bits required to access the last byte in a page
- n_p : Number of bits used to contain the page number in a logical address
- n_f : Number of bits used to contain the frame number in a physical address

The size of a page, s , is a power of 2; say, $s = 2^{n_b}$. Hence the least significant n_b bits in a logical address give us b_i . The remaining bits in a logical address form p_i . We can obtain the values of p_i and b_i simply by grouping the bits of a logical address as follows:

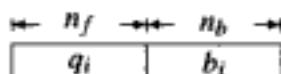


where $n_p = l_l - n_b$. Use of a power of 2 as the page size similarly simplifies construction of the effective memory address. Let page p_i be allocated page frame q_i .

Since pages and page frames have identical sizes, n_b bits are also needed to address the bytes in a page frame. Let $n_f = l_p - n_b$. Physical address of byte '0' of page frame q_i is



Hence physical address of byte b_i in page frame q_i is given by



The MMU can obtain this address simply by concatenating q_i and b_i to obtain an l_h bit number. Example 5.9 illustrates address translation.

Example 5.9 A computer system uses 32 bit logical addresses and a page size of 4K bytes. 12 bits are adequate to address the bytes in a page. Thus, the higher order 20 bits in a logical address represented p_i and the 12 lower order bits represented b_i . For a memory size of 256M bytes, $l_p = 28$. Thus, the higher order 16 bits in a physical address represent q_i . If page 130 exists in page frame 48, $p_i = 130$, and $q_i = 48$. If $b_i = 600$, the logical and physical addresses look as follows:

Logical address	Physical address
← 20 → 12 → 0 ... 010000010 001001011000	← 16 → 12 → 0 ... 00110000 001001011000

During address translation, the MMU obtains p_i and b_i merely by grouping the bits as shown above. The 130th entry of the page table is now accessed to obtain q_i , which is 48. This number is concatenated with b_i to form the physical address.

5.8 SEGMENTATION

A segment is a logical entity in a program, e.g., a function, a data structure, or an object. Hence it is meaningful to manage a segment as a unit—load it into the memory for execution or share it with other programs. In the logical view, a process consists of a set of segments. The physical view consists of non-adjacent areas of memory allocated to segments.

A process Q consists of four logical entities with the symbolic names `main`, `database`, `search` and `retrieve`. While coding the program, the programmer declares these as four segments in Q. The compiler/assembler generates logical addresses while translating the program. Each logical address used in Q has the form (s_i, b_i) where s_i and b_i are the id's of a segment and a byte within a segment. For example, the instruction corresponding to a statement `call get_sample`, where `get_sample` is a procedure in segment `retrieve`, may use the operand address `(retrieve, get_sample)`, or may use a numeric representation for s_i and b_i .

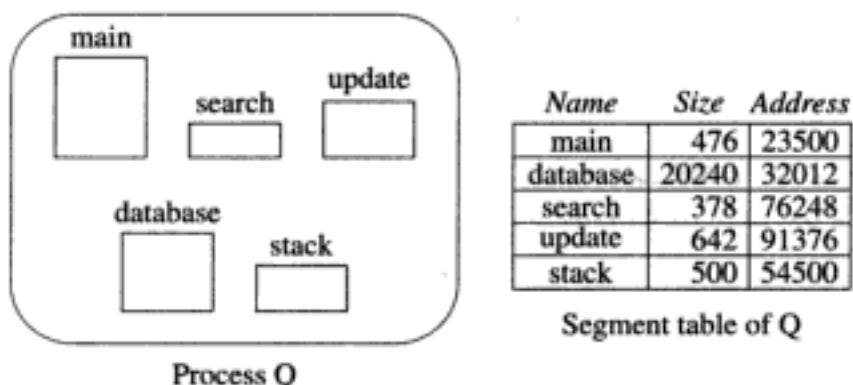


Fig. 5.18 A process Q in segmentation

Figure 5.18 shows how the kernel handles process Q. The left half of the figure shows the logical view of process Q. To facilitate address translation, the kernel constructs a segment table for Q. Each entry in this table shows the size of a segment of Q and address of the memory area allocated to it. The MMU uses the segment table to perform address translation. Segments do not have standard sizes, hence the simplification of bit concatenation used in paging is not applicable. Calculation of the effective memory address therefore involves addition of b_i to the start address of s_i according to Eq. (5.1), so address translation is slower than in paging. In Figure 5.18, if `get_sample` has the offset 232 in segment `retrieve`, address translation of `(retrieve, get_sample)` would yield the address $91376 + 232 = 91608$.

Memory allocation for each segment is performed as in the contiguous memory allocation model. The kernel keeps a free list of memory areas. While loading a process, it searches through this list to perform first fit or best fit allocation to each segment of the process. When a process terminates, the memory areas allocated to its segments are added to the free list. External fragmentation exists because segment sizes vary.

The task of putting each logical address in the form (s_i, b_i) is performed by a compiler or assembler. While compiling a reference to a symbol xyz, it has to decide which segment xyz belongs to.

5.9 SEGMENTATION WITH PAGING

In this approach, each segment in a program is paged separately. Accordingly, an integral number of pages is allocated to each segment. This approach simplifies memory allocation and speeds it up, and also avoids external fragmentation. A page table is constructed for each segment, and a pointer to the page table is kept in the `segm_cnt`'s entry in the segment table. Address translation for a logical address (s_i, b_i) is now done in two stages. In the first stage, the entry of s_i is located in the segment table, and the address of its page table is obtained. The byte number b_i is now split into a pair (ps_i, bp_i) , where ps_i is the page number in segment s_i , and bp_i is the byte number in page p_i . The effective address calculation is now completed as in paging.

i.e., the frame number of ps_i is obtained and bp_i is concatenated with it to obtain the effective address.

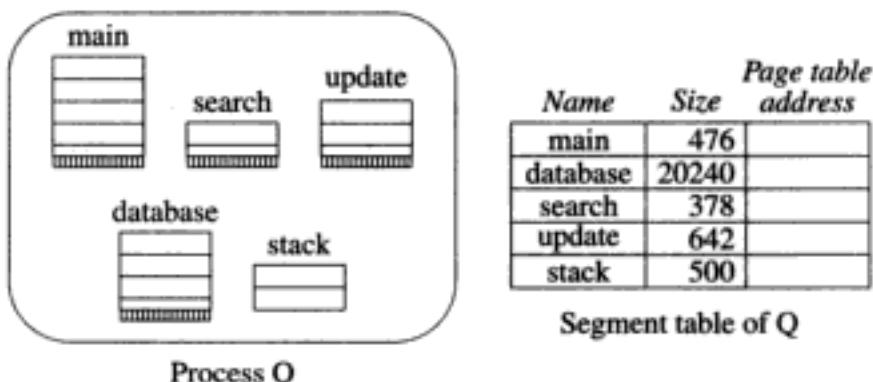


Fig. 5.19 A process Q in segmentation and paging

Figure 5.19 shows process Q of Figure 5.18 in a system using segmentation with paging. Each segment is paged independently, so internal fragmentation exists in the last page of each segment. Each segment table entry now contains a pointer to the page table of the segment. The size field in a segment's entry is used to facilitate a bound check for memory protection.

5.10 KERNEL MEMORY ALLOCATION

The kernel creates and destroys many data structures during its operation. These are mostly *control blocks* that are used to control the allocation and use of resources in the system. Some examples of control blocks are: The process control block (PCB) created for every process, event control block (ECB) created whenever the occurrence of an event is anticipated, I/O control block (IOCB) created for an I/O operation and file control block (FCB) created for every open file. Memory and CPU time used by the kernel constitute overhead. Hence creation and destruction of data structures by the kernel must be fast, and the kernel must also make efficient use of memory.

Lifetimes of kernel data structures are tied to lifetimes of related entities like processes or related activities like I/O operations, hence they do not possess any predictable relationships with one another. This feature rules out use of the kernel stack for creating these data structures. The kernel must use a heap. Sizes of control blocks are known while designing the OS. The kernel uses this feature to make memory allocation simple and efficient—memory released when one control block is destroyed can be allocated to create a similar control block in future. A separate free list can be maintained for each type of control block to realize this advantage.

Kernel memory allocators in Unix and Solaris

Unix and Solaris systems use noncontiguous memory allocation with paging, so memory allocators use an integral number of pages and make special efforts to use each page effectively. Three kernel memory allocators are discussed in this section:

- McKusick–Karels allocator
- Lazy buddy allocator
- Slab allocator.

The McKusick–Karels and lazy buddy allocators allocate memory areas that are powers of 2 in size. These allocators, and other powers-of-two allocators, face a peculiar problem related to cache performance. When an object is accessed, it is loaded into the processor cache. A performance problem arises because some parts of an object are accessed more frequently than others. Because of address alignment on a boundary that is a power of 2, the frequently accessed parts of objects are mapped into the same areas of the cache. Hence some parts of the cache face a lot of contention while others do not. This lopsided cache contention leads to poor performance of the cache. The slab allocator uses an interesting technique to avoid this cache performance problem.

Descriptions of these three allocators follow. In interest of consistency with our previous discussion, we use slightly different terminology than used in the literature dealing with these allocators.

McKusick–Karels allocator This modified powers-of-two allocator is used in Unix 4.4 BSD. The allocator uses an integral number of pages. The basic operating principle is to divide each page into blocks of equal size and store the size information against the logical address of a page. This principle helps to eliminate the header element used in the powers-of-two allocators to store size of a block or address of a free list (see Fig 5.13). This can be explained as follows: Since all blocks in a page are of the same size, the size of a block can be found by finding the address of the page in which it is located. Thus, the size need not be stored in each free block. While freeing a block, the free list to which the block should be added can also be found by finding the address of the page in which it is located. Hence the address of the free list need not be stored in each allocated block. As a consequence of eliminating the header element, allocation is superior to the powers-of-two allocators when a memory request is for an area whose size is an exact power of 2. A block of identical size can be allocated to satisfy the request, whereas a powers-of-two allocator would have allocated a block whose size is the next power of 2.

The allocator asks for a free page when it does not find a block of the size it is looking for. This page is divided into blocks of the desired size. One of these blocks is allocated to the process making the present request and the remaining blocks are entered in the appropriate free list. If no free page exists with the allocator, it asks the paging system for a new page to be allocated to it. To ensure that it does not

consume a larger number of pages than necessary, the allocator marks a page as free when all blocks in the page are free.

The McKusick-Karels allocator achieves a higher memory utilization factor than the powers-of-two allocator because the header element is not stored in each block. However, unlike the buddy system it does not coalesce adjacent free blocks. Instead it marks a page as free when all blocks in the page become free. It also lacks a feature to return free pages to the paging system. Thus, the total number of pages allocated to the allocator is the largest number of pages it has used at any time in its life. This feature may lead to a low memory utilization factor.

Lazy buddy allocator The buddy system may perform one or more splits at every allocation and one or more coalescing actions at every release. Some of these actions may have been unnecessary because a coalesced block may be split in future. The basic design principle of the lazy buddy allocator is to delay coalescing actions whenever possible. This is done in the expectation that a data structure requiring the same amount of memory as a released block may be created in future. Such a block can be allocated without a split. Hence delaying the decision to coalesce blocks can avoid the overhead of splitting under some conditions.

The lazy buddy allocator used in Unix 5.4 works as follows: Control blocks with same size are considered to constitute a class of blocks. Coalescing decisions for a class are made on the basis of rates at which data structures of the class are created and destroyed. The allocator characterizes the behavior of the OS with respect to a class of blocks into three states called *lazy*, *reclaiming* and *accelerated*. For simplicity we refer to these as *states* of a class of blocks.

In the lazy state, allocations and releases for blocks of a class occur with matching frequencies. Hence coalesced free blocks may be split soon. Both coalescing and splitting can be avoided by delaying coalescing. In the reclaiming state, releases occur at a faster rate than allocations so it is a good idea to coalesce at every release. In the accelerated state releases occur much faster than allocations hence it is desirable to coalesce at a faster rate. The allocator should attempt to coalesce a block being released, and, additionally, it should also try to coalesce other blocks that have been released in the past.

The lazy buddy allocator maintains the free list as a doubly linked list. This way both start and end of the list can be accessed equally easily. A bit map is maintained to indicate the allocation status of blocks. In the lazy state the allocator does not perform any coalescing. A block being released is simply added to the head of the free list. No effort is made to coalesce it with its buddy. It is also not marked free in the bit map. This way the block will not be coalesced even if its buddy is released in future. Such a block is said to be *locally free*. Being at the head of the list, this block will be allocated before any other block in the list. Its allocation is efficient and fast because the bit map does not need to be updated—it still says that the block is allocated.

In the reclaiming and accelerated states a block is both added to the free list and marked free in the bit map. Such a block is said to be *globally free*. Globally free blocks are added to the end of the free list. In the reclaiming state the allocator tries to coalesce a globally free block transitively with its buddy. Eventually a block is added to some free list—either to a free list to which the block being released would have belonged, or to a free list containing larger sized blocks. Note that the block being added to a free list could be a locally free block or a globally free block depending on the state of the OS with respect to that class of blocks. In the accelerated state the allocator tries to coalesce the block being released, same as in the reclaiming state. Additionally, it also tries to coalesce one other locally free block—the block found at the start of the free list—with its buddy.

The state of a class of blocks is characterized as follows: Let A, L and G be the number of allocated, locally free and globally free blocks of a class, respectively. The total number of blocks of a class is given by $N = A + L + G$. A parameter called *slack* is computed as follows:

$$\text{slack} = N - 2 \times L - G$$

A class is said to be in the lazy, reclaiming or accelerated state if the value of *slack* is ≥ 2 , 1, or 0, respectively. (The allocator ensures that slack is never < 0 .) The coalescing overhead is different in these three states. There is no overhead in the lazy state. Hence release and allocation of blocks would be done fast, and processes would experience minimum delays in their operation due to allocation and release of data structures. In the reclaiming state the overhead would be comparable with those in the buddy system whereas in the accelerated state the overhead would be heavier than in the buddy system. It has been shown that the average delays using the lazy buddy allocator are 10 to 32 percent lower than in the case of a buddy allocator.

The implementation of the lazy buddy algorithm in Unix 5.4 uses two kinds of blocks. Small blocks vary in size between 8 and 256 bytes. Large blocks vary between 512 and 16K bytes. The allocator obtains memory from the paging system in terms of 4K byte areas. Some part of the area is used to keep the bit map of the blocks in that area. The blocks in the area are collectively called a pool. When all blocks in the pool are free, the pool is returned to the filing system. This overcomes the disadvantage seen in the case of the McKusick-Karels allocator.

Slab allocator The slab allocator of the Solaris 2.4 system has two special features:

- It uses initialized objects, which enhances object reuse efficiency
- Memory allocation is aimed at better cache behavior.

Each object is a kernel data structure. The slab allocator allocates all kernel objects of same class together in a pool. For small objects, a pool consists of many slabs and each slab contains many objects. (Large objects are not discussed here.) A

slab is organized in a standard sized area allocated by the paging system. A new slab can be created by obtaining an additional memory area from the paging system, and an unused slab can be returned to the paging system. The slabs of a pool are entered in a doubly linked list to facilitate addition and deletion of slabs. A slab may be full, partially empty, or empty depending on the number of active objects existing in it.

To facilitate searches for an empty slab, the doubly linked list containing the slabs of a pool is sorted according to the status of a slab—all full slabs are at the start of the list, partially empty slabs are in the middle and empty slabs are at the end of the list. Each slab contains a free list from which empty objects can be allocated. Each pool contains a pointer to the first slab that contains an empty object. This arrangement makes allocation efficient. When the allocator runs out of empty objects of a kind, it obtains an additional area from the paging system, constructs a new slab and enters it in the doubly linked list of slabs.

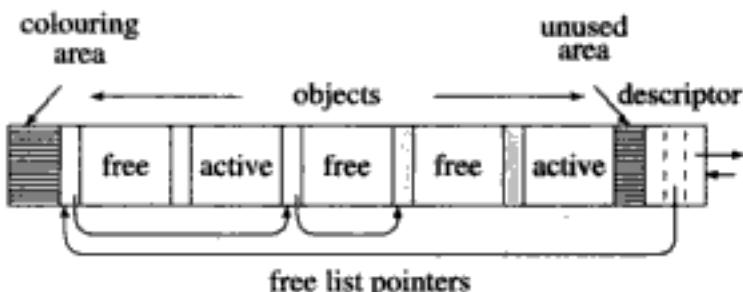


Fig. 5.20 Format of a slab

A slab contains objects of only one kind. Figure 5.20 shows the format of a slab. When the allocator obtains a memory area from the paging system, it forms a slab by formatting the area to contain an integral number of objects, a free list containing all objects and a descriptor field that contains the count of active objects and the free list header. Each object is initialized. This action involves initializing different fields in it to object-specific information like fixed strings of constant values. When allocated, the object can be used straightaway. At deallocation time, the object is brought back to its allocation time status. Since some fields of the objects never change, or change in such a manner that their values at deallocation time are the same as their values at allocation time, this approach eliminates repetitive overhead of object formation suffered in other allocators. However, use of initialized objects has some implications for the memory utilization factor. Each object needs to have a pointer field associated with it, which is used to enter it in the free list. Since an object is initialized, this pointer field must exist external to it even when the object is free (see Figure 5.20).

The slab allocator provides improved cache behavior by avoiding the cache performance problem faced by powers-of-two allocators and their variants described at the start of this Section. Each slab contains a reserved area at its start called the

coloring area (see Figure 5.20). The allocator makes the size of this coloring area different in different slabs of a pool. Consequently, objects in the same pool have different alignments with respect to the closest multiples of a power of 2. This feature is used to ensure that these objects map into different areas of the processor cache, thus avoiding excessive cache contention and improving the cache performance.

The slab allocator also provides a better memory utilization factor because it allocates only the required amount of memory for each object. Thus, unlike the McKusick-Karels and lazy buddy allocators, no internal fragmentation exists on a per object basis; only external fragmentation exists in the form of unused area in each slab. It has been found that fragmentation is only 14 percent in the slab allocator as against 45 and 46 percent in the McKusick-Karels and lazy buddy allocators, respectively. The average allocation times are also better than in the other allocators.

5.11 A REVIEW OF RELOCATION, LINKING AND PROGRAM FORMS

A program P written in some programming language L goes through several transformations before it reaches execution. Figure 5.21 contains a schematic of these transformations. Program P is translated by a translator for L (which could be a compiler or an assembler) to produce a program form called an *object module*. The object module contains the instructions and data of P, and information required for its relocation and linking. The linker processes the object module of P to produce a ready-to-execute program form called a *binary program*. If P uses some standard functions, the linker includes their object modules from a library. It stores the generated binary program in a library. The loader loads it into the memory area allocated to P, prepares it for execution in this memory area, and passes control to it for execution.

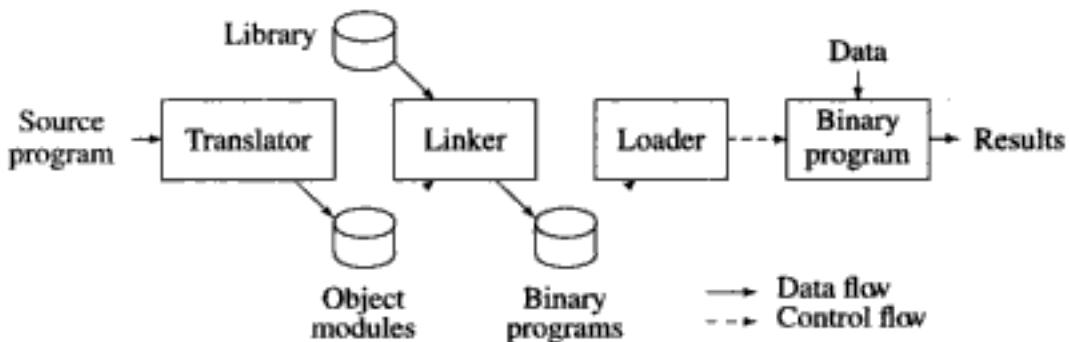


Fig. 5.21 A schematic of program execution

Translated, linked and load time addresses The *origin* of a program is the start address of the memory area in which the program is expected to execute. While compiling a program, a translator is given an origin specification (otherwise, it assumes a default value, typically 0.) This is called the *translated origin* of the program. The translator uses the value of the translated origin to allocate addresses to its instruc-

tions and data. These are called *translation time addresses*. The *execution start address* or simply the *start address* of a program is the address of the instruction with which its execution begins. The start address assigned by the translator is the *translated start address* of the program. Every object module contains the translated origin and translated start address of a program and uses its translation time addresses.

The origin of a program may have to be changed by the linker or the loader. Many object modules may use the same origin and the same translation time addresses. Address conflicts arise when such object modules are to be included in the same binary program. The linker changes the origin specified in some object modules to resolve these address conflicts. The origin of the binary program generated by the linker is called its *linked origin*. The address of the memory area where a binary program is to be loaded for execution is called its *load origin*. The origin of a program has to be changed by the loader if its linked origin differs from its load origin. A change of origin leads to changes in the execution start address and in the addresses assigned to instructions and data.

In this Section we discuss different forms of a program, their properties, their processing by the linker and loader, and the advantages of using them in an operating system. We use programs written in a simple hypothetical assembly language to illustrate the concepts of relocation and linking.

A simple assembly language An assembly language statement has the following format:

[Label] <Opcode> <operand spec>, <operand spec>

The first operand is always a CPU register—AREG, BREG, CREG or DREG. The second operand refers to a memory byte using a symbolic name. Self-explanatory opcodes like ADD and MULT are used to designate arithmetic operations. The MOVER instruction moves a value from the memory operand to the register operand, while the MOVEM instruction does the opposite. All arithmetic is performed in a register and sets a *condition code*. The condition code can be tested by a Branch on Condition (BC) instruction. The assembly statement corresponding to it has the format

BC <condition code spec>, <memory address>

where <condition code spec> is a character string with obvious meaning, e.g., GT, EQ, etc. The BC instruction transfers control to the instruction with the address <memory address> if the current value of condition code matches <condition code spec>. For simplicity, all addresses and constants are in decimal, and all instructions occupy 4 bytes. The opcode, register operand and memory operand of an instruction occupy 2, 1 and 3 digits, respectively. The sign is not a part of an instruction.

5.11.1 Relocation

Figure 5.22 shows program P, an assembly program, and its generated code. The statement START 500 indicates that translated origin of the program should be 500.

The translation time address of LOOP is therefore 504. The address of A is 540. Instructions in bytes with addresses 532 and 500 use these addresses to refer to LOOP and A, respectively. These addresses depend on the origin of the program in an obvious way. Instructions using such addresses are called *address sensitive instructions*. A program containing address sensitive instructions can execute correctly only if it is loaded in the memory area whose start address coincides with the origin of the program. To execute from some other memory area, addresses in address sensitive instructions have to be suitably modified. This process is called *program relocation*.

	<u>Statement</u>	<u>Address</u>	<u>Code</u>
	START 500		
	ENTRY TOTAL		
	EXTERN MAX, ALPHA		
	READ A	500)	+ 09 0 540
LOOP		501)	
	:		
	MOVER AREG, ALPHA	518)	+ 04 1 000
	BC ANY, MAX	519)	+ 06 6 000
	:		
	BC LT, LOOP	538)	+ 06 1 501
	STOP	539)	+ 00 0 000
A	DS 1	540)	
TOTAL	DS 1	541)	
	END		

Fig. 5.22 Assembly program P and its generated code

If linked origin \neq translated origin, relocation must be performed by the linker. If load origin \neq linked origin, relocation must be performed by the loader. In general, a linker always performs relocation, whereas some loaders do not. A loader is an *absolute loader* if it cannot handle the situation load origin \neq linked origin. Relocation requires knowledge of translated and linked origins and information about address sensitive instructions. Example 5.10 illustrates relocation of P.

Example 5.10 The translated origin of program P in Figure 5.22 is 500. The translation time address of symbol A is 540. The instruction corresponding to the statement READ A is an address sensitive instruction. If the linked origin is 900, A would have the link time address 940, so the address in the READ instruction should be corrected to 940. Similarly the instruction in translated memory byte 532 contains 504, the address of LOOP, which should be corrected to 901. (Note that operand addresses in the instructions with addresses 516 and 520 also need to be 'corrected'. This is an instance of linking, which is discussed in Section 5.11.2.)

5.11.2 Linking

The MOVER and BC statements in program P use the operands ALPHA and MAX (see Figure 5.22). These are data and instruction bytes located in some other program or in

a library module. The linker obtains their addresses and puts them in the instructions corresponding to the MOVER and BC statements. *Linking* is the process of binding an external reference to the correct link time address.

A *public definition* is a symbol defined in one translation unit that may be referenced in other translation units. An *external reference* is a reference to a symbol that is not defined in the translation unit. ENTRY and EXTRN statements in an assembly program list its public definitions and external references, respectively. The translator puts information about the ENTRY and EXTRN statements in an object module for use by the linker. Example 5.11 illustrates linking.

Example 5.11 In program P of Figure 5.22 the statement ENTRY TOTAL indicates that a public definition of TOTAL exists in the program. Note that LOOP and A are not public definitions even though they are defined in the program. The statement EXTRN MAX, ALPHA indicates that the program contains external references to MAX and ALPHA. The assembler does not know the address of an external symbol, so it puts zeroes in the address fields of instructions using these symbols. Now consider program Q described below:

<u>Statement</u>		<u>Address</u>	<u>Code</u>
START	200		
ENTRY	ALPHA		
- -			
ALPHA	DS	25	232) + 00 0 025
	END		

Program P contains an external reference to symbol ALPHA that is a public definition in Q with the translation time address 232. Let the link origin of P be 900. Its size is 42 bytes. The link origin of Q is therefore 942, and the link time address of ALPHA is 974. Linking is performed by putting the link time address of ALPHA in the instruction of P that uses ALPHA, i.e., by putting the address 974 in the instruction with the translation time address 516 in P.

Static and dynamic linking The linker shown in Figure 5.21 performs static linking using the schematic illustrated in Example 5.11. It generates a binary program that does not contain any unresolved external references. Dynamic linking is performed when a binary program contains unresolved references to external symbols and one of these references is encountered during execution of a binary program. It resolves the external reference using the schematic illustrated in Example 5.11.

Dynamic linking is implemented using the following arrangement: If a reference to an external symbol is to be resolved dynamically, the static linker links it to a standard module whose sole purpose is to call the dynamic linker to resolve the external reference. The dynamic linker is activated when such an external reference is encountered during execution of the program. It searches object module libraries to locate an object module that contains the required symbol as a public definition. This object module is linked to the binary program using the schematic illustrated in Example 5.11. Since an object module may contain an external reference to a

symbol that is a public definition in the binary program, the dynamic linker maintains information about public definitions all through the execution of a binary program.

Dynamic linking has the advantages mentioned in Section 5.2. However, it leads to one complication—the memory allocation model of Figure 5.4 is not applicable because the code and data components of a dynamically linked module are allocated memory only after execution of a program has started. By that time some program controlled dynamic data (PCD data) would have been allocated in the heap (see Example 5.12). This situation complicates heap management actions, particularly reuse of memory.

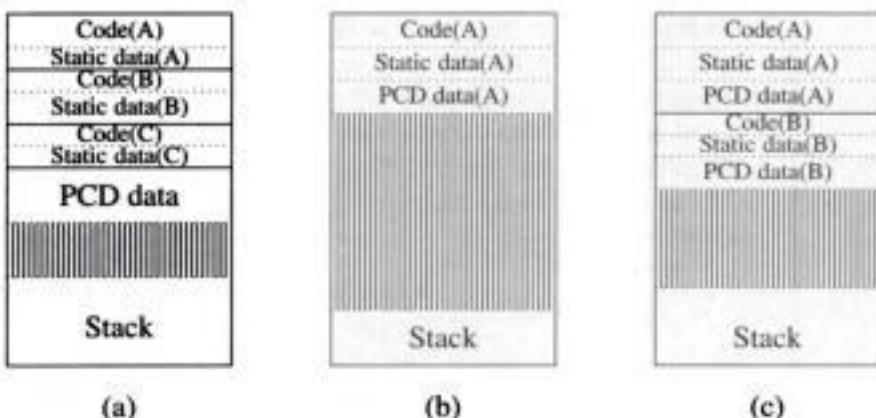


Fig. 5.23 Static and Dynamic linking

Example 5.12 A program consists of three object modules X, Y and Z. Figure 5.23(a) illustrates components of the program in memory at the start of execution if static linking is performed. Figure 5.23(b) illustrates the situation using dynamic linking when some procedure of module X is in execution. Figure 5.23(c) illustrates the situation when a procedure of module X invokes a procedure of module Y. Now the PCD data of X and Y are separated by the code and static data of Y.

5.11.3 Program Forms for Use in Operating Systems

Three features of a program are important for its execution in an OS:

- How much memory does the program need?
- Can the program execute from any area of memory, or does it have to be executed from a specific memory area?
- Can the code of the program be shared by several users concurrently?

The size of a program and its ability to execute from any area of memory is important for its turn-around time because the program can be executed only when a memory area that is large enough to accommodate it becomes available. To execute the program in that memory area, it should either be available as a binary program whose load origin coincides with the start address of the area, or it has to be relocated to satisfy this condition. The former alternative would be desirable. However,

it implies that several copies of the program, each with a different load origin, would have to be prepared and stored on a disk. Thus, it requires large disk areas to be committed to each program. The latter alternative avoids this requirement, but it requires linking and loading to be performed every time the program is to be executed. Shareability of a program is important if the program is to be used by several users at the same time. If a program is not shareable, several copies of the program would have to exist in memory at the same time.

Some of the considerations mentioned above are not important in computer systems that provide large virtual memories. However, it is useful to know different program forms and their features discussed here.

Table 5.4 Program forms employed in operating systems

Program form	Features
Object module	Contains instructions and data of a program and information for its relocation and linking.
Binary program	Ready-to-execute form of a program.
Dynamically linked program	Linking is performed in a lazy manner, i.e., an object module defining a symbol is linked to a program when that symbol is referenced during its execution. Object modules that define symbols that are used in a program but are not referenced during an execution need not be linked to it at all.
Self-relocating program	The program can relocate itself to execute from any area of memory.
Reentrant program	The program can be executed on several sets of data concurrently.

Table 5.4 summarizes features of various program forms. A dynamically linked program potentially reduces the memory requirement of a program. The overlay structure helps to reduce the memory requirement of a program. The self-relocating program form eliminates the need to have several copies of a program on a disk, each with a different load origin. The self-relocating program form is not important when noncontiguous memory allocation is used, and both overlay structured programs and self-relocating programs are unnecessary in a computer system using virtual memory. The reentrant program form avoids the need to have multiple copies of a program in memory.

5.11.3.1 Self-relocating Programs

A self-relocating program performs relocation of its own address sensitive instructions. Consequently, it can execute from any area of memory. The following provisions are necessary in a self-relocating program:

1. It should know its translated origin and its execution start address. It should also know addresses of its address sensitive instructions.

2. It must contain a *relocating logic*, i.e., code to perform its own relocation.

The start address of the relocating logic is specified as the execution start address of the program. Hence the relocating logic gains control when the program is loaded for execution, and uses the information in item 1 to perform its own relocation. Self-relocation is faster than relocation by a linker for obvious reasons.

5.11.3.2 Sharing of Programs

Programs can be shared in both static and dynamic manner. Consider two programs A and B that use a common program C. Static sharing of C is performed using static linking. Hence C is included in both A and B and the identity of C is lost in the binary programs produced by the linker. If programs A and B are initiated simultaneously, two copies of C would exist in the memory (see Figure 5.24(a)). Thus, static sharing of a program is simple to implement, but wastes memory during execution of programs that share it.

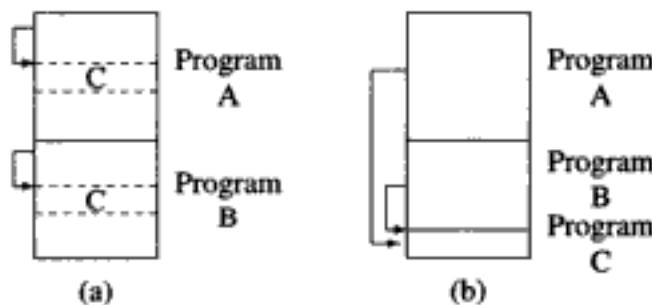


Fig. 5.24 Sharing of a program: (a) static sharing (b) dynamic sharing

Dynamic sharing implies that the code of a program is shared by other programs during their execution. It is implemented using dynamic linking. When program B needs to use program C in a shared mode, the OS links C to it. In Figure 5.24(b) this fact is depicted by drawing an arrow from B to C. The OS also notes the fact that program C is to be used in the shared mode. When program A needs to use program C the kernel checks whether a copy of C already exists in memory. Since it does, the kernel links the existing copy of C to A. Thus only one copy of C exists even when it is used by more than one program. Dynamic sharing saves memory, however its implementation is complex. The kernel has to keep track of shared programs in memory and perform dynamic linking. The shared program has to be coded as a reentrant program to avoid mutual interference by programs that share it.

Reentrant programs When program C is dynamically shared by programs A and B, the data used by C on behalf of A should be protected from interference by its execution on behalf of B. This is achieved by allocating separate copies of the data space of C for use by A and B. The trick is to allocate each data area dynamically, and hold its address in a CPU register. Contents of the CPU registers get saved as

part of the CPU state when a program is preempted, and are put back into the CPU registers when the program is scheduled again. These actions would ensure that different invocations of C will not interfere with each other's data.

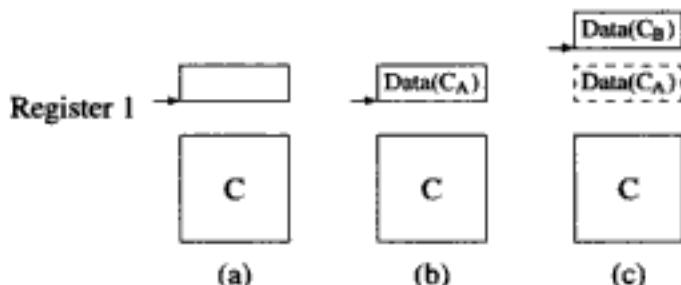


Fig. 5.25 Reentrant program (a) Structure of program C, (b)-(c) Invocations by A and B.

Figure 5.25 illustrates this arrangement. Program C is coded such that it assumes register 1 to point to the start of its data area (see Figure 5.25(a)). Data items in this area are accessed using different offsets from the address contained in register 1. When A calls C, a data area is allocated for this invocation. This is depicted as Data(C_A) (see Figure 5.25(b)). Program A may allocate this area as part of its PCD data and load its address in register 1 while calling C. Alternatively, C may itself allocate this area at the start of its execution on behalf of A. When execution of A is preempted, contents of register 1 would be stored in A's PCB. When C is called by B, a data area Data(C_B) is similarly allocated and register 1 is set to point at the start of this area (see Figure 5.25(c)). Thus executions of programs A and B do not interfere with one another.

EXERCISE 5

1. Explain why program controlled dynamic data (PCD data) cannot be allocated on a stack.
2. A program P contains large arrays whose dimension bounds are specified as constants, e.g., A(100000). Comment on validity of the following statement: "Memory requirements and execution times of P would be identical irrespective of whether P uses static memory allocation or dynamic memory allocation."
3. Two versions of a program P, called P_s and P_d , are developed using static and dynamic memory allocation, respectively. When an effort is made to execute P_s , an OS gives the message "Insufficient memory, cannot run the program." However, the same OS executes program P_d without any difficulties. Explain why this happens.
4. A *worst fit* allocator always splits the largest free memory area while making an allocation. Compare its advantages and drawbacks with the first fit and best fit allocators.
5. Explain why free lists in a buddy system are doubly linked.
6. When a memory block is freed, a memory allocator makes an effort to merge it with one or both of its neighbors. Comment on validity of the following statement in this context: "If sizes of neighboring blocks are known, it is adequate to have a tag at only one boundary of each block; however, if sizes of neighboring blocks are not known, it

is essential to have tags at both boundaries of each block."

7. Discuss memory fragmentation in the buddy system and powers-of-two allocators.
8. Boundary tags are used as tags in a buddy system. Discuss execution efficiency and memory utilization efficiency of this method.
9. The tags in a buddy system are stored in a hash table. For each free block with address a and size s , a pair (a, s) is stored in the hash table. Evaluate the effectiveness of this method.
10. A buddy system allocator is allocated an area of 64K bytes. Blocks of size 2K, 11K, 120 bytes and 20K are allocated in that order.
 - (a) Show the allocation status and free lists of the allocator. How many splits were performed?
 - (b) Show the allocation status and free lists of the allocator after the block of 120 bytes is freed. How many coalesce operations were performed?
11. (a) Justify the statement : "When a block is released in a buddy system the number of free blocks in the system may increase by 1, may remain unchanged, or may decrease by a number between 1 and n , both inclusive, where n depends on the memory available with the system."
(b) Determine the value of n if the minimum block size in the buddy system of Problem 10 is 16 bytes.
12. A Fibonacci buddy system uses blocks whose sizes are multiples of the terms of the Fibonacci series, for example 16, 32, 48, 80, 128, ... Hence the size of a block is the sum of sizes of two immediately smaller blocks.
Compare the execution efficiency and memory efficiency of the Fibonacci buddy system with the binary buddy system.
13. A memory allocator works as follows: Small memory areas are allocated using a buddy system. Large memory areas are allocated using a free list and a first fit allocator. Comment on the efficiency and memory utilization achieved by this allocator.
14. Does the fifty percent rule apply to the following allocators?
 - (a) Buddy system
 - (b) Powers-of-two allocator
 - (c) Slab allocator
15. The kernel of an OS receives requests for memory allocation at a high rate. It is found that a large fraction of the requests are for memory areas of size 100, 300 and 400 bytes (we call these as 'standard' sizes). Other requests are for areas of various other sizes. Design a memory allocation scheme in which no fragmentation arises while allocating areas of standard sizes and no internal fragmentation arises while allocating areas of other sizes.
16. Compute the slack for each class of buffers if a lazy buddy allocator were to be used instead of the buddy allocator in Problem 10.
17. A powers-of-two allocator uses a minimum block size of 16 bytes and a maximum block size of 32 K bytes. It starts its operation with at least one free block of each size. Compare its operation with the buddy system if it processes the same requests as in Problem 10.
18. Comment on the following statements:

- (a) Self-relocating programs are less efficient than relocatable programs.
 - (b) There would be no need for linkers if all programs are coded as self relocating programs.
19. A self-relocating program needs to know its own load address for use in the relocating logic. A program can obtain its own load address by making a novel use of the "branch to subroutine" (BSUB) instruction: The BSUB instruction transfers control to a subroutine and loads the return address in a CPU register. The program can use the return address to calculate its own load address. Write a self-relocating program that uses this technique to determine its own load address.

BIBLIOGRAPHY

Linkers and Loaders are described in Dhamdhere (1999).

Knuth (1973) is the classical starting point for a study of contiguous memory management. Hoare and McKeag (1971) survey various memory management techniques. Randell (1969) is an early paper on the motivation for virtual memory systems. Denning (1970) describes the fundamentals of virtual memory systems.

Vahalia (1996) describes the various kernel memory allocators used in Unix systems. McKusick and Karels (1988) describe the McKusick-Karels memory allocator. Lee and Barkley (1989) describe the lazy buddy allocator. Both these allocators are used in Unix. Bonwick (1994) and Bonwick and Adams (2001) describe the slab allocator. Mauro and McDougall (2001) describe use of the slab allocator in Solaris, while Beck *et al* (2002), and Bovet and Cesati (2003) describe its implementation in Linux. The Windows kernel uses several memory allocation policies for its own memory requirements. It implements buddy-system like allocation for medium sized blocks and heap based allocation for small block sizes. Russinovich and Solomon (2005) describe kernel memory allocation in Windows.

1. Beck, M., H. Bohme, M. Dziadzka, U. Kunitz, R. Magnus, C. Schroter, and D. Verworner (2002): *Linux Kernel Programming, Third edition*, Pearson Education.
2. Bonwick, J. (1994): "The slab allocator: An object-caching kernel memory allocator," *Proceedings of the Summer 1994 Usenix Technical Conference*, 87–98.
3. Bonwick, J. and J. Adams (2001): "Extending the slab allocator to many CPUs and arbitrary resources," *Proceedings of the 2001 Usenix Annual Technical Conference*, 15–34.
4. Bovet, D. P., and M. Cesati (2003): *Understanding the Linux Kernel*, O'Reilly, Sebastopol.
5. Denning, P. J. (1970): "Virtual Memory," *Computing Surveys*, 2 (3), 153–189.
6. Dhamdhere, D. M. (1999): *Systems Programming and Operating Systems* (2nd revised edition), Tata McGraw Hill, New Delhi.
7. Hoare, C. A. R., and R. M. McKeag (1971): "A survey of store management techniques," in *Operating Systems Techniques*, by C.A.R. Hoare and R.H. Perrott (Eds.) Academic Press, London.
8. Knuth, D. E. (1973): *The Art of Computer Programming* (2nd edition), Vol. I : Fundamental Algorithms, Addison-Wesley, Reading.
9. Kuck, D. J., and D. H. Lowrie (1970): "The use and performance of memory hierarchies," in *Software Engineering*, 1, J.T. Tou (Ed.), Academic Press, New York.

10. Lee, T. P., and R. E. Barkley (1989): "A watermark-based lazy buddy system for kernel memory allocation," *Proceedings of the Summer 1989 USENIX Technical Conference*, 1–13.
11. Mauro, J., and R. McDougall (2001): *Solaris Internals—Core Kernel Architecture*, Prentice-Hall.
12. McKusick, M. K., and M. J. Karels (1988): "Design of a general-purpose memory allocator for the 4.3 BSD Unix kernel," *Proceedings of the Summer 1988 USENIX Technical Conference*, 295–303.
13. Peterson, J. L. and T. A. Norman (1977): "Buddy systems," *Communications of the ACM*, **20** (6), 421–431.
14. Randell, B. (1969): "A note on storage fragmentation and program segmentation," *Communications of the ACM*, **12** (7), 365–369.
15. Russinovich, M. E., and D. A. Solomon (2005): *Microsoft Windows Internals, Fourth edition*, Microsoft Press, Redmond.
16. Vahalia, U. (1996): *Unix Internals—The New Frontiers*, Prentice-Hall, Englewood Cliffs.

chapter 6

Virtual Memory

Virtual memory is an illusion that a computer system possesses more memory than it actually does. This illusion makes a process independent of the size of real memory. It also permits a large number of processes to share a computer system without constraining each other. Virtual memory is implemented through the part of the memory hierarchy consisting of memory and a disk—the code and data of a process are stored on a disk and parts of it are loaded into memory when needed during execution of the process. It makes use of the model of noncontiguous memory allocation and contains both hardware and software components.

Performance of virtual memory depends on the rate at which parts of a process have to be loaded into the memory from a disk. The principle of locality of reference holds the promise that this rate would be low if an adequate amount of memory is allocated to a process. Practicality of virtual memory now depends on controlling the amount of memory allocated to a process and on deciding which parts of a process to keep in memory. Both these tasks are performed by the software component of the virtual memory.

We start by discussing the principle of locality of reference and the techniques used to decide which parts of a process to keep in memory. The techniques to control the amount of memory allocated to a process are then discussed. Hardware features to speed up operation of processes in virtual memory and to help the virtual memory software are also discussed.

6.1 VIRTUAL MEMORY BASICS

Users always want more from a computer system—more resources and more services. The need for more resources is satisfied either by obtaining more efficient use of existing resources, or by creating an illusion that more resources exist in the system. A virtual memory is what its name indicates—it is an illusion of a memory that is larger than the real memory, i.e., RAM, existing in a computer system. As we

discussed in Section 1.1, this illusion is a part of a user's abstract view of memory. Thus, a user or his process sees only the virtual memory. The kernel implements the illusion using a combination of hardware and software means. We refer to the software component of virtual memory as the *virtual memory handler*.

The basis of virtual memory is the model of noncontiguous memory allocation (see Section 5.6). Each process is assumed to consist of parts called *process components*. The parts can be loaded into non-adjacent areas of memory for execution. The memory management unit (MMU) translates every operand or instruction address used by a process into the address of the memory byte where the operand or instruction actually resides. Use of the noncontiguous memory allocation model reduces the problem of memory fragmentation since a free area of memory can be reused even if it is not large enough to hold an entire process. More user processes can be accommodated in memory this way, which benefits both users and the OS.

The illusion of a large memory is created by enabling execution of a process whose size exceeds the size of memory. This is achieved by keeping a process on a disk and loading only its required portions in memory at any time. The kernel uses this idea to reduce memory allocation to processes in general—that is, even processes that can fit in the memory are not loaded fully into the memory. This strategy further increases the number of processes that can be accommodated in memory.

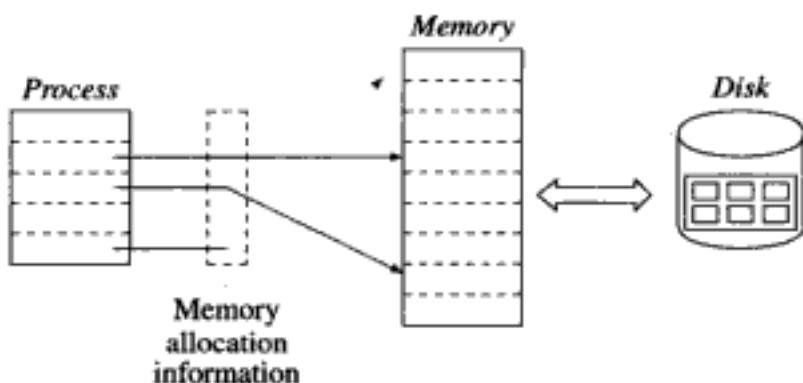


Fig. 6.1 Overview of virtual memory

Figure 6.1 shows a schematic of virtual memory. A process contains five components. Three of these components are presently in memory. Information about the memory areas where these components exist is maintained in a data structure of the virtual memory handler. This information is used by the MMU during address translation. When an instruction in the process refers to a process component that is not in memory, it is loaded from the disk. Occasionally the virtual memory handler removes some process components from memory to make place for loading other components. We are now ready to define virtual memory.

Definition 6.1 (Virtual memory) Virtual memory is a memory hierarchy consisting

of a computer system's memory and a disk, that enables a process to operate with only some portions of its address space in memory.

Two fundamental approaches used to implement virtual memory are:

- Paging
- Segmentation

These approaches differ in the manner in which the boundaries and sizes of process components are determined. In paging, each process component is called a *page*. All pages are identical in size. Page size is determined by the architecture of the computer system and demarcation of pages in a process is performed implicitly by the architecture. In segmentation, each process component is a *segment*. A programmer declares some significant logical entities in a process as segments for the purpose of virtual memory implementation. Thus identification of process components is performed by the programmer, and segments can have different sizes. As we shall see in later Sections, paging and segmentation have different implications for memory utilization and sharing of programs and data.

The virtual memory systems using paging and segmentation are called *paged virtual memory systems* and *segmented virtual memory systems*, respectively. Some systems use a combined segmentation-and-paging approach to obtain advantages of both the approaches. We begin this Chapter with an overview of operation of a virtual memory system.

6.1.1 Demand Loading of Process Components

Virtual memory uses demand loading of process components to give the illusion of a large memory. When the execution of a process is to be started, the virtual memory handler loads only one of its components in memory. This component contains its first instruction. Other components are loaded only when needed. This way the memory allocated to a process can be smaller than its size. To keep the memory commitment low, the virtual memory handler removes a process component from memory if that component might not be needed for some time during execution of a process.

Performance of a process in a virtual memory system depends on the rate at which components of a process have to be loaded into the memory. Good performance is obtained if this rate is low. The virtual memory handler uses special techniques to ensure this. These techniques exploit the principle of locality of reference, which we shall discuss in Section 6.2.2.1. A programmer can also contribute to good performance by coding a process in such a manner that it does not require too many of its components at any time during execution (see Problem 7 in Exercise 6).

6.1.2 An Example of Virtual Memory Operation

Figure 6.2 shows execution of process A in a virtual memory. The process consists of four components 1–4. For convenience, we call them as A-1, A-2, A-3 and

A-4. Logical start address, size and start address of memory area allocated to each of these components are as follows:

Process component	A-1	A-2	A-3	A-4
Logical start address	0	3000	9000	13000
Size	3000	6000	4000	4000
Start address of memory area	47000	15000	38000	-

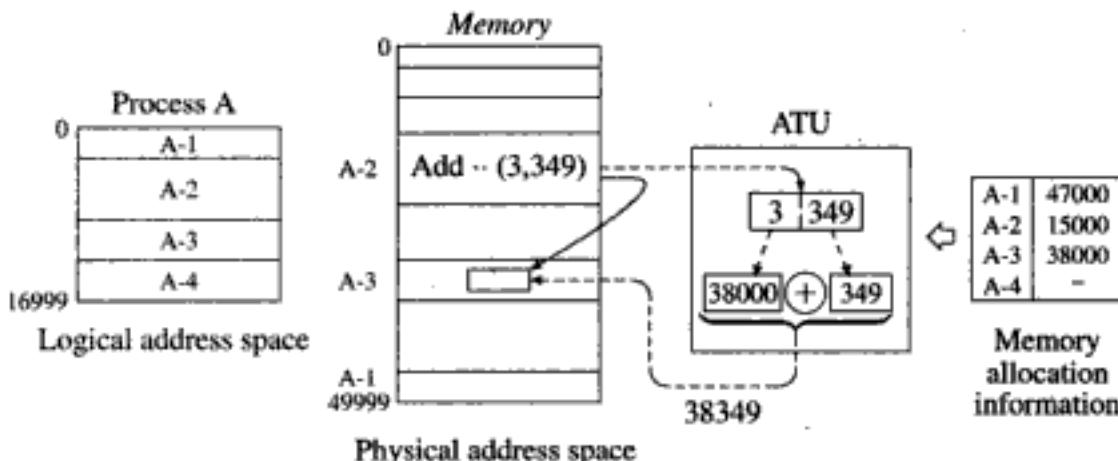


Fig. 6.2 Logical address space, physical address space and address translation

The logical address space of A consists of addresses 0 ... 16999. The computer system has a memory of 50K bytes, hence the physical address space consists of addresses 0 ... 50K - 1. Some part of the physical address space is occupied by components of process A, while the remaining space may be occupied by components of other processes in the system. Note that only components A-1, A-2 and A-3 exist in memory; component A-4 is not present in memory.

Information about start addresses of memory areas allocated to components of A is given to the MMU when A is dispatched. When the instruction $\text{Add } \dots (3, 348)$ of component A-2 is executed, the MMU obtains the address of component number 3, i.e., 38000. Following Eq. (5.1), it computes $38000 + 348$ to obtain effective memory address of byte 348 in this component. An access is now made to the memory location with address 38348. If some address in component 4, e.g., (4, 228), is referenced, the virtual memory handler would load component A-4 in memory and make its start address known to the MMU.

6.2 DEMAND PAGING

6.2.1 Overview of Paging

A process is viewed to consist of pages, numbered from 0 onwards. Each page is of size s bytes. The memory of the computer system is viewed to consist of *page frames*, where a page frame is a memory area that has the same size as a page. Page frames are numbered from 0 to $nf - 1$ such that $nf \times s$ is the size of the memory. The physical

address space consists of addresses from 0 to $nf \times s - 1$. At any moment, a page frame may be free, or it may contain a page of some process. Each logical address used in a process is considered to be a pair (p_i, b_i) , where p_i is a page number and b_i is an offset in p_i , $0 \leq b_i < s$. The size of a page (and hence that of a page frame) is chosen to be a power of 2. This choice simplifies address translation and also makes it more efficient (see Section 5.7).

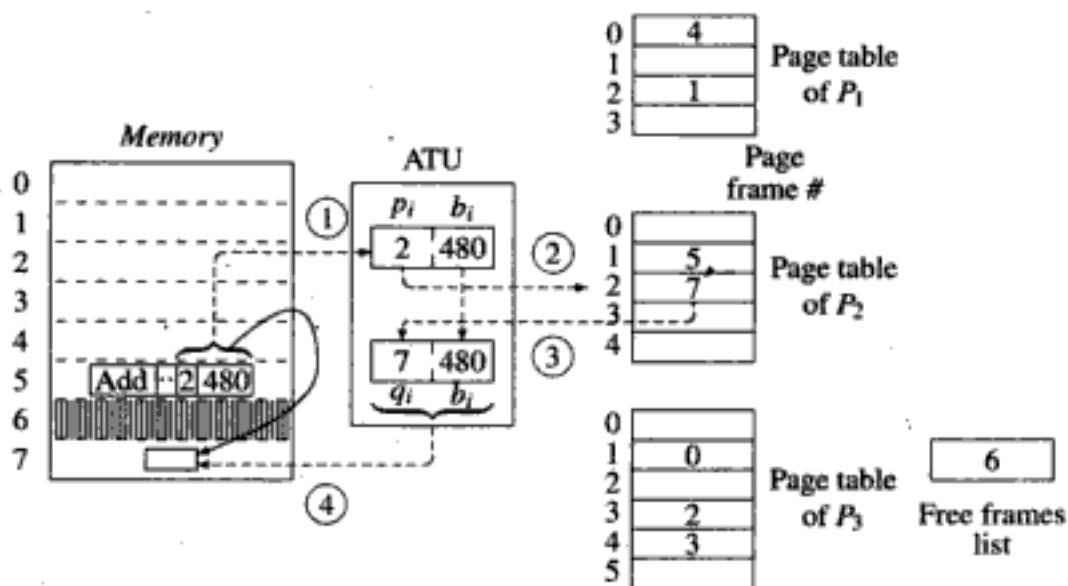


Fig. 6.3 Address translation in a paged virtual memory system

Figure 6.3 shows an overview of a paged virtual memory system in which a page is assumed to contain 1K bytes. Three processes P_1 , P_2 and P_3 , have some of their pages in memory. The memory contains 8 page frames numbered from 0 to 7. Memory allocation information for a process is stored in a *page table*. Each entry in the page table contains memory allocation information for one page of a process. It contains the page frame number where a page resides. Process P_2 has its pages 1 and 2 in memory. They occupy page frames 5 and 7 respectively. Process P_1 has its pages 0 and 2 in page frames 4 and 1, while process P_3 has its pages 1, 3 and 4 in page frames 0, 2 and 3, respectively. The free frames list contains a list of free page frames. Currently only page frame 6 is free.

Process P_2 is currently executing the instruction 'Add .. 2528', so the MMU uses P_2 's page table for address translation. The MMU views the operand address 2528 as the pair (2, 480). It now accesses the entry for page 2 in P_2 's page table. This entry contains frame number 7, so the MMU forms the address $7 \times 1024 + 480$ and uses it to make a memory access. In effect, byte with the offset 480 in page frame 7 is accessed.

6.2.2 Demand Paging Preliminaries

In demand paging, a page is loaded in memory when needed, i.e., when a logical address generated by a process points to a page that is not present in memory. To facilitate demand paging, the entire logical address space of a process is maintained on a secondary storage device like a fast disk. We call this disk the *paging device*. While initiating execution of a process, an area is allocated on the paging device for its logical address space and its code and data are copied into this space. This space is called the *swap space* of the process. A page is loaded from this area into the memory when needed. When the virtual memory handler decides to remove a page from memory, the page is copied back into the swap space if it was modified since the last time it was loaded in memory. This way the swap space contains an up-to-date copy of every page that is not present in memory.

Demand paging involves interaction between hardware and software components of the virtual memory, i.e., between the MMU and the virtual memory handler. Three concepts are important in understanding operation of demand paging. These are

- Page faults (also called missing page interrupts)
- Page-in and page-out operations
- Page replacement.

New fields are added to entries in the page table to facilitate implementation of these concepts. The format of a page table entry is as shown in Figure 6.4. The *valid bit* field contains a boolean value to indicate whether the page exists in memory, e.g., 1 to indicate 'resident in memory' and 0 to indicate 'not resident in memory'. The *misc info* field contains useful information for page-in/page-out and page replacement operations. It is divided into four subfields. Information in the *prot info* field is used to protect the page. *ref info* contains information concerning references made to the page while it is in memory. As discussed later, this information is used for page replacement decisions. The *modified* field indicates whether the page has been modified, i.e., whether it is *dirty*. The *other info* field contains other information concerning a page, e.g., address of the disk block in swap space where a copy of the page is maintained.

The address translation procedure is described in Table 6.1. It raises a page fault if the page is not present in memory. The page fault leads to page in, page out and page replacement operations to load the page in memory. Details of these operations are described in the following.

Page faults While performing address translation for a logical address (p_i, b_i) , the MMU checks the valid bit of the page table entry of p_i (see Step 2 in Table 6.1). If it indicates that p_i is not present in memory, the virtual memory handler raises an interrupt called a *page fault* or a *missing page interrupt*. The interrupt action (see Section 2.1.1) transfers control to the interrupt handler, which invokes the virtual memory handler when it finds that the interrupt is a page fault. The interrupt handler

Field	Misc info					
	Valid bit	Page frame #	Prot info	Ref info	Modified	Other info

Field	Description
Valid bit	Indicates whether the page described by the entry currently exists in memory. This bit is also called the <i>presence</i> bit.
Page frame #	Contains number of the page frame occupied by the page.
Prot info	Protection information for the page (whether the page can be read from or written into by processes)
Ref info	Information concerning references made to the page while it is in memory.
Modified	Indicates whether the page has been modified while in memory, i.e., whether it is dirty. This bit is also called the <i>dirty</i> bit.
Other info	Other useful information concerning the page, e.g., its position in the swap space.

Fig. 6.4 Fields in a page table entry

Table 6.1 Steps in address translation

1. A logical address is viewed as a pair (p_i, w_i) , where w_i consists of the lower order s_w bits of the address, and p_i consists of the higher order s_p bits.
2. p_i is used to index the page table. A page fault is raised if the field *valid bit* of the page table entry contains a 0.
3. The *page frame #* field of the PT entry contains a frame number represented as an s_b bit number. It is concatenated with w_i to obtain the effective memory address. This is the physical address of the word.
4. A memory access is made using the effective memory address.

passes the page number that caused the page fault, i.e., p_i , to the virtual memory handler. The virtual memory handler loads page p_i in memory.

Figure 6.5 contains an overview of the virtual memory handler's actions when process P_2 is in execution. While translating the logical address (3,682), the MMU raises a page fault because the valid bit of page 3's entry is 0. When the virtual memory handler gains control, it knows that a reference to page 3 caused the page fault. The *Misc info* field of the page table entry of page 3 contains address of the

disk block that contains page 3. The virtual memory handler obtains this address. It now consults the free frames list and finds that page frame 6 is currently free. It allocates this page frame to page 3 and starts an I/O operation to load page 3 in page frame 6. When the I/O operation completes, it updates page 3's entry in the page table by setting the valid bit to 1 and putting 6 in the page frame # field. Execution of the instruction 'Sub .. (3,682)', which caused the page fault, is now resumed. Logical address (3,682) would be translated to physical address $6 \times s + 682$, which is the address of the byte with offset 682 in page frame 6.

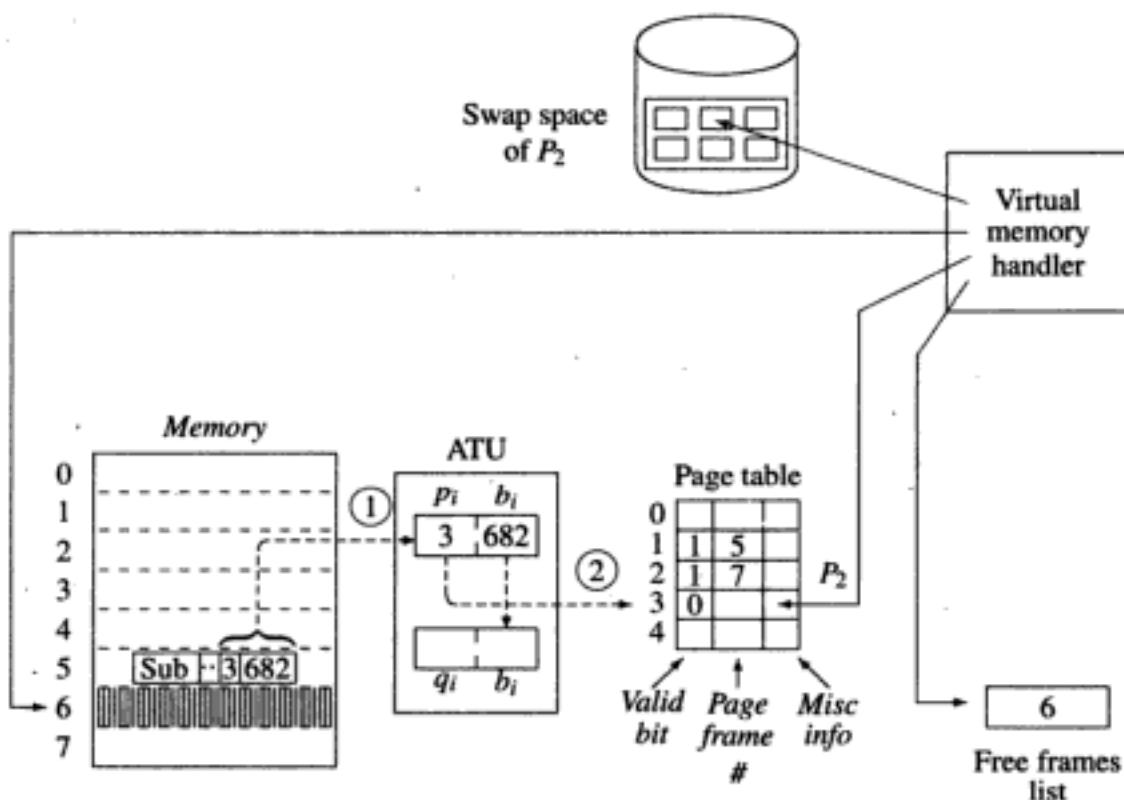


Fig. 6.5 Demand loading of pages

Page-in, page-out and page replacement operations A page is loaded in memory when a page fault occurs for it. This is called a *page-in* operation. In Figure 6.5, a free page frame existed when a page fault occurred. This may not be the case at every page fault. Hence the virtual memory handler may have to remove a page from memory to create free space for loading a required page. If the page being removed had been modified after it was last loaded in memory, the swap space does not contain its up-to-date copy. Such a page needs to be copied from the memory to the disk block allocated to it in the swap-space of the process. This operation is called a *page-out* operation. Loading of a new page into a page frame that previously contained another page is called a *page replacement* operation. It may involve a page-

out operation if the previous page occupying the page frame was modified while in memory, and involves a page-in operation to load the new page.

The page-in and page-out operations required to implement demand paging constitute *page I/O*. The term *page traffic* is used to describe movement of pages in and out of memory. Note that page I/O is distinct from I/O operations performed by processes, which we will call *program I/O*. A process that encounters a page fault becomes *blocked* until the required page is loaded into memory, so execution performance of the process suffers. The CPU can be switched to execution of another process; however, system efficiency would suffer if all processes in the system are blocked due to page faults.

Effective memory access time We use the following notation to compute the effective memory access time in demand paging:

- pr_1 probability that a page exists in memory
- t_a memory access time
- t_{pfh} time overhead of page fault handling.

where pr_1 is called the *hit ratio* in memory. t_{pfh} includes the time consumed by the page-in, page-out and page replacement operations described earlier. It is about two orders of magnitude larger than t_a .

A process's page table exists in memory when the process is in operation. Hence, if a page being referenced also exists in memory, accessing an operand with the logical address (p_i, b_i) consumes two memory cycles—one to access the entry of p_i in the process's page table for address translation, and the other to access the operand from memory using the effective memory address of (p_i, b_i) . If the page is not present in memory, a page fault is raised after referencing the page table, i.e., after one memory cycle. Now the required page is loaded in memory and its page table entry is updated to record the frame number of the page frame where it is loaded. t_{pfh} includes the time consumed by the initial access to the page table that raised the page fault, and the time consumed by the page-in or page replacement operation. We ignore the process switching overhead and assume that the process is scheduled immediately after a page-in or page-replacement operation completes. The effective memory access time is calculated as follows:

$$\text{Effective memory access time} = pr_1 \times 2 \times t_a + (1 - pr_1) \times (t_a + t_{pfh} + 2 \times t_a) \quad (6.1)$$

The effective memory access time can be improved by reducing the number of page faults. Loading pages before they are needed by a process may reduce the number of page faults. The Windows NT operating system performs preloading speculatively—when a page fault occurs, it loads the required page and also a few adjoining pages of the process. This action does not achieve an improvement in the memory access time if the preloaded page is not referenced by the process.

The Linux operating system permits a process to specify which pages should be preloaded. A programmer may use this facility to improve the effective memory access time.

6.2.2.1 Page Replacement

Page replacement becomes necessary when a page fault occurs and no free page frames exist in memory. In principle, the virtual memory handler could select a page frame at random and replace the page contained in it. However, another page fault would result when the replaced page is referenced again, so it is important to replace a page that is not likely to be referenced in immediate future. The principle of locality of reference provides valuable clues for identifying such pages.

Locality of reference The principle of locality of reference can be stated as follows:

A logical address generated while executing an instruction in a process is likely to be in proximity of logical addresses generated during the previous few executed instructions of the process.

Processes exhibit good locality for two reasons. Execution of a program is mostly sequential in nature since only 10–20 percent instructions are branch instructions. Thus, the next instruction to be executed typically follows the previously executed instruction in the logical address space. References to non-scalar data, e.g., arrays, also tend to be in close proximity of previous references because processes tend to perform similar operations on several elements of an array.

Let *proximity region* of a logical address a_i contain all logical addresses that are in close proximity of a_i . For simplicity, we assume that the proximity region of a_i lies entirely within the page that contains a_i , say page p_i , that is, a logical address in the proximity region of a_i is located within page p_i itself.

We define the *current locality* of a process to be the set of pages referenced in its previous few instructions. Principle of locality indicates that the logical address used in an instruction is likely to refer to a page that is in its current locality. Given this likelihood we can expect that too many page faults may not occur during operation of a process if pages in its current locality are present in memory. Thus, one can expect demand paged virtual memory systems to operate with reasonable levels of efficiency.

Note that locality of reference does not imply an absence of page faults. Page faults can occur due to two reasons: First, the proximity region of a logical address may not fit into a page, hence an access in its proximity may refer to a page that is not included in the current locality of the process. Second, an instruction or data referenced by a process may not exist in its current locality. We call this situation a shift in the locality of a process. It typically occurs when a process makes a transition from one action in its logic to another.

Example 6.1 In Figure 6.6, bullets indicate the last few logical addresses used during execution of a process P. Dashed boxes show proximity regions of these logical addresses. Note that proximity regions of logical addresses referenced in page 4 overlap with one another and extend beyond boundaries of the page, and similarly for logical addresses referenced in page 6. Thus, proximity regions are located in pages 0, 1, 3, 4, 5, 6 and 7, however the current locality of P is the set of pages {0, 1, 4, 6}.

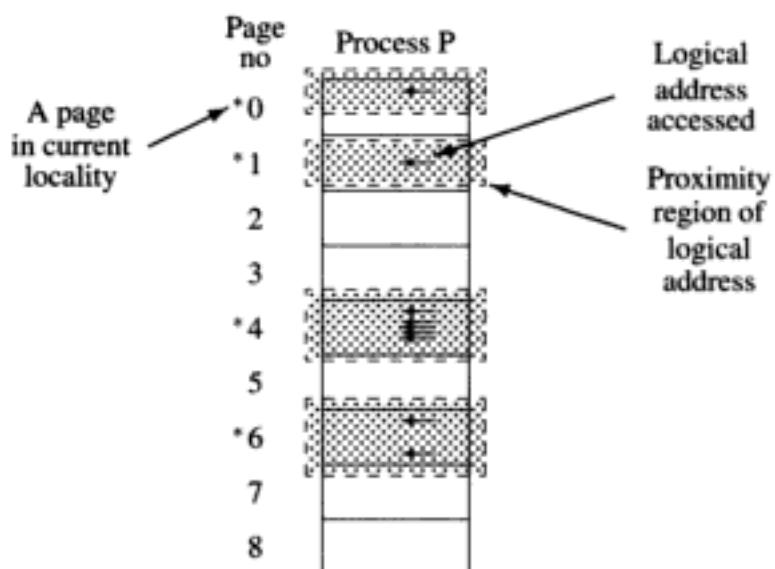


Fig. 6.6 Proximity regions of previous references and current locality of a process

The locality principle helps to decide which page should be replaced when a page fault occurs. Let us assume that the number of page frames allocated to a process P is a constant, hence one of P's pages existing in memory must be replaced. Let t_1 and t_2 be the periods of time for which pages p_1 and p_2 have not been referenced during the execution of P. Let $t_1 > t_2$, implying that some byte of page p_2 has been referenced or executed (as an instruction) more recently than any byte of page p_1 . Hence page p_2 is more likely to be a part of the current locality of the process than page p_1 , that is, a byte of page p_2 is more likely to be referenced or executed than a byte of page p_1 . We use this argument to choose page p_1 for replacement when a page fault occurs. If many pages of P exist in memory, we can rank them according to the last references made to them and replace the page that has been least recently referenced.

Figure 6.7 shows how the page fault rate of a process should vary with the amount of memory allocated to it. We call it the *desirable page fault characteristic*. It is a graph of the page fault rate of a process, i.e., the number of page faults per unit execution time, against the number of page frames allocated to the process. The page fault rate decreases monotonically with the number of page frames, that is, the page fault rate is smaller if a larger amount of memory is allocated to a process. As dis-

cussed later, this property is useful for designing the virtual memory handler actions to counter poor system performance. Note that the slope of the curve is larger when memory allocation is modest. This implies that the page fault rate increases sharply if we reduce the memory allocation too far, or drops rapidly if the allocation is small and we increase it. The curve is flatter in the region of high memory allocation. In this region, a change in memory allocation has a relatively smaller impact on the page fault rate.

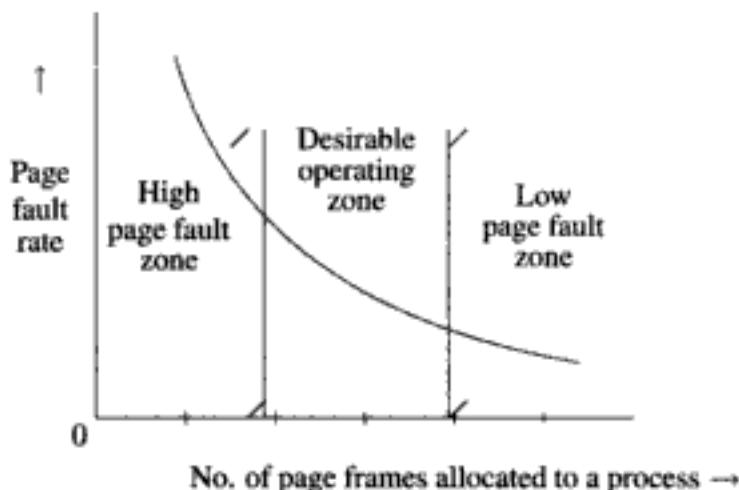


Fig. 6.7 Desirable variation of page fault rate with memory allocation

How much memory should the OS allocate to a process? Two opposite factors influence this decision. From Figure 6.7, we see that an over-commitment of memory would imply a low page fault rate, hence good process performance. However, an over-commitment of memory would mean that a smaller number of processes would fit in memory. This situation could cause CPU idling and poor system performance. An under-commitment of memory would cause a high page fault rate, which would lead to poor process performance. The desirable operating zone marked in Figure 6.7 avoids the regions of low and high memory allocation.

Thrashing Consider a process that is operating to the left of the desirable operating zone in Figure 6.7. This process operates in the region of high page fault rates. If all processes in the system operate in high page fault rate zones, the CPU would be mostly busy performing page traffic and process switching. CPU efficiency would be low and system performance, measured either in terms of average response time or throughput, would be poor. This situation is called *thrashing*.

Definition 6.2 (Thrashing) Thrashing is the coincidence of high page traffic and low CPU efficiency.

Note that low CPU efficiency may result from other causes like a low degree of multiprogramming; however, in these situations high page traffic would not exist.

From Figure 6.7, it is seen that the cause of thrashing is an under-commitment of memory to processes. The cure is to increase the memory allocation for each process. This may have to be achieved by removing some processes from the memory—that is, by reducing the degree of multiprogramming.

Note that this reasoning is qualitative in nature. It does not help us in deciding how much memory, i.e., how many page frames, we should allocate to a process. The main problem in deciding the memory allocation for a process is that the page fault characteristic, i.e., the slope of the curve and the page fault rate in Figure 6.7, varies with processes. In fact, even for the same process the characteristic is likely to be different for different data. Thus it is reasonable to assume that the optimum amount of memory to be allocated to a process would have to be determined dynamically by considering the execution behavior of a process. This issue is discussed in Section 6.4.

6.2.2.2 Optimal Page Size

The size of a page is defined in the architecture of a computer system. It determines the number of bits required to represent the byte offset in a page. The paging hardware implicitly uses this information while decomposing a logical address into a pair (p_i, b_i) . Page size also determines

1. Memory wastage due to internal fragmentation
2. Size of the page table for a process
3. Page fault rates when a fixed amount of memory is allocated to a process.

Consider a process P of size z bytes. A page size of s bytes implies that the process has pages $0 \dots n - 1$, such that $n = \lceil z/s \rceil$, where $\lceil z/s \rceil$ indicates that the value of z/s is rounded upwards. Average internal fragmentation is $s/2$ bytes because the last page would be half empty on the average. The number of entries in the page table is n . Thus internal fragmentation varies directly with the page size, while page table size varies inversely with it.

Interestingly, page fault rate also varies with page size if a fixed amount of memory is allocated to P. This can be explained as follows: The number of pages of P in memory varies inversely with the page size. Twice as many pages of P would exist in memory if the page size is made $s/2$. Now assume the proximity region of an instruction as defined in Section 6.2 to be small compared to $s/2$, so proximity regions of the last few logical addresses generated by a process would fit completely within the pages containing them. Hence when the page size is $s/2$, the memory contains twice as many proximity regions of recent logical addresses as when the page size is s bytes. From the page fault characteristic of Figure 6.7, page fault rates would be smaller for smaller page sizes.

We can compute the page size that minimizes the memory penalty due to the first two factors. If $s \ll z$, and each page table entry occupies one byte of memory, the optimal value of s is $\sqrt{2z}$. Thus, the optimal page size is 400 bytes for a process

size of 80 K bytes, and it is only 800 bytes for a process of 320 K bytes. However, computer systems tend to use larger page sizes (e.g., Pentium and MIPS use page sizes of 4K bytes or more, Sun Ultrasparc uses page sizes of 8K bytes or more and the Powerpc uses a page size of 4K bytes) due to the following reasons:

1. Page table entries tend to occupy more than one byte.
2. Hardware costs are high for smaller page sizes. For example, the cost of address translation increases if a larger number of bits is used to represent a page number.
3. Fast disks that are used as paging devices tend to operate less efficiently for smaller disk block sizes.

The decision to use larger page sizes than the optimal value implies somewhat larger page fault rates for a process. This feature represents a tradeoff between the hardware cost and process efficiency.

6.2.3 Paging Hardware

We discussed the fundamental provision for address translation in Section 6.2.1 (see Figure 6.3). The paging hardware performs the additional functions listed in Table 6.2 to speed up address translation and to support the virtual memory handler functions. We describe some techniques used in implementing these functions. Case studies of paging hardware in a few systems are included at the end of the Section.

Table 6.2 Functions of the paging hardware

- *Address translation and generation of page faults:* Paging hardware contains features to speed up address translation. It also supports paging in a multiprogrammed system by maintaining page tables for several programs and using the correct one for address translation.
- *Memory protection:* This function avoids mutual interference between user processes. Special techniques have to be used due to noncontiguous nature of memory allocation.
- *Providing page replacement support:* The hardware collects information concerning references made to a program's pages. The virtual memory handler uses this information to decide which page to replace when a page fault occurs.

6.2.3.1 Address Translation and Page Fault Generation

Address translation buffers For a logical address (p_i, b_i) , the MMU adds $p_i \times l_{PT_entry}$, where l_{PT_entry} is the length of a page table entry, to the start address of the page table. This provides the address of the page table entry of page p_i . It is now accessed to perform address translation. The page table format is as shown in Figure 6.4. If l_{PT_entry} is a power of 2, $p_i \times l_{PT_entry}$ can be computed efficiently by shifting the value of p_i by a few bits.

If page tables are located in the memory, each access to memory would consume two memory cycles—one for accessing the page table and one for the actual memory access. An *address translation buffer* is used to speed up address translation (see Figure 6.8). The buffer is a small and fully associative memory that contains information concerning some pages of the currently executing process. Each entry in the buffer is a pair of the form (page #, page frame #) for some page that exists in the memory. The buffer is looked up in a content-based manner using a page number as the key—that is, the page number is compared simultaneously with the page # field of all entries in the buffer. The lookup either returns the entry of the page or ends in a failure. For a logical address (p_i, b_i) , the MMU first accesses the buffer (see arrows marked 2' and 3' in Figure 6.8). If entry of the page is not found in the buffer, address translation is performed using the page table. This makes address translation fast if information about the required page is contained in the buffer. The address translation buffer is called a *translation look-aside buffer* (TLB) because of the manner in which it is used.

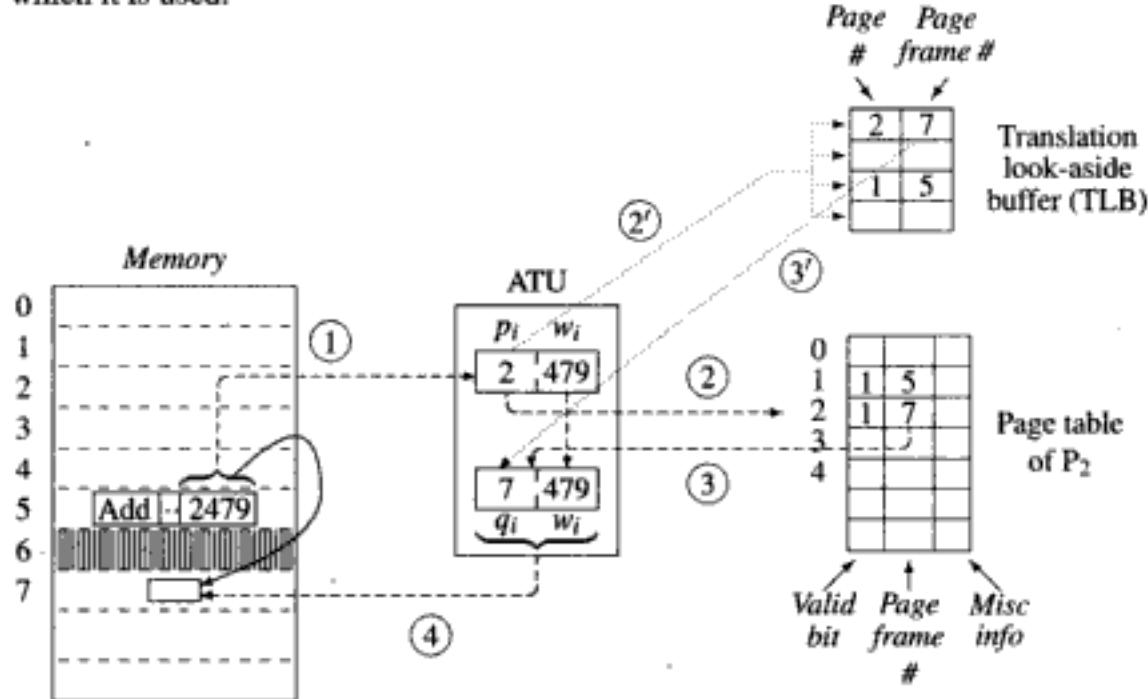


Fig. 6.8 Translation look-aside buffer

Associative memories are expensive, hence the TLB contains only a few entries. The paging hardware keeps information about the last few pages referenced by a process in the TLB. Whenever the TLB lookup fails for a page, the hardware forms a pair (page #, page frame #) and puts it in the TLB. This pair displaces some existing pair from the TLB.

To see how TLB speeds up address translation, we compute the effective memory access time using the following notation:

pr_1 probability that a page exists in memory
 pr_2 probability that a page entry exists in TLB
 t_a memory access time
 t_f access time of TLB
 t_{pfh} time overhead of page fault handling.

Typically t_t is at least an order of magnitude smaller than t_a , which is about two orders of magnitude smaller than t_{ref} . pr_2 is called the *hit ratio in TLB*.

When TLB is not used, the effective memory access time is given by Eq. (6.1). When TLB is used, pr_2 is the probability that an entry for the required page exists in TLB. The probability that a page table reference is necessary and sufficient for address translation is $(pr_1 - pr_2)$. The time consumed by each such reference is $(t_f + 2 \times t_a)$ since an unsuccessful TLB search would precede the page table lookup. A page fault occurs at all other times. Hence the effective memory access time is

$$\text{Effective memory access time} = pr_2 \times (t_t + t_a) + (pr_1 - pr_2) \times (t_t + 2 \times t_a) + (1 - pr_1) \times (t_t + t_a + t_{pfh} + t_t + 2 \times t_a) \quad (6.2)$$

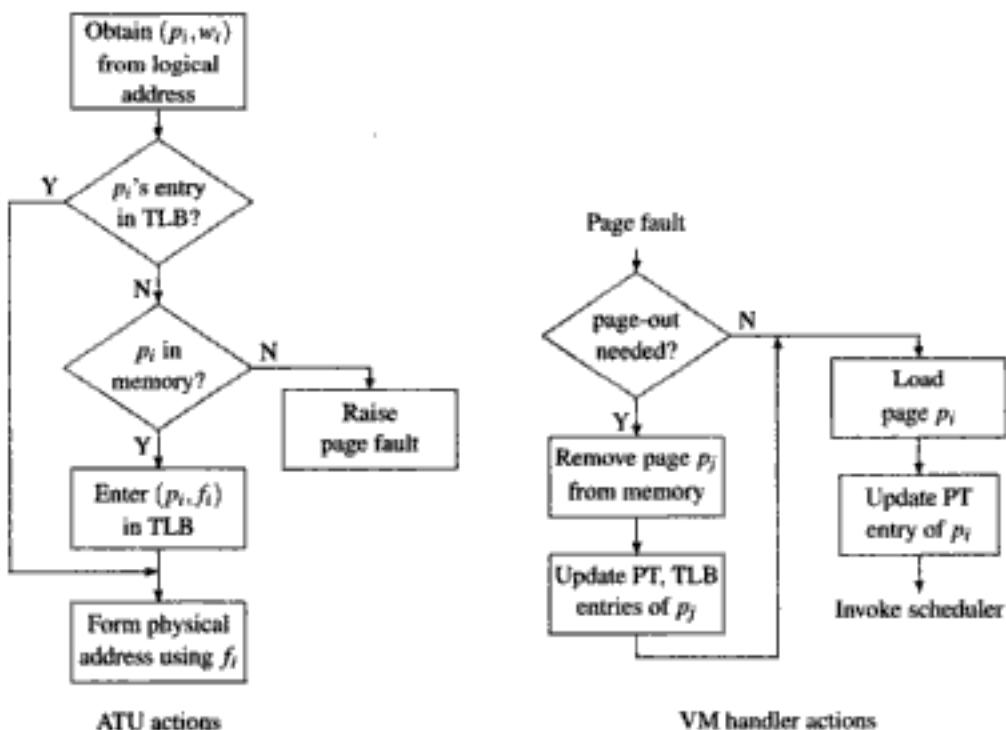


Fig. 6.9 Summary of actions in demand paging

Figure 6.9 summarizes the hardware and software actions involved in address translation and page fault handling for a logical address (p_i, b_i) . The MMU first performs a TLB search for p_i . If it finds a pair (p_i, f_i) , it completes the address

translation using f_i . If TLB search fails, the MMU refers to the page table entry of p_i to check whether p_i exists in memory. If it exists in page f_i , the MMU makes an entry (p_i, f_i) in the TLB and completes the address translation using f_i . If p_i is not present in memory, the MMU generates a page fault. All these actions are performed by the hardware.

The virtual memory handler is activated by a page fault. It checks to see whether an empty page frame is available to load the page; otherwise, it initiates a page out operation for some page p_j to free the page frame f_j occupied by it. p_j 's page table entry is updated to indicate that it is no longer present in memory. If p_j has an entry in TLB, the virtual memory handler erases it by executing an 'erase TLB entry' instruction to prevent possibility of incorrect address translation at p_j 's next reference. A page-in operation is now performed to load p_i in page frame f_j , and p_i 's page table entry is updated when the page-in operation is completed. Execution of the instruction that gave rise to the page fault would be repeated when the process is scheduled again. This time p_i is found to exist in memory, hence the MMU uses information in the page table to complete the address translation. It also makes an entry (p_i, f_j) in the TLB.

Superpages Sizes of computer memories and processes have grown rapidly since the 1990's. However, TLB sizes have not kept pace with this increase because the fully associative nature of TLBs makes them expensive. Consequently, *TLB reach*, which is the product of page size and the number of entries in a TLB, has increased marginally, but its ratio with the memory size has shrunk by a factor of over 1000. Consequently, TLB hit ratios are poor and average memory access times are higher (see Eq. 6.2). Processor caches have also become larger than the TLB reach. This affects performance of the cache because access to instructions or data in a cache may be slowed down due to TLB misses and look-ups through page tables. One way of countering these problems is to use a larger page size, so that the TLB reach becomes larger. However, it leads to larger internal fragmentation and more page I/O.

The notion of superpages was evolved to counter the problem created by low TLB reach. A *superpage* is like a page of a process, except that its size is a power of two multiple of the size of a page, and its start address in both the logical and physical address spaces is aligned on a multiple of its size. Most modern architectures permit a few standard superpage sizes and permit a TLB entry to be used for a page or for a superpage. This feature increases the TLB reach without increasing the size of the TLB, and helps to obtain a larger TLB hit ratio.

The virtual memory handler exploits the notion of superpages by adapting the size and number of superpages in a process to its execution characteristics. It may combine some pages of a process into a superpage of an appropriate size if the pages are accessed frequently, and satisfy the requirement of contiguity and address alignment in the logical address space. This action is called a *promotion*. The virtual

memory handler may have to relocate the individual pages in memory to satisfy contiguity and address alignment in the memory. A promotion increases the TLB reach, and releases some of the TLB entries that were assigned to individual pages of the new superpage.

If the virtual memory handler finds that some pages in a superpage are not accessed frequently, it may decide to disband the superpage into its individual pages. This step, called *demotion*, frees some memory, which can be used to load other pages. Thus, it has the potential to reduce the page fault frequency.

Address translation in a multiprogrammed system Figure 6.10 illustrates address translation in a multiprogrammed system. Page tables for many processes exist in the memory. The MMU contains a special register called the *PT address register* (PTAR) to point to the start of a page table. The MMU uses contents of this page table for address translation. For a logical address (p_i, b_i) , the MMU computes $\text{PTAR} + p_i \times l_{\text{PT_entry}}$, where $l_{\text{PT_entry}}$ is the length of a page table entry, to obtain address of the page table entry of page p_i .

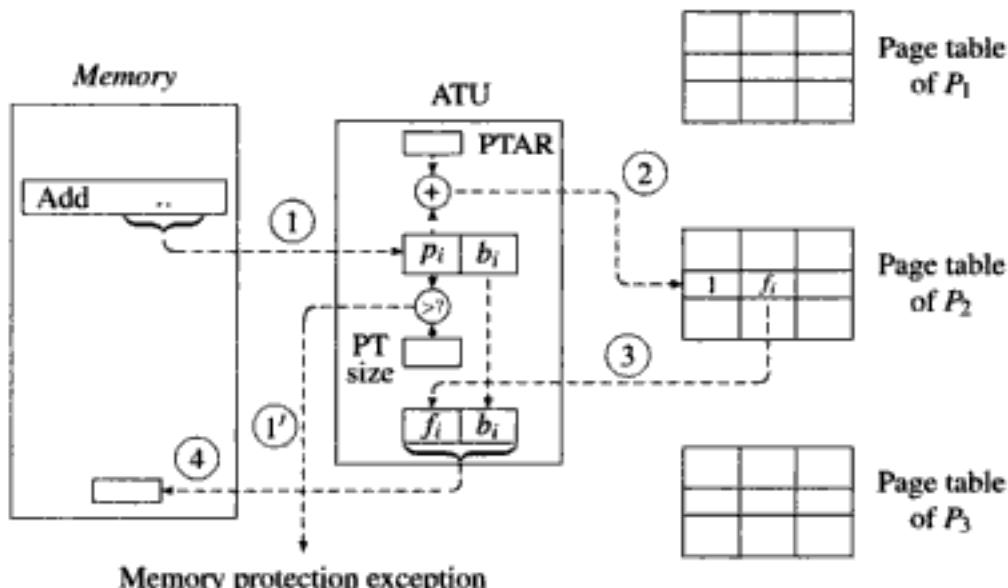


Fig. 6.10 Address translation in a multiprogrammed system

PTAR has to be loaded with the correct address whenever a process is scheduled. To facilitate this, the kernel can store the page table address of each process in its PCB. Since the TLB is used by all processes in the system, the kernel must erase its contents whenever a new process is scheduled.

6.2.3.2 Memory Protection

In a multiprogrammed paged virtual memory system, protection exceptions can arise only in the following two ways:

- *An address lies outside the logical address space of a process:* A process may use a logical address (p_i, b_i) that exceeds the size of its logical address space. Since the kernel considers a process to contain an integral number of pages, this can happen only if p_i is larger than the page number of the last page in it. In this case the MMU must abort address translation and raise a memory protection exception. Failure to do this would be disastrous as an out-of-range page number may point to an entry in the page table of some other process!
- *A process exceeds its access privileges:* A process might attempt to access a page in an invalid manner, e.g., it might attempt to modify a page to which it has only a read privilege. This can happen in a system that permits sharing of pages.

Steps shown in Table 6.3 implement memory protection. A special register, called the *PT size register*, is provided in the MMU for this purpose (see Figure 6.10). This register is loaded with the page number of the last page in the currently executing process. The MMU checks the page number in each logical address against this register before accessing the page table (see Step 2 in Table 6.3). It raises a memory protection interrupt if the page number exceeds page table size. The page table size information is maintained in the PCB of a process. It is loaded in the MMU when the process is scheduled.

Table 6.3 Steps in implementing protection in logical address space

1. View a logical address as a pair (p_i, w_i) .
2. Raise a memory protection exception if p_i exceeds contents of the PT size register (see the arrow marked 1' in Figure 6.10). Compute $\text{PTAR} + p_i \times l_{\text{PT_entry}}$ for accessing the page table entry for p_i .
3. Check the kind of access being made with the access privileges stored in the *misc info* field of p_i 's page table entry. Raise a memory protection exception in case of a conflict.

The access privileges of a process to a page can be stored in the *misc info* field of its page table entry. During address translation, the MMU can check the kind of access being made to a page against this information (see Step 3 in Table 6.3). The access privileges can be bit-encoded for efficient access. Each bit in the field would correspond to one kind of access to the page (e.g., read, write, etc.). It is set 'on' if the process possesses the corresponding access privilege to the page.

6.2.3.3 Support for Page Replacement

The virtual memory handler needs two kinds of information to make page replacement decisions that minimize page faults and the number of page-in and page-out operations:

1. Time when a page was last used.

2. Whether a page is *dirty*, i.e., whether a write operation has been performed on any byte in the page. (A page is *clean* if it is not dirty.)

The time of last use indicates how recently a page has been used in the process. This information is useful for deciding whether the page may be a part of the current locality of a process. As we shall see in Section 6.3 when we discuss page replacement policies, this information can also be used to differentiate between pages for the purpose of page replacement. Information about whether a page is clean or dirty is used to decide whether a page-out operation is necessary during page replacement. If a page is clean, an up-to-date copy already exists in the swap space of the process. Hence no page-out operation is needed; its copy in memory can be simply overwritten by performing a page-in operation for the page to be loaded. For a dirty page, a page-out operation must be performed because its copy in the swap space is outdated. A page-in operation for the new page to be loaded can be started only after the page-out operation is completed. A single bit is sufficient to indicate whether a page is clean or dirty, whereas a number of bits may be necessary to record the time of last use.

6.2.4 I/O Operations in a Paged Environment

An I/O operation in a process specifies the number of bytes to be transferred and the logical address of the data area, which is the area of memory that participates in the data transfer. The data area may span several pages of the process. To avoid disruption of the I/O operation, all pages of the data area should exist in memory throughout the operation, so the kernel loads all pages of the data area into the memory and puts an *I/O fix* on each page before starting the operation. The virtual memory handler does not replace any of these pages until the I/O fix is removed at the end of the I/O operation. A simple way to implement I/O fixing of pages is to add an I/O fix bit in the *misc info* field of each page table entry.

The I/O subsystem does not contain an MMU, hence I/O has to be performed using physical addresses. The I/O handler uses information from the page table of the process to replace the logical address of the data area by its physical address.

A key issue in implementing an I/O operation is that pages containing a data area may not be assigned contiguous physical addresses. The scatter-and-gather feature provided in most I/O subsystems can be used to handle this issue. A 'scatter read' operation can deposit parts of its data in non-contiguous areas of memory. For example, it can read the first few bytes from an I/O record into a page frame located in one part of memory and the remaining bytes into another page frame located in a different part of memory. Analogously, a 'gather write' can draw its data from non-contiguous memory areas and write it into one record on an I/O device. Example 6.2 illustrates how a scatter read operation is used to implement an I/O operation that spans two pages in a process.

If an I/O subsystem does not provide the scatter-and-gather feature, the kernel can handle the situation in two ways. It can either put pages containing the data area

contiguously in physical memory, or it can first read the data into a kernel area that has contiguous physical addresses and then copy it to the data area in the process. Analogous provisions can be made to support a write operation.

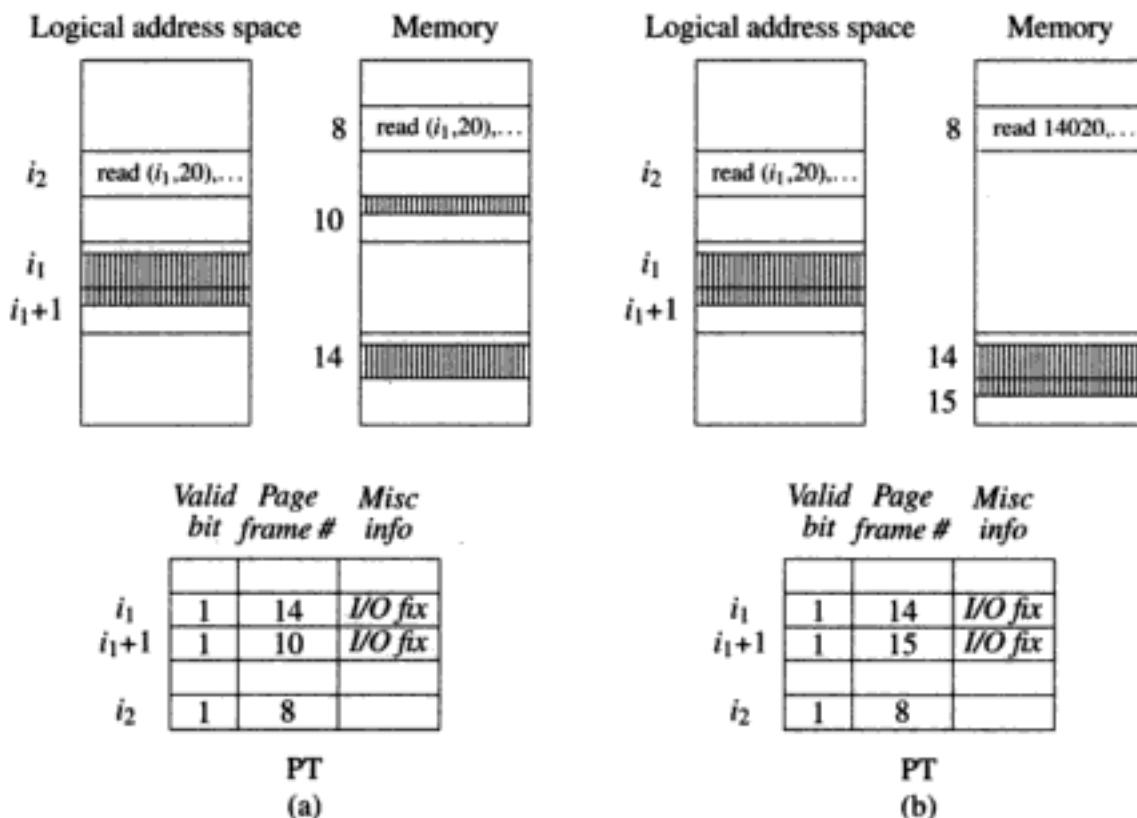


Fig. 6.11 I/O operations in virtual memory systems

Example 6.2 Page i_2 of a process P contains a system call ‘*perf.io* (read, 827, ($i_1, 520$))’, where 827 is the count of data bytes to be read, and ($i_1, 520$) is the logical address of the start of the data area. Figure 6.11 illustrates how the I/O operation is implemented. The page size is 1K bytes, hence the data area is situated in pages i_1 and $i_1 + 1$ of the process. Before initiating the I/O operation, the virtual memory handler is invoked to load pages i_1 and $i_1 + 1$ into memory. They are loaded into page frames 14 and 10 of the memory. The virtual memory handler puts an I/O fix on these pages by setting bits in the *misc info* field of their page table entries. These pages are not replaced until the I/O fix is removed at the end of the I/O operation. I/O handler now generates a scatter-read operation to read the first few bytes starting at byte offset 520 in page frame 14, and the remaining bytes starting at byte offset 0 in page frame 15. The I/O fix on pages 14 and 15 is removed when the I/O operation completes.

6.2.5 The Virtual Memory Handler

The virtual memory handler uses two data structures. The page table, whose entry format is shown in Figure 6.4, and the free frames list. With few exceptions, notably the IBM/370 architecture, the *ref info* and *modified* fields are set by the paging hard-

ware. All other fields are set by the virtual memory handler.

Table 6.4 summarizes the functions performed by the virtual memory handler. We discuss the first four functions in this Section. Other functions, viz. page replacement, allocation of physical memory to processes, and implementation of page sharing; are discussed in the next few Sections.

Table 6.4 Functions performed by the virtual memory handler

1. *Manage the logical address space of a program:* Organize the logical address space on the paging device, maintain the page table of the program, perform page-in and page-out operations.
2. *Manage the physical memory:* Keep track of occupied and free page frames in memory.
3. *Implement memory protection:* Maintain the information needed for memory protection.
4. *Collect information for page replacement:* Paging hardware provides information concerning page references. This information is maintained in appropriate data structures for use by the page replacement algorithm.
5. *Perform page replacement:* Perform replacement of a page when a page fault arises and all page frames in memory are occupied.
6. *Allocate physical memory to programs:* Decide how much memory should be allocated to a program and revise this decision from time to time to suit the needs of the program and the OS.
7. *Implement page sharing:* Arrange sharing of pages between programs.

Management of the logical address space of a process The virtual memory handler employs the following sub-functions for this purpose:

1. Organize a copy of the instructions and data of the process on the paging device
2. Maintain the page table
3. Perform page-in and page-out operations
4. Perform process initiation.

As mentioned earlier in Section 6.2, the logical address space of a process is organized in the swap space of the process when a process is initiated. When a reference to a page leads to a page fault, the page is loaded from the swap space using a page-in operation. When a dirty page is to be removed from the memory, a page-out operation copies it from the memory into a disk block in the swap space. Thus the swap space contains an up-to-date copy of every process page that is not in physical memory and of every page that is in memory but has not been modified since it was last loaded. For other pages the copy in memory is current and the swap space contains a stale (i.e., outdated) copy.

One issue in swap space management is size of the swap space for a process. Most virtual memory systems permit the logical address space of a process to grow dynamically during its execution. This can happen due to a variety of reasons. The

size of stack or PCD data areas may grow (see Section 5.3.2), the process may dynamically link more modules or may perform memory mapping of files (see Section 6.6). An obvious approach to handling dynamic growth of address spaces is to allocate swap space dynamically and non-contiguously; however, this approach faces the problem that the virtual memory handler may run out of swap space during the execution of a process.

To initiate the execution of a process, only the page containing its first instruction needs to be loaded in the memory. Other pages would be brought in on demand. Details of the page table and the page-in and page-out operations have been described earlier in Sections 6.2.1 and 6.2.

Management of the physical memory The free frames list is maintained at all times. A page frame is taken off this list to load a new page, and a frame is added to it when a page-out operation is performed. All page frames allocated to a process are added to the list when the process terminates.

Protection Protection information, which consists of access privileges of the process to its pages, is stored in the page table at process initiation time. The protection hardware is initialized by simply loading the page table start address and page table size information into appropriate MMU registers while scheduling a process. As seen before, this is achieved by putting these items of information in the PCB of a process. The information now gets transferred to the MMU registers as a part of the dispatching action.

Collection of information for page replacement The *ref info* field of an entry in the page table indicates when a page was last referenced and the *modified* field indicates whether it has been modified since it was last loaded in memory. Page reference information is useful only so long as a page exists in physical memory; it is reinitialized the next time a page-in operation is performed for the page.

Most architectures provide a single bit in the *ref info* field to collect page reference information. This information is not adequate to select the best candidate for page replacement, so the virtual memory handler may periodically reset the bit used to store this information. We discuss this aspect in Section 6.3.1.

Example 6.3 A virtual memory system has a physical memory that consists of eight page frames numbered from 0 to 7. Three processes P_1 , P_2 and P_3 exist in the system. Figure 6.12 shows the page table of process P_1 . P_1 is in execution. It consists of five pages numbered 0 to 4. Page frames 2, 7 and 4 have been allocated to its pages 1, 2 and 3 respectively. The other pages of P_1 do not exist in memory presently. Page frames 1 and 5 have been allocated to pages 8 and 4 of process P_2 , respectively, and page frames 0, 3 and 6 have been allocated to pages 4, 1 and 3 of process P_3 , respectively. Thus, all page frames of physical memory have been allocated to processes and no free page frames exist in the system.

Figure 6.12(a) illustrates the situation in the system at time instant t_{11}^+ . Only page table of P_1 is shown in the figure since process P_1 is in execution. Contents of the *ref*

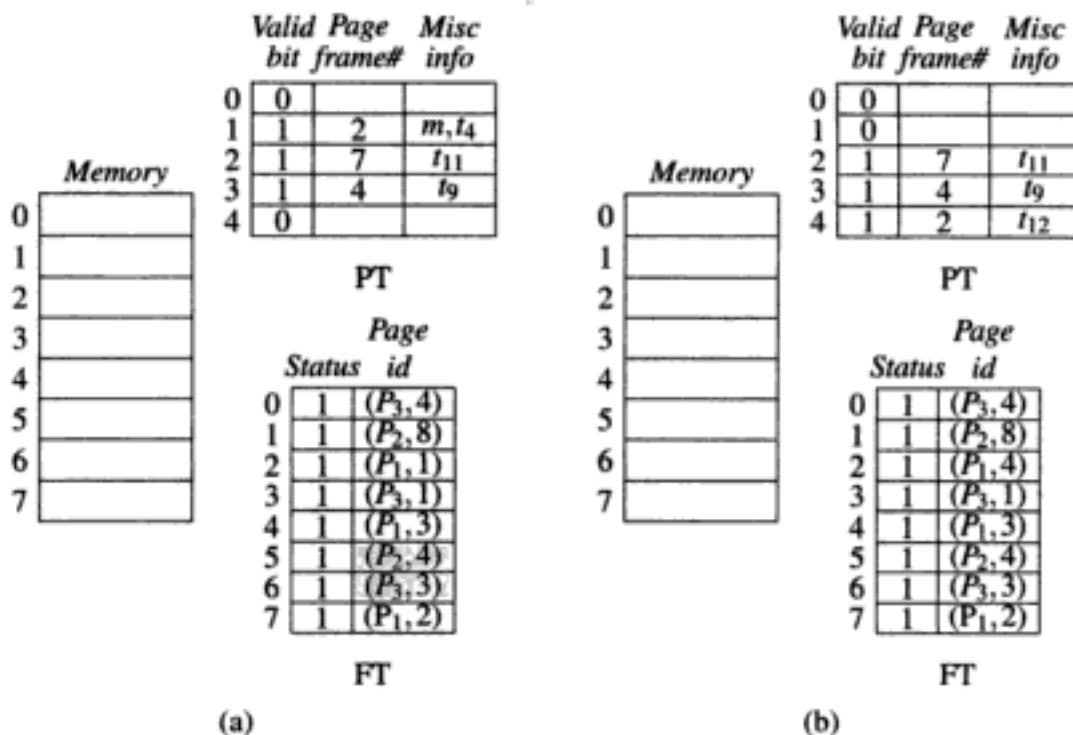


Fig. 6.12 Operation of the virtual memory handler

info and *modified* fields are shown in the *misc info* field. Pages 1, 2 and 3 were last referenced at time instants t_4, t_{11} and t_9 , respectively. Page 1 was modified sometime after it was last loaded, so the *misc info* field of its page table entry contains the information m, t_4 .

At time instant t_{12} process P_1 gives rise to a page fault for page 4. Since all page frames in physical memory are occupied, the virtual memory handler decides to replace page 1 of the process. The page has been modified since it was last loaded, hence a page-out operation is necessary. The *page frame #* field of the page table entry of page 1 indicates that the page exists in page frame 2. The virtual memory handler now performs a page-out operation to write contents of page frame 2 into the swap area reserved for page 1 of P_1 . A page-in operation is now initiated for page 4 of P_1 . At the end of the operation, the page table entry of page 4 is modified to indicate that it exists in memory in page frame 2, and execution of P_1 is resumed. P_1 immediately makes a reference to page 4, hence the page reference information of page 4 indicates that it was last referenced at t_{12} . Figure 6.12(b) indicates the virtual memory handler data structures at time instant t_{12}^+ .

6.2.6 Practical Page Table Organizations

Inverted page tables Data structures of the virtual memory handler present a good example of the space-time tradeoff in systems programs. Large processes have large page tables, so the virtual memory handler data structures occupy a lot of memory. To get an idea of the memory requirements of a page table, consider a contemporary computer system using logical addresses that contain 32 bits or more. Size of the

logical address space is 4G bytes or more. If the page size is 1K bytes, 4 million pages can exist in a process. Each page table entry is a few bytes in length, hence the page table of a process can occupy several Mbytes.

To conserve memory some systems dispense with page tables altogether. The frame table, which is also called the *inverted page table* (IPT), is used for address translation as well. This approach saves a considerable amount of memory since the size of IPT is governed only by size of the physical memory; it is independent of the number and sizes of processes being executed. However, use of IPT slows down address translation and also slows down the virtual memory handler functions. IPTs are used in many systems including IBM RS 6000 and AS 400 systems. Information useful in page replacement decisions is stored in IPT.

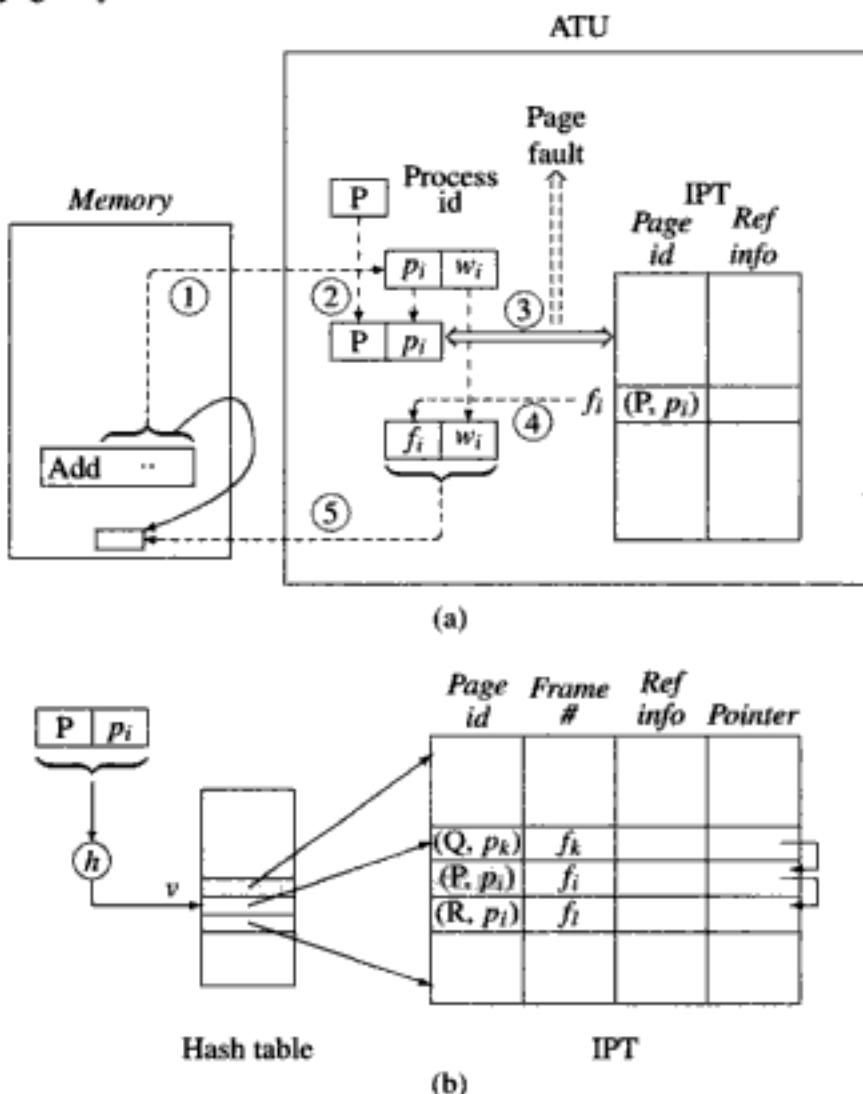


Fig. 6.13 Inverted page tables: (a) Concept, (b) implementation using a hash table

Figure 6.13(a) illustrates use of IPT in address translation. Each entry of IPT contains a pair (process id, page number). While scheduling a process, the scheduler copies the id of the process from its PCB into a register of the MMU. Let this id be P . Address translation for a logical address in process P consists of the following actions:

1. The pair (p_i, b_i) is extracted from a logical address.
2. Using the process id P , the pair (P, p_i) is formed.
3. The pair (P, p_i) is searched in IPT. A page fault is raised if the page is not found in IPT.
4. If the pair exists in entry f_i of IPT, the page frame number f_i is copied for use in address translation.
5. Address translation is completed using f_i and b_i .

These steps are shown as circled numbers 1 to 5 in Figure 6.13(a).

Search for (P, p_i) in IPT slows down address translation. An expensive solution to this problem is to store the IPT in an associative memory that can locate the entry of (P, p_i) through an associative search. A more practical solution is to use a hash table to speed up the search.

Figure 6.13(b) shows an inverted page table using a hash table. A pair consisting of a process id and a page number is stored in the *page id* field, and the frame number allocated to the page is stored in the *frame #* field of the inverted page table. The hash table is used to locate the entry of a required pair (P, p_i) in the inverted page table efficiently during address translation. We present this inverted page table organization in three steps. First, we discuss a simple hash table organization. Then, we discuss how the inverted page table is constructed by the virtual memory handler. Finally, in Example 6.4, we discuss how the inverted page table is used by the MMU during address translation.

To hash a pair (P, p_i) , we first concatenate the bit strings representing P and p_i to obtain a larger bit string. We now interpret this bit string as an integer number x , and apply the following hash function h to it:

$$h(x) = \text{remainder } (\frac{x}{a})$$

where a is the size of the hash table, which is typically some prime number $> \#frames$, the number of page frames in memory. $h(x)$, which is in the range $0 \dots a - 1$, is an entry number in the hash table. Let v designate its value. Since the total number of pages of all processes in memory is much larger than $\#frames$, hashing of many process id–page id pairs may produce the same value v . To incorporate this possibility, the pair (P, p_i) is not entered in the v^{th} entry of the hash table. Instead, it is stored in an entry of the inverted page table, and all entries in the inverted page table that are occupied by such process id–page id pairs that hash into the value v are entered into a linked list. The v^{th} entry of the hash table points to the first entry in this linked list. Similarly, process id–page id pairs that hash into some other entry of the hash

table, say the w^{th} entry, form a separate linked list that is pointed to by the w^{th} entry of the hash table.

The inverted page table is constructed and maintained by the virtual memory handler as follows: When page p_i of process P is loaded in page frame f_i in memory, the virtual memory handler constructs the pair (P, p_i) for it. It hashes this pair to obtain an entry number, say v . The virtual memory handler then selects a free entry in the inverted page table and enters the pair (P, p_i) in the *page id* field and the frame number f_i in the *frame #* field of this entry. It now adds this entry in the linked list starting on the v^{th} entry of the hash table as follows: It copies the v^{th} entry of the hash table into the *pointer* field of the inverted page table entry just formed and enters the entry number of the inverted page table entry just formed into the v^{th} entry of the hash table. When this page is removed from memory, the virtual memory handler deletes its entry from the linked list starting on the v^{th} entry of the hash table. In Figure 6.13(b), the three pages (R, p_1) , (P, p_i) and (Q, p_k) , which were loaded into page frames f_1 , f_i and f_k , respectively, are entered into the same linked list because they all hashed into the v^{th} entry of the hash table. Example 6.4 describes how the MMU uses the inverted page table during address translation.

Example 6.4 Address translation using the IPT illustrated in Figure 6.13(b) is implemented as follows: Let the logical address be (p_i, b_i) . The pair (P, p_i) is hashed to obtain an entry number v in the hash table. The linked list starting on this entry is searched. The pair (P, p_i) does not match with the pair (Q, p_k) found in the first entry of the list, hence the next entry in the list is accessed using the pointer field. A match is now detected and the logical address (f_i, b_i) is formed using the frame id f_i found in the entry. If hashing of a pair (S, p_j) formed for a logical address (p_j, w_j) of process S also gives the same entry number v in the hash table, a page fault would be generated after searching all entries in the linked list starting on v^{th} entry of the hash table.

Two-level page tables We have so far assumed that the entire page table of a process is always in memory. This is impractical for large page tables. For example, in a system using 32 bit logical addresses and a page size of 1K bytes, a process can have 4 million pages. If the size of a page table entry is 4 bytes, a page can contain 256 page table entries, hence the page table would contain 16,000 pages! Memory requirements of a page table can be reduced by paging the page table itself and loading its pages on demand just like pages of processes. This approach gives rise to a two-tiered arrangement in which a higher level page table contains entries pointing to pages of the page table and the page table contains entries that point to pages of a process. This arrangement reduces memory commitment to a page table since only some of its parts are in memory at any time.

Figure 6.14 illustrates the concept of a two-level page table. The memory contains two kinds of pages—process pages and pages of the page table, i.e., PT pages, i.e., pages of the page table. For address translation of a logical address (p_i, b_i) in a process P , page p_i of process P should exist in memory. Also, the page of P 's page table, which contains the entry for page p_i , should also exist in memory. In

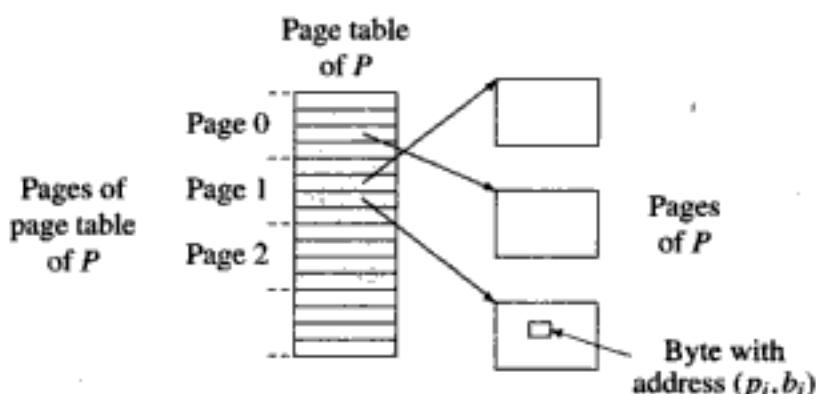


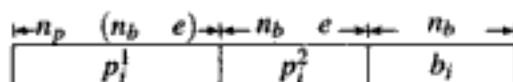
Fig. 6.14 Concept of a two-level page table

Figure 6.14, this is page 1 of P's page table. During address translation, the MMU has to determine id of this PT page and ensure its presence in memory.

As mentioned in Section 6.2.1, the page number and byte offsets of a logical address (p_i, b_i) are represented in n_p and n_b bits. The length of a logical address, i.e., l_i , is $n_p + n_b$. If the number of bytes occupied by each page table entry is a power of two, the number of page table entries that fit in one page would also be a power of two. Let each page table entry occupy 2^e bytes, then the number of page table entries in one page of a page table is $2^{n_b}/2^e$, i.e., 2^{n_b-e} . Hence id of the PT page that contains the page table entry for page p_i of a process is given by $\lfloor \frac{p_i}{2^{n_b-e}} \rfloor$ where $\lfloor \dots \rfloor$ indicates a truncated integer value. This number is contained in $n_p - (n_b - e)$ higher order bits in p_i . The lower order $n_b - e$ bits indicate which entry in this page contains information concerning page p_i .

Address translation requires two memory accesses to access the higher order page table and the page table of the process. Figure 6.15 illustrates address translation for a logical address (p_i, b_i) . It consists of the following steps:

1. The address (p_i, b_i) is regrouped into three fields



The contents of these fields are p_i^1 , p_i^2 and b_i , respectively.

2. p_i^1 is a page number in the higher level page table. This page contains p_i 's entry in the page table. The MMU checks whether this page exists in memory and raises a page fault if it does not. The page fault is serviced by virtual memory handler to load the page in memory.
3. p_i^2 is the entry number for p_i in this page. The MMU uses information in this entry to check whether page p_i exists in memory and raises a page fault if it does not. The virtual memory handler services the page fault and loads page p_i in memory.

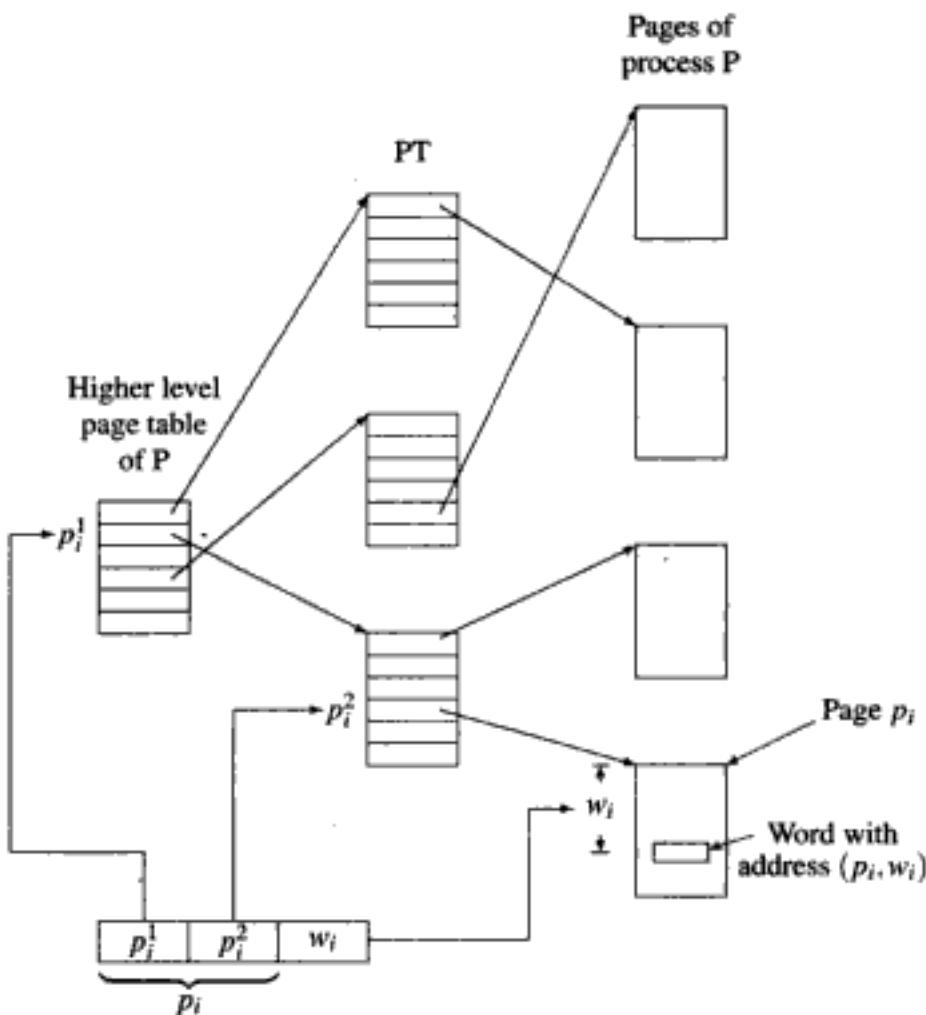


Fig. 6.15 Two-level page table organization

4. Contents of p_i 's page table entry are used to perform address translation.

The Intel 80386 architecture uses two-level paging. The higher level page table is called a page directory. A page reference is implemented using an entry in the page directory and an entry in the page table. The hardware turns on the accessed bits in these entries. If the reference is a write reference, the hardware additionally turns on the dirty bit in the page table entry.

Multi-level page tables Multi-level page table organizations are used when size of the higher level page table in a two-level page table organization is itself large. As seen earlier, in a system using 32 bit logical addresses, a page size of 1K bytes, and page table entries of 4 bytes, the page table would contain 16,000 pages. The size of the higher level page table would be 64K bytes. To reduce memory commitment for page tables further, the higher level page table can itself be paged. This arrangement results in a three-level page table structure.

Address translation in this environment is performed by an obvious extension of address translation in two-level page tables. A logical address (p_i, b_i) is split into four components p_i^1, p_i^2, p_i^3 and b_i , and the first three components are used to address three levels of the page table. Thus address translation requires up to three memory accesses. In computer systems using 64 bit addresses even the highest level page table in a three-level page table organization may become too large. Four-level page tables are used to overcome this problem. Three and four-level page tables have been used in the Sun Sparc and Motorola 68030 architectures, respectively. Both these architectures use 32 bit logical addresses.

6.3 PAGE REPLACEMENT POLICIES

A page replacement operation becomes necessary if a free page frame does not exist when a page fault occurs during execution of some process $proc_i$. A page replacement decision has two important aspects:

1. Whether a page belonging to process $proc_i$ itself should be replaced, or whether the number of page frames allocated to $proc_i$ should be increased.
2. What criteria should be used to select the page to be replaced.

As discussed later in this Section, these aspects are handled separately by the virtual memory handler. Thus when a page fault occurs, virtual memory handler replaces a page of the same process if all page frames allocated to the process are occupied. Decision to increase or decrease memory allocation to a process is taken independently.

Page replacement is a policy decision based on the page reference information available in the page table. It is implemented using page-in and page-out operations as mechanisms. Figure 6.16 depicts the arrangement of policy and mechanism modules of the virtual memory handler. The page-in and page-out mechanisms interact with the paging hardware to implement their functionalities. They also update the page and frame tables to reflect their actions.

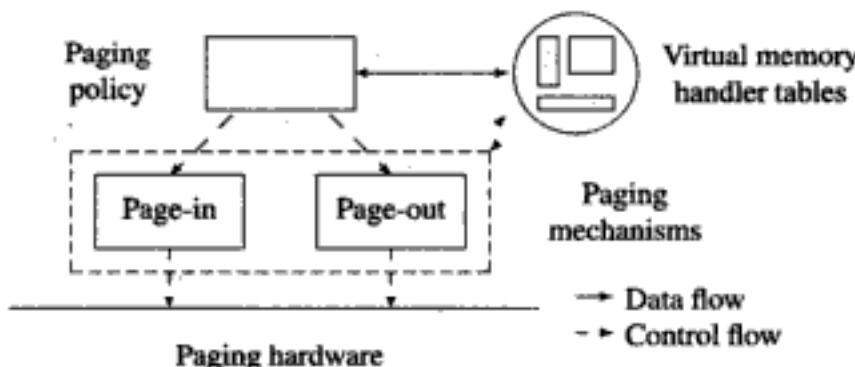


Fig. 6.16 Virtual memory handler modules in a paged virtual memory system

We evaluate the following three page replacement policies to see how well they consider the current locality of a process while making page replacement decisions:

- Optimal page replacement policy
- First-in-First-out (FIFO) page replacement policy
- Least Recently Used (LRU) page replacement policy.

We use the following notation in our discussion:

$alloc_i$: Number of page frames allocated to process $proc_i$

Page reference strings A page reference string of a process is the sequence of page numbers that appear in logical addresses generated by a process. It string can be constructed through software intervention during the execution of a process. An empty sequence of page numbers is created when a process is about to be initiated. For every logical address used during the execution of the process, the page number in the logical address is added to the end of the sequence. These actions involve substantial CPU overhead, so page reference strings are constructed only for processes that are used to study the performance of page replacement policies. Note that execution of a process with different data can lead to different page reference strings.

We introduce the notion of a logical time instant to help us refer to a specific page reference in a page reference string. A *logical time instant* is the time in a logical clock of the process, i.e., in a clock that is advanced only when the process is in the *running* state. We designate logical time instants as t_1, t_2, \dots . A logical time instant is used as an index into a page reference string. For ease of use, logical time instants are shown as a string associated with the page reference string. This string is called the *reference-time string*. Example 6.5 illustrates the page reference string and the reference time string for a process.

Example 6.5 A computer system provides instructions with length 4 bytes each, and uses a page size of 1K bytes. It executes the following nonsense program in which A and B are assumed to exist in pages 2 and 5, respectively:

```

START    2040
READ    B
LOOP    MOVER  AREG, A
        SUB    AREG, B
        BC     LT, LOOP
        ...
STOP
A      DS     2500

B      DS     1
END

```

The page reference string and the reference-time string for the process are as follows:

page reference string : 1, 5, 1, 2, 2, 5, 2, 1, ...
reference time string : $t_1, t_2, t_3, t_4, t_5, t_6, t_7, t_8 \dots$

The logical address of the first instruction is 2040, hence it lies in page 1. The first page reference in the string is therefore 1. This occurs at logical time instant t_1 . B,

the operand of the instruction, is situated in page 5, so the second page reference in the string is 5. The next instruction is located in page 1 and refers to A, which is located in page 2, hence the next two page references are to pages 1 and 2. The next two instructions are located in page 2 and the instruction with label LOOP is located in page 1, hence the next four page references are to pages 2, 5, 2 and 1, respectively.

Optimal page replacement Optimal page replacement implies making page replacement decisions in such a manner that the total number of page faults during the execution of a process is the minimum possible, i.e., no other sequence of page replacement decisions can lead to a smaller number of page faults. To achieve optimal page replacement, at each page fault the page replacement policy should consider all alternative page replacement decisions, analyze their implications for future page faults, and then select the best alternative. Belady[1966] showed that this can be achieved simply by using the following rule: At a page fault replace the page whose next reference is farthest in the page reference string.

Optimal page replacement is infeasible because the virtual memory handler does not have knowledge of the future behavior of a process. However, it provides a useful basis for evaluating performance of other page replacement policies.

FIFO page replacement At every page fault the FIFO page replacement policy replaces the page that was loaded into memory earlier than any other page of the process. To facilitate FIFO page replacement, the PIT entry of a page is used to record the time when the page was last loaded into memory. When a page fault occurs, this information is used to determine $p_{earliest}$, the page that was loaded earlier than any other page of the process. This page is replaced by the required page.

LRU page replacement The LRU policy uses the principle of locality of reference as the basis for its replacement decisions. Its operation can be described as follows: At every page fault the *least recently used* (LRU) page is replaced by a new page. The page table entry of a page records the time when the page was last referenced. This information is initialized when a page is loaded, and it is modified every time the page is referenced. When a page fault occurs, PIT is searched to locate the page p_{lru} whose last reference is earlier than that of every other page. This page is replaced with the new page.

Example 6.6 A page reference string and the reference-time string for a process P are as follows:

$$\text{page reference string} : 0, 1, 0, 2, 0, 1, 2, \dots \quad (6.3)$$

$$\text{reference time string} : -t_1, t_2, t_3, t_4, t_5, t_6, t_7 \dots \quad (6.4)$$

Figure 6.17 illustrates operation of the optimal, FIFO and LRU page replacement policies for this page reference string with $alloc = 2$. For convenience, we show only two fields of the page table, viz. *valid bit* and *misc info* and assume that the page reference information is stored in the *misc info* field.

The left column shows operation of optimal page replacement. Page reference information is not shown in the page table since information concerning past references is not needed for optimal page replacement. A page fault occurs at logical time instant t_4 when page 2 is referenced. Page 1 is replaced because its next reference is farther in the page reference string than that of page 0. At time t_6 page 1 replaces page 0 because page 0's next reference is farther than that of page 2.

		Optimal			FIFO			LRU		
Time	Page instant ref	Valid	Misc	Replace-	Valid	Misc	Replace-	Valid	Misc	Replace-
		bit	info	ment	bit	info	ment	bit	info	ment
t_1	0	0	1		0	1	t_1	0	1	t_1
		1	0		1	0		1	0	
		2	0		2	0		2	0	
t_2	1	0	1		0	1	t_1	0	1	t_1
		1	1		1	1	t_2	1	1	t_2
		2	0		2	0		2	0	
t_3	0	0	1		0	1	t_1	0	1	t_3
		1	1		1	1	t_2	1	1	t_2
		2	0		2	0		2	0	
t_4	2	0	1		0	0		0	1	t_3
		1	0		1	1	t_2	1	0	
		2	1		2	1	t_4	2	1	t_4
t_5	0	0	1		0	1	t_5	0	1	t_5
		1	0		1	0		1	0	
		2	1		2	1	t_4	2	1	t_4
t_6	1	0	0		0	1	t_5	0	1	t_5
		1	1		1	1	t_6	1	1	t_6
		2	1		2	0		2	0	
t_7	2	0	0		0	0		0	0	
		1	1		1	1	t_6	1	1	t_6
		2	1		2	1	t_7	2	1	t_7

Fig. 6.17 Comparison of page replacement policies with $alloc = 2$.

The middle column of Figure 6.17 illustrates the operation of the FIFO replacement policy. A page fault occurs when page 2 is referenced at time t_4 . *Misc info* field shows that page 0 was loaded earlier than page 1, hence page 0 is replaced by page 2.

The last column of Figure 6.17 illustrates the operation of the LRU replacement policy. The *misc info* field of the page table indicates when a page was last referenced. At

time t_4 , page 1 is replaced by page 2 because the last reference of page 1 is later than the last reference of page 0. The total number of page faults suffered by the optimal, FIFO and LRU policies are 4, 6 and 5, respectively.

When we analyze why LRU performed better than FIFO in Example 6.6, we find that FIFO removed page 0 at time t_4 but LRU did not do so because it had been referenced later than page 1. This decision is consistent with the principle of locality, which indicates that because Page 0 was referenced more recently than Page 1, it has a higher probability of being referenced than Page 1. Thus, LRU does not interfere with the locality of a process whereas FIFO does.

It is interesting to consider whether the LRU policy possesses the desirable page fault characteristic of Figure 6.7, i.e., whether the LRU policy can guarantee that the page fault rate would not be higher if the process were to be executed with more memory allocated to it. We use the following notation to discuss this issue.

$\{p_i\}_n^k$: Set of pages existing in memory at time instant t_k^+ if $alloc_i = n$ all through the execution of process $proc_i$ (t_k^+ implies a time after time instant t_k but before t_{k+1}).

Definition 6.3 (Stack property) A page replacement policy possesses the stack property if

$$\{p_i\}_n^k \subseteq \{p_i\}_m^k \text{ if } n < m.$$

Consider 2 executions of process $proc_i$, one with $alloc_i = n$ all through the execution, and another with $alloc_i = m$, such that $n < m$. If a page replacement policy possesses the stack property, then at identical points during these executions of $proc_i$ (i.e., at identical logical time instants) all pages that were in memory when $alloc_i = n$ would also be in memory when $alloc_i = m$.

It can be shown that if a page replacement policy possesses the stack property, its page fault characteristic is analogous to the desirable page fault characteristic discussed in Section 6.2.2. Consider a logical time instant t_g during executions of $proc_i$ with n and m pages, respectively. For brevity we refer to these executions as 'execution n ' and 'execution m '. If the page replacement policy possesses the stack property, all n pages that are in memory when $alloc_i = n$ are also in memory when $alloc_i = m$. In addition, the memory also contains $m - n$ other pages of the process. If any of these pages are referenced in the next few page references of $proc_i$, page faults would occur in execution n but not in execution m ! Thus, the page fault rate in execution n would be larger than in execution m . Hence the page fault characteristic of Figure 6.7 holds. The page fault rates would be identical if these pages are not referenced in the next few page references; however, in no case would the page fault rate increase when the memory allocation for a process is increased.

If a page replacement policy does not possess the stack property, then $\{p_i\}_m^k$ does not contain some page(s) contained in $\{p_i\}_n^k$. References to these pages would result

in page faults, so it is possible that execution m has a larger number of page faults than execution n .

The FIFO page replacement policy does not possess the stack property, but the LRU policy does. Example 6.7 illustrates these facts.

Example 6.7 Consider the following page reference and reference time strings for a process:

$$\text{page reference string : } 5, 4, 3, 2, 1, 4, 3, 5, 4, 3, 2, 1, 5, \dots \quad (6.5)$$

$$\text{reference time string : } t_1, t_2, t_3, t_4, t_5, t_6, t_7, t_8, t_9, t_{10}, t_{11}, t_{12}, t_{13}, \dots \quad (6.6)$$

FIFO	$alloc_f = 3$	<table border="1"> <tbody> <tr><td></td><td></td><td>3</td><td>3</td><td>3</td><td>4</td><td>4</td><td>4</td><td>4</td><td>4</td><td>2</td><td>2</td><td>2</td></tr> <tr><td></td><td></td><td>4</td><td>4</td><td>1</td><td>1</td><td>1</td><td>5</td><td>5</td><td>5</td><td>5</td><td>5</td><td>5</td></tr> <tr><td></td><td></td><td>5</td><td>5</td><td>2</td><td>2</td><td>2</td><td>3</td><td>3</td><td>3</td><td>3</td><td>1</td><td>1</td></tr> <tr><td>5</td><td>4*</td><td>3*</td><td>2*</td><td>1*</td><td>4*</td><td>3*</td><td>5*</td><td>4</td><td>3</td><td>2*</td><td>1*</td><td>5</td></tr> </tbody> </table>			3	3	3	4	4	4	4	4	2	2	2			4	4	1	1	1	5	5	5	5	5	5			5	5	2	2	2	3	3	3	3	1	1	5	4*	3*	2*	1*	4*	3*	5*	4	3	2*	1*	5													
		3	3	3	4	4	4	4	4	2	2	2																																																							
		4	4	1	1	1	5	5	5	5	5	5																																																							
		5	5	2	2	2	3	3	3	3	1	1																																																							
5	4*	3*	2*	1*	4*	3*	5*	4	3	2*	1*	5																																																							
	$alloc_f = 4$	<table border="1"> <tbody> <tr><td></td><td></td><td>2</td><td>2</td><td>2</td><td>2</td><td>2</td><td>2</td><td>3</td><td>3</td><td>3</td><td>3</td><td>3</td></tr> <tr><td></td><td></td><td>3</td><td>3</td><td>3</td><td>3</td><td>3</td><td>3</td><td>4</td><td>4</td><td>4</td><td>4</td><td>5</td></tr> <tr><td></td><td></td><td>4</td><td>4</td><td>4</td><td>4</td><td>4</td><td>5</td><td>5</td><td>5</td><td>5</td><td>1</td><td>1</td></tr> <tr><td>5</td><td>5</td><td>5</td><td>5</td><td>1</td><td>1</td><td>1</td><td>1</td><td>1</td><td>1</td><td>2</td><td>2</td><td>2</td></tr> <tr><td>5</td><td>4*</td><td>3*</td><td>2*</td><td>1*</td><td>4</td><td>3</td><td>5*</td><td>4*</td><td>3*</td><td>2*</td><td>1*</td><td>5*</td></tr> </tbody> </table>			2	2	2	2	2	2	3	3	3	3	3			3	3	3	3	3	3	4	4	4	4	5			4	4	4	4	4	5	5	5	5	1	1	5	5	5	5	1	1	1	1	1	1	2	2	2	5	4*	3*	2*	1*	4	3	5*	4*	3*	2*	1*	5*
		2	2	2	2	2	2	3	3	3	3	3																																																							
		3	3	3	3	3	3	4	4	4	4	5																																																							
		4	4	4	4	4	5	5	5	5	1	1																																																							
5	5	5	5	1	1	1	1	1	1	2	2	2																																																							
5	4*	3*	2*	1*	4	3	5*	4*	3*	2*	1*	5*																																																							
LRU	$alloc_l = 3$	<table border="1"> <tbody> <tr><td></td><td></td><td>3</td><td>3</td><td>3</td><td>4</td><td>4</td><td>4</td><td>4</td><td>4</td><td>1</td><td>1</td><td>1</td></tr> <tr><td></td><td></td><td>4</td><td>4</td><td>1</td><td>1</td><td>1</td><td>5</td><td>5</td><td>5</td><td>2</td><td>2</td><td>2</td></tr> <tr><td></td><td></td><td>5</td><td>5</td><td>2</td><td>2</td><td>2</td><td>3</td><td>3</td><td>3</td><td>3</td><td>3</td><td>5</td></tr> <tr><td>5</td><td>4*</td><td>3*</td><td>2*</td><td>1*</td><td>4*</td><td>3*</td><td>5*</td><td>4</td><td>3</td><td>2*</td><td>1*</td><td>5*</td></tr> </tbody> </table>			3	3	3	4	4	4	4	4	1	1	1			4	4	1	1	1	5	5	5	2	2	2			5	5	2	2	2	3	3	3	3	3	5	5	4*	3*	2*	1*	4*	3*	5*	4	3	2*	1*	5*													
		3	3	3	4	4	4	4	4	1	1	1																																																							
		4	4	1	1	1	5	5	5	2	2	2																																																							
		5	5	2	2	2	3	3	3	3	3	5																																																							
5	4*	3*	2*	1*	4*	3*	5*	4	3	2*	1*	5*																																																							
	$alloc_l = 4$	<table border="1"> <tbody> <tr><td></td><td></td><td>2</td><td>2</td><td>2</td><td>5</td><td>5</td><td>5</td><td>5</td><td>1</td><td>1</td><td>1</td><td>1</td></tr> <tr><td></td><td></td><td>3</td><td>3</td><td>3</td><td>3</td><td>3</td><td>3</td><td>3</td><td>3</td><td>3</td><td>3</td><td>3</td></tr> <tr><td></td><td></td><td>4</td><td>4</td><td>4</td><td>4</td><td>4</td><td>4</td><td>4</td><td>4</td><td>4</td><td>4</td><td>5</td></tr> <tr><td>5</td><td>5</td><td>5</td><td>5</td><td>1</td><td>1</td><td>1</td><td>1</td><td>1</td><td>1</td><td>2</td><td>2</td><td>2</td></tr> <tr><td>5</td><td>4*</td><td>3*</td><td>2*</td><td>1*</td><td>4</td><td>3</td><td>5*</td><td>4</td><td>3</td><td>2*</td><td>1*</td><td>5*</td></tr> </tbody> </table>			2	2	2	5	5	5	5	1	1	1	1			3	3	3	3	3	3	3	3	3	3	3			4	4	4	4	4	4	4	4	4	4	5	5	5	5	5	1	1	1	1	1	1	2	2	2	5	4*	3*	2*	1*	4	3	5*	4	3	2*	1*	5*
		2	2	2	5	5	5	5	1	1	1	1																																																							
		3	3	3	3	3	3	3	3	3	3	3																																																							
		4	4	4	4	4	4	4	4	4	4	5																																																							
5	5	5	5	1	1	1	1	1	1	2	2	2																																																							
5	4*	3*	2*	1*	4	3	5*	4	3	2*	1*	5*																																																							

Fig. 6.18 Performance of FIFO and LRU page replacement

Figure 6.18 shows the behavior of the FIFO and LRU page replacement policies. Each column of boxes shows page frames allocated to the process. Numbers in the boxes indicate pages in memory *after* executing the memory reference marked under the column. Page references marked with '*' cause page faults and result in page replacement.

We have $\{p_i\}_4^{12} = \{2, 1, 4, 3\}$, while $\{p_i\}_3^{12} = \{1, 5, 2\}$ for FIFO page replacement. Thus, FIFO page replacement does not possess the stack property. This leads to a page fault at t_{13} when $alloc_f = 4$, but not when $alloc_f = 3$. Thus, a total of 10 page faults arise in 13 time instants when $alloc_f = 4$, while 9 page faults arise when $alloc_f = 3$. For LRU, we see that $\{p_i\}_3 \subset \{p_i\}_4$ at all logical time instants.

Figure 6.19 illustrates the page fault characteristic of FIFO and LRU page replacement for page reference string (6.5). For simplicity the vertical axis shows

the total number of page faults rather than the page fault frequency. Figure 6.19(a) illustrates a surprising aspect of FIFO page replacement—the page fault frequency of a process increases as its memory allocation is increased. This anomalous behavior was first reported by Belady, hence it is known as Belady's anomaly. The virtual memory handler cannot use FIFO page replacement because increasing the allocation to a process may increase the page fault frequency of a process. This feature would make it impossible to tackle thrashing in the system. The graph for LRU page replacement shows that the number of page faults is a non-increasing function of $alloc$. Thus it is possible to combat thrashing by increasing the value of $alloc$.

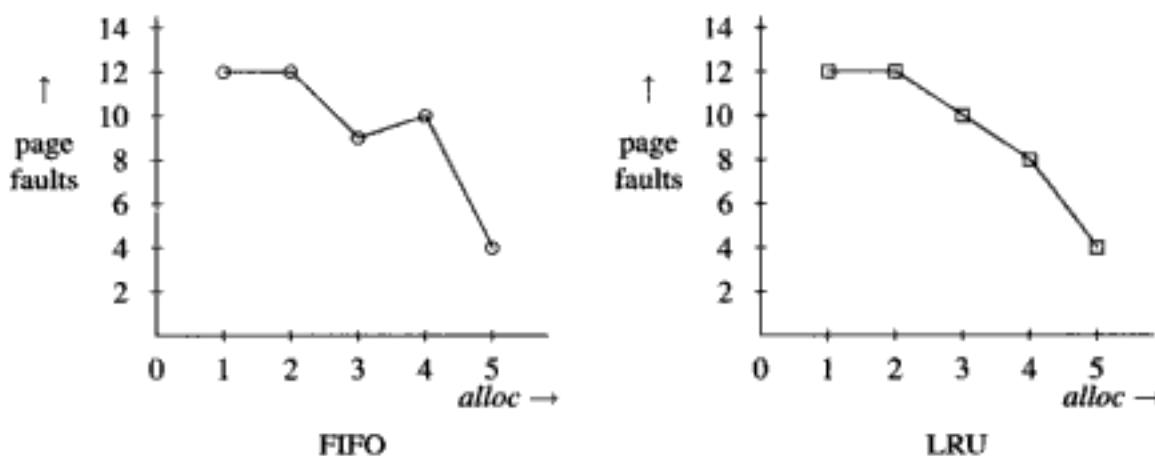


Fig. 6.19 (a) Belady's anomaly in FIFO page replacement, (b) LRU page replacement

6.3.1 Practical Page Replacement Policies

LRU should be an automatic choice for implementation in a virtual memory handler because it possesses the stack property. However, LRU page replacement is not feasible because most computer systems do not provide sufficient bits in PIT entries to store the time of last reference. Therefore virtual memory handlers have to use algorithms that can be implemented using the meager page replacement support provided by a computer system. These algorithms are crude approximations to either LRU or FIFO page replacement.

LRU approximations Many computer systems provide a single reference bit to collect page reference information. While making a page replacement decision, the virtual memory handler can use this information to find a page that has not been referenced for some time. Such a page is replaced. If the virtual memory handler finds that all pages have their reference bits 1 it resets the bits for all pages and arbitrarily selects one page for replacement. This is a crude approximation to LRU page replacement. Some obvious improvements are possible in this approach. The reference bit does not give true LRU information, so the virtual memory handler cannot differentiate between pages that have their reference bits 0. To reduce the

cost of replacement, the virtual memory handler can check the modify bits of all pages whose reference bits are 0 and select a page that has not been modified. The case study of the Unix virtual memory handler in Section 6.7 describes some other LRU approximations used in practice.

FIFO approximations Support from paging hardware is not crucial in FIFO page replacement because FIFO ordering of pages can be maintained by the virtual memory handler itself. It can maintain a queue in which pages appear in the order in which they were last loaded in physical memory. When a page fault occurs, the first page in the queue is removed from memory and its entry is removed from the queue. Id of a newly loaded page is added to the end of the queue.

A refinement called *second-chance algorithm* is used in some virtual memory handlers. In this refinement a crude effort is made to differentiate between frequently accessed pages and non-so-frequently accessed pages using a single reference bit. The reference bit of a page is reset when it is loaded in memory. It is set every time the page is accessed. At a page fault, the page at the head of the queue is not replaced if it has its reference bit set. Instead, its reference bit is reset and its entry is shifted to the end of the queue. In effect, the page is allowed to survive in physical memory but in future it will receive the same treatment as if it had been loaded in memory at the current instant of time. The page that is at the head of the modified queue is now considered for replacement in a similar manner.

Thus pages that are referenced frequently keep receiving a new lease of life in the physical memory until they stop being referenced frequently. The virtual memory handler concludes the latter when the page reaches the head of the queue and its reference bit is 0. It is convenient to implement this approximation using a circular queue. A pointer called *head* points to the current head of the queue. If the page located at the head of the queue is considered to be frequently accessed, its reference bit is reset and *head* is simply advanced to point to the next page in the queue; otherwise, the page at the head of the queue is removed from memory and from the queue. The new page is added at the current position of the pointer and the pointer is advanced. This algorithm is sometimes called a *clock algorithm* for obvious reasons. (Refinements of the clock algorithm are used in the Unix virtual memory handler—see Section 6.7.)

Example 6.8 Figure 6.20 illustrates operation of the FIFO and second-chance algorithms when *alloc* = 3 for the page reference string

page reference string : 5, 4, 3, 2, 1, 4, 3, 5, 4, 3, 4, 1, 4, ... (6.7)

reference time string : $t_1, t_2, t_3, t_4, t_5, t_6, t_7, t_8, t_9, t_{10}, t_{11}, t_{12}, t_{13} \dots$ (6.8)

A circular queue of pages is maintained to implement the second-chance policy. The symbol ' \leftarrow ' marks the head of the queue. A ' \cdot ' after a page number in the queue implies that the reference bit of the page is 1. Execution of the process is initiated

after loading Page 5 in memory. All page references in the time interval $t_2 - t_8$ give rise to page faults. Up to time instant t_3 empty page frames exist in the memory allocated to the process, hence new pages are loaded straightaway. After t_3 a page fault leads to replacement of the page at the head of the queue. When a page is loaded its reference bit is initialized to 0.

At t_9 , reference to Page 4 does not lead to a page fault since it is already in memory. Its reference bit is set to 1. The next two references, which are to Pages 3 and 4, also have a similar effect. The queue pointer is not advanced since page replacement is not performed. At t_{12} , the reference to Page 1 leads to a page fault. Pages 4 and 3, which appear at the head of the queue, are not replaced since they have their reference bits 1. Instead their reference bits are set to 0 and the queue pointer is advanced. The pointer now points at Page 5, which has its reference bit 0. Hence Page 5 is replaced by Page 1. Page 4 is in memory when it is referenced at t_{13} . Thus one less page fault is suffered than in the case of FIFO page replacement.

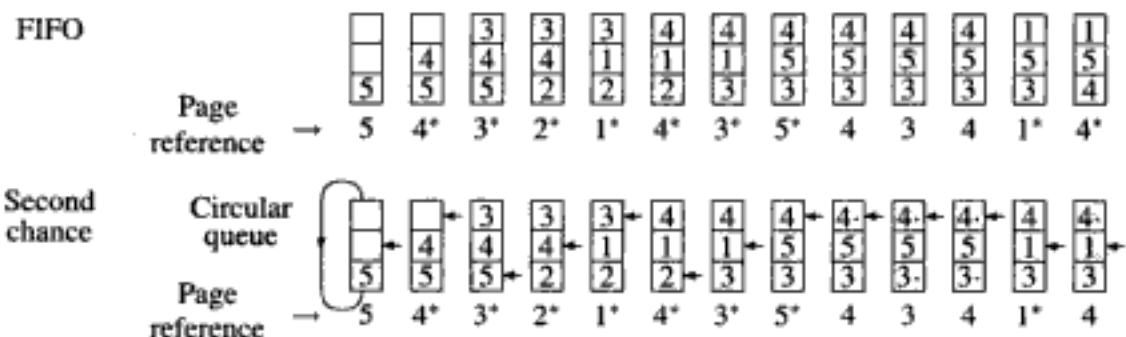


Fig. 6.20 Performance of FIFO and second-chance page replacement

Variations of second chance algorithm The notion of 'second chance' can be used in any setting where a virtual memory handler maintains a list of candidates for replacement. A candidate for replacement is a page that can be dispensed with because it has not been referenced for some time. Its entry in the page table of its process is changed to 'not present in memory'. However, it is not removed from memory—it exists in memory until it is overwritten by a new page. When a page fault occurs, the virtual memory handler checks whether the required page exists in the list of candidates for replacement. If so, the page is simply 'reconnected' to the process and removed from the list of candidates. This action avoids a page-in operation. If the required page is not present in the list of candidates, the virtual memory handler loads it in the page frame occupied by the page that is at the head of the list of candidates.

Replacement of a modified page requires a page-out operation to precede a page-in operation to load a new page. It incurs a higher overhead of page replacement. It also causes a delay in the loading of a new page. These effects can be countered using the following arrangement:

- The virtual memory handler always favors a candidate page that has not been

modified. It chooses a modified candidate page only if an unmodified candidate page does not exist.

- While performing replacement of unmodified pages, the virtual memory handler also initiates page-out operations on modified candidate pages. When the page-out operation on a modified page is completed, it is marked ‘not modified’. It would be considered for replacement at the next page fault.

Due to the first provision, a page-out needs to be performed during a page replacement operation only if the virtual memory handler has run out of unmodified candidate pages. The second provision tries to convert a modified candidate page into an unmodified candidate page before it is chosen for replacement. Thus the two provisions together reduce delays caused by page-out operations that have to be performed during page replacement. In the best case, these delays would be eliminated completely.

6.4 MEMORY ALLOCATION TO A PROCESS

Section 6.2 described how an over-commitment or under-commitment of memory to processes can adversely affect a system’s performance. An over-commitment of memory to processes leads to good process performance. However, the degree of multiprogramming in the system would be low, so the CPU–I/O overlap would be poor and system performance would suffer. An under-commitment of memory to processes leads to poor performance of each process, resulting in high page fault rates and heavy page traffic in the system. This is thrashing (see Def. 6.2). In this situation we notice an increase in the paging and scheduling overheads, high page I/O and low CPU efficiency.

The desirable operating zone for a process, shown in Figure 6.7, avoids over-commitment or under-commitment of memory to a process. Keeping each process in the desirable operating zone would safeguard both good process performance and good system performance. However, it is not clear how the virtual memory handler would decide the correct number of page frames to be allocated to each process, that is, the correct value of *alloc* for each process.

Thus the key issues in memory allocation in a virtual memory system are whether to vary the allocation to a process and, if so, how to vary it. We discuss three representative approaches to determining the memory allocation for processes:

- Fixed allocation, local replacement
- Variable allocation, global replacement
- Variable allocation, local replacement

In local replacement a page fault is serviced by replacing a page of the same process, whereas in global replacement the page which is to be replaced may belong to any process in the system. When replacement is local, performance of a process

is independent of behavior of other processes in the system. This is not so when replacement is global.

In fixed allocation using local replacement, memory allocation decisions are performed statically. The memory to be allocated to a process is determined by some criteria when the process is initiated. In a simple-minded policy, the memory allocated to a process could be a fixed fraction of its size. Page replacement is always performed locally. The approach is easy to implement. The cost of page replacement is moderate as only pages of the executing process participate in a page replacement decision. However, the approach suffers from all problems connected with a static decision. A process may suffer from under-commitment or over-commitment of memory. This fact would affect its own performance and performance of the system. The system may encounter thrashing.

In variable allocation using global replacement, the page replacement policy uses the page reference information for all processes in the system to pick a candidate for replacement. Allocation for the currently executing process increases if a page belonging to another process is replaced; otherwise, it remains unchanged. This allocation policy faces two difficulties. The allocation for the executing process may grow too large. For example, if LRU replacement policy is used, the virtual memory handler would replace pages of other processes most of the time because their last references would precede references to pages of the executing process. This could lead to an over-commitment of memory to the executing process. A blocked process would lose the page frames allocated to it, with the result that it would face high page fault rates when it is scheduled again. However, the page fault rate would decrease as it executes.

The variable allocation, local replacement approach varies the allocation of a process to avoid both under-commitment and over-commitment of memory. This approach avoids thrashing. It also avoids a situation in which a blocked process would lose its page frames, so a process would not face high page fault rates every time it is scheduled. However, this approach requires the virtual memory handler to determine the correct value of *alloc* for a process at any time. In the following we describe an approach to determining a practical value of *alloc*.

Working set model The notion of a *working set* helps the virtual memory handler to decide how many and which pages of a process should be in memory to obtain good execution performance of the process. A virtual memory handler using the working set model is said to use a *working set memory allocator*.

Definition 6.4 (Working set) *The working set of a process is the set of process pages that have been referenced in the previous Δ instructions of the process, where Δ is a parameter of the system.*

The previous Δ instructions are said to constitute the *working set window*. We introduce the following notation for our discussion:

$WS_i(t, \Delta)$: Working set for process $proc_i$ at time t for window size Δ

$WSS_i(t, \Delta)$: Size of the working set for process $proc_i$, i.e., the number of pages in $WS_i(t, \Delta)$.

Note that $WSS_i(t, \Delta) \leq \Delta$ because a page may be referenced more than once in a working set window. We omit (t, Δ) when t and Δ are either unimportant or obvious from the context.

A working set memory allocator keeps the working set of a process in memory at all times. Thus, at every time instant t , pages of $proc_i$ that are included in $WS_i(t, \Delta)$ are in memory, and $alloc_i = WSS_i(t, \Delta)$. If this is not possible, the allocator sets $alloc_i = 0$ and suspends $proc_i$. There are two reasons why this approach can be expected to provide good performance of a process. First, from the principle of locality of reference, there is a high probability that pages referenced by a process would be already in memory. Second, if the value of Δ is carefully chosen, there is no danger of undercommitment of memory to a process. This feature also avoids thrashing in the system.

The working set size of a process changes dynamically, so memory allocation for processes also changes dynamically. Hence the working set memory allocator must vary the number of processes in memory. For example, the degree of multiprogramming should be decreased if

$$\sum_k WSS_k(t, \Delta) > s_m$$

where s_m is the total number of page frames in memory and $\{proc_k\}$ is the set of processes in memory. The working set memory allocator removes some processes from memory until $\sum_k WSS_k(t, \Delta) \leq s_m$.

The degree of multiprogramming should be increased if $\sum_k WSS_k(t, \Delta) < s_m$ and there exists a process $proc_g$ such that

$$WSS_g(t, \Delta) \leq (s_m - \sum_k WSS_k(t, \Delta)).$$

$proc_g$ should now be allocated $WSS_g(t, \Delta)$ page frames and its execution should be started or resumed.

Variations in the degree of multiprogramming are implemented as follows: The virtual memory handler maintains two items of information for each process— $alloc_i$ and WSS_i . When the degree of multiprogramming is to be reduced, the virtual memory handler decides which process is to be suspended. Let this be $proc_i$. The virtual memory handler now performs a page-out operation for each modified page of $proc_i$ and changes the status of all page frames allocated to it to *free*. $alloc_i$ is now set to 0, however the value of WSS_i is left unchanged. When the degree of multiprogramming is to be increased and the virtual memory handler decides to resume execution of $proc_i$, it sets $alloc_i = WSS_i$ and allocates the number of page frames indicated by WSS_i . It now loads one page of $proc_i$. This is the page that contains the next instruction to be executed. Other pages are loaded when page faults occur. An alternative

would have been to load all pages of WS_i while resuming execution of $proc_i$. However, this approach may lead to redundant loading of pages because some pages in WS_i may not be referenced again.

Performance of a working set memory allocator is sensitive to the value of Δ . If Δ is too large, memory would contain some pages that are not likely to be referenced. This would result in overcommitment of memory to processes. It would also force the virtual memory handler to reduce the degree of multiprogramming, thereby affecting system performance. If Δ is too small, there is a danger of undercommitment of memory to processes, leading to a rise in page fault frequency and the possibility of thrashing.

Implementation of a working set memory allocator Use of a working set memory allocator suffers from one practical difficulty. It is expensive to determine $WS_i(t, \Delta)$ and $alloc_i$ at every time instant t . To address this difficulty, a virtual memory handler using a working set memory allocator can determine the working sets of all processes periodically rather than at every time instant. Working sets determined at the end of an interval are used to decide values of $alloc$ for use during the next interval. Example 6.9 illustrates this approach.

Example 6.9 A virtual memory handler has 60 page frames available for allocation to user processes. The working sets of all processes are recomputed at time instants $t_{j \times 100}^+, j = 1, 2, \dots$. Following the computation of working sets, the virtual memory handler handles each process $proc_i$ as follows: It sets $alloc_i = WSS_i$ if it can allocate WSS_i page frames to it, otherwise it sets $alloc_i = 0$ and removes all pages of $proc_i$ from the memory. For each process, the value of $alloc$ assigned at $t_{j \times 100}^+$ is held constant until $t_{(j+1) \times 100}^+$. The process of computing working sets and determining $alloc_i$ for all processes is repeated at $t_{(j+1) \times 100}^+$.

Figure 6.21 illustrates operation of the system. It shows values of $alloc_i$ and WSS_i for all processes $proc_i$ at time instants $t_{100}^+, t_{200}^+, t_{300}^+$ and t_{400}^+ . At t_{100}^+ , $\sum_{i=1 \dots 3} WSS_i = 52$, $WSS_4 = 10$ and $alloc_4 = 0$. This implies that 8 page frames are free, however execution of $proc_4$ has been suspended because its working set size is 10 page frames. At t_{200}^+ , values of WSS_i , $i = 1, \dots, 3$ are recomputed. The value of WSS_4 is carried over from t_{100}^+ since $proc_4$ has not been executed in the interval $t_{100} - t_{200}$. $alloc_i$, $i = 1, \dots, 3$ are now assigned new values. $proc_4$ still cannot be swapped in for lack of memory since five page frames are free and $WSS_4 = 10$. At t_{300}^+ , $proc_4$ is swapped in, however it is swapped out again at t_{400}^+ . Note that the smallest allocation for $proc_2$ is 11 page frames and the largest allocation is 25 page frames during the interval $t_{100} - t_{400}$. This variation is performed to adjust its memory allocation to its recent behavior.

Expansion and contraction of $alloc$ is performed as follows: At t_{200}^+ , the virtual memory handler decides to reduce $alloc_1$ from 14 page frames to 12 page frames. It would use an LRU-like policy to remove two pages of $proc_1$. At t_{300}^+ , it increases $alloc_1$ to 14 page frames. It would allocate two more page frames to $alloc_1$. These page frames would be used when page faults arise during the execution of the process.

The virtual memory handler can use the reference bits provided by the paging hardware to determine the working sets. Reference bits of all pages in memory

	t_{100}		t_{200}		t_{300}		t_{400}	
	WSS	alloc	WSS	alloc	WSS	alloc	WSS	alloc
<i>proc₁</i>	14	14	12	12	14	14	13	13
<i>proc₂</i>	20	20	24	24	11	11	25	25
<i>proc₃</i>	18	18	19	19	20	20	18	18
<i>proc₄</i>	10	0	10	0	10	10	12	0

Fig. 6.21 Operation of a working set memory allocator

can be turned off when working sets are determined. These bits will be turned on again as these pages get referenced during the next interval. While performing page replacement, the virtual memory handler can keep track of replaced pages whose reference bits were on. The working set at the end of the next interval will consist of these pages and all pages in memory whose reference bits are on.

Implementation of working sets in this manner faces one problem. Resetting of reference bits at the end of an interval would interfere with page replacement decisions. If a page fault occurs in a process soon after working sets were determined, most pages of the process in memory would have their reference bits off, so the virtual memory handler cannot differentiate between these pages for the purpose of page replacement. If some processes either remain blocked or do not get an opportunity to execute all through an interval, their allocations would shrink unnecessarily. This effect makes it difficult to decide on the correct size of the working set window.

An alternative is to use a working set window for each process individually; however, that would complicate the virtual memory handler and add to its overhead. It would also not address the issue of interference with the page replacement decisions. Due to these reasons few OSs actually use working set memory allocators. Nevertheless the notion of working sets is important in understanding the page fault behavior of processes in a virtual memory system.

6.5 SHARED PAGES

Sharing of programs was discussed in Section 5.11.3.2. *Static sharing* results from static binding performed by a linker or loader before execution of a program begins (see Section 5.11.3.2). With static binding, if two processes *A* and *B* statically share program *C*, then *C* is included in the code of both *A* and *B*. Let the 0th page of *C* become page *i* of process *A* (see Figure 6.22(a)). If the Add instruction in page 1 of program *C* has an operand in page 4 of *C*, the instruction would be relocated to use an address located in the *i + 4th* page of process *A*. If the 0th page of *C* becomes page *j* in process *B*, the Add instruction would be relocated to use an address located in *j + 4th* page of process *B*. Thus, each page of program *C* has two copies in the address spaces of *A* and *B*. These copies may exist in memory at the same time if

processes *A* and *B* are in operation simultaneously.

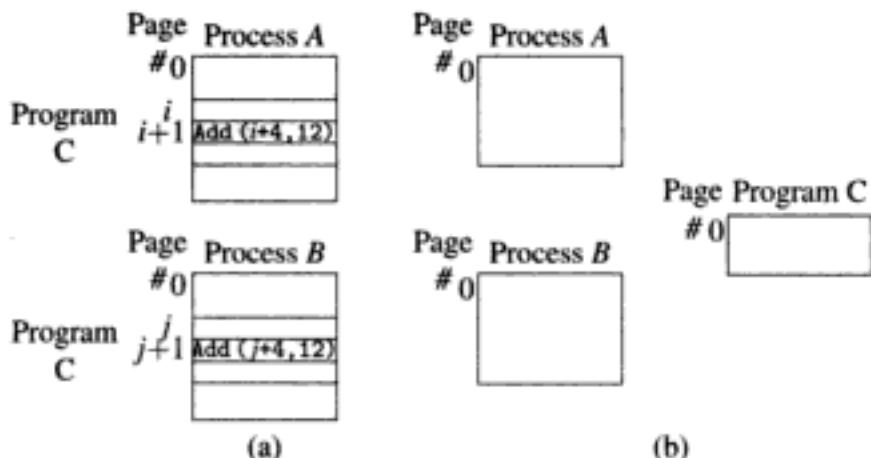


Fig. 6.22 Sharing of program C by processes *A* and *B*: (a) Static, and (b) Dynamic binding

Dynamic binding (see Section 5.2) can be used to conserve memory by binding the same copy of a program or data to several processes. In this case, the program or data to be shared would retain its identity (see Figure 6.22(b)), and it would be dynamically bound to several programs by a dynamic linker invoked by the virtual memory handler. This is achieved as follows: Process *A* makes a system call to bind program *C* (or some data) to a specific logical address in its logical address space. Let this address be aligned on the start of page *i*. The kernel now invokes the virtual memory handler, which creates entries in the page table of *A* for pages of program *C*, and sets a flag in each page table entry to indicate that it is a shared page. It now invokes the dynamic linker, which changes the Add instruction of program *C* to Add (*i*+4, 12). When a reference to an address in program *C* page faults, the virtual memory handler checks whether the required page of *C* has been already loaded in memory. If so, it puts the page frame number of the page in the relevant entry of *A*'s page table. Similar actions are performed when process *B* binds program *C* to the start address of page *i*. Figure 6.23 shows the arrangement set up by these bindings.

Two conditions should be satisfied for dynamic binding of programs to work. The program to be shared should be coded as a reentrant program so that it can be invoked by many processes at the same time (see Section 5.11.3.2). The program should also be bound to identical logical addresses in every process that shared it. It would ensure that an instruction like Add (*i*+4, 12) in page *i*+1 of Figure 6.23 will function correctly in each of the processes. These conditions are unnecessary when data is dynamically bound to several processes; however, sharing processes would have to synchronize their accesses to the shared data to prevent race conditions.

When sharing of pages is implemented by making the page table entries of sharing processes point at the same page frame, page reference information for shared pages will be dispersed across many page tables. The page replacement algorithm will have to gather this information together to get the correct picture about refer-

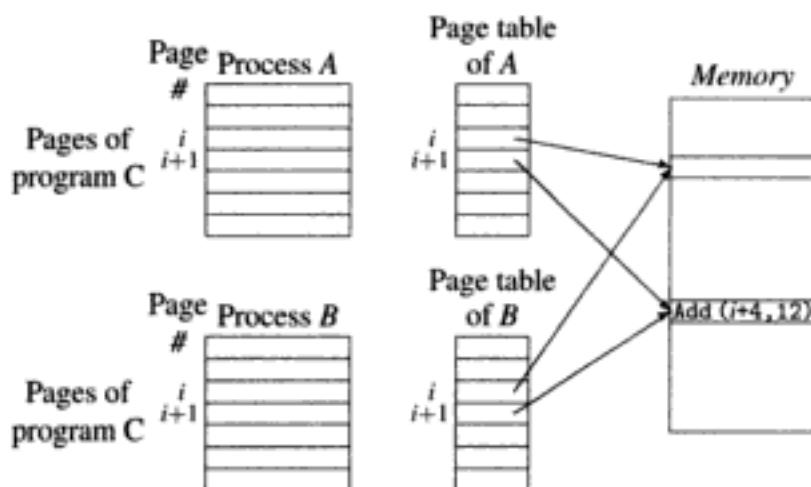


Fig. 6.23 Dynamic sharing of program C by processes A and B

ences to shared pages. This is rather cumbersome. A better method would be to maintain information concerning shared pages in a separate *shared pages table* and collect page reference information for shared pages in page entries in this table. This arrangement also permits a different page replacement criterion to be used for managing shared pages. In Section 6.9, we describe a related technique used in Windows operating systems.

6.5.1 Copy-on-Write

The copy-on-write feature is used to conserve memory when data in shared pages could be modified but the modified values are to be private to a process. When processes A and B dynamically bind such data, the virtual memory handler sets up the arrangement shown in Figure 6.24(a), which is analogous to the arrangement illustrated in Figure 6.23 except that the page table entries of the data pages contain bit flags 'c' to indicate that the copy-on-write feature is to be employed when any of these pages is modified. If process A now tries to modify page k , the MMU raises a page fault on seeing that page k is a copy-on-write page. The virtual memory handler now makes a private copy of page k for process A, accordingly changes the page frame number stored in page k 's entry in the page table of A, and also turns off the copy-on-write bit in this entry (Figure 6.24(b)). Other processes sharing page k would continue to use the original copy of page k in memory; each of them too would get a private copy of the page if they modified it.

In Unix systems, a child process starts off with the code and data of the parent process; however, it can modify the data and the modified values are private to it. Use of the copy-on-write feature for the entire address spaces of the parent and child processes avoids copying of code pages because they are not modified; only data pages would be copied if they are modified.

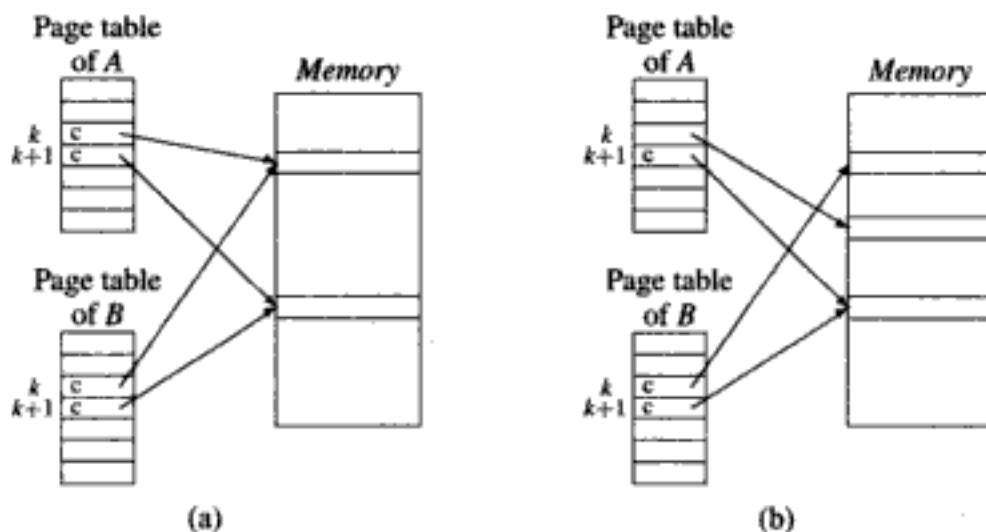


Fig. 6.24 Implementing copy-on-write: (a) before, and (b) after process A modifies page k

6.6 MEMORY MAPPED FILES

Memory mapping of a file by a process binds that file to a part of the logical address space of the process. This binding is performed when the process makes a *memory map* system call; it is analogous to dynamic binding of programs and data discussed earlier in Section 6.5. After memory mapping a file, the process refers to data in the file as if it were data located in pages of its own address space, and the virtual memory handler coordinates with the file system to load page-sized parts of the file into memory on demand. When the process updates the data contained in such pages, the *modified* bits of the pages are set *on* but the data is not immediately written out into the file; dirty pages of data are written out to the file when the page frames containing them are to be freed. When the process makes a *memory unmap* call, the virtual memory handler deletes the file from the logical address space of the process. Any dirty pages that still contain the file's data are written out at this time.

Figure 6.25 shows the arrangement used for memory mapping of file *info* by process A. Note that the page-in and page-out operations on those pages of process A that do not belong to file *info* involve the swap space of the process and are performed by the virtual memory handler. Reading and writing of data from file *info* are performed by the file system in conjunction with the virtual memory handler. If several processes memory map the same file, we have an arrangement analogous to that shown in Figure 6.23; these processes would effectively share the memory mapped file.

Table 6.5 summarizes the advantages of memory mapping of files. Memory mapping makes file records accessible using the virtual memory hardware. This is inherently more efficient. A process may use memory mapping to reduce the number of memory-to-memory copy operations as follows: When a process accesses some data in a non-memory mapped input file, the file system first copies the record into a

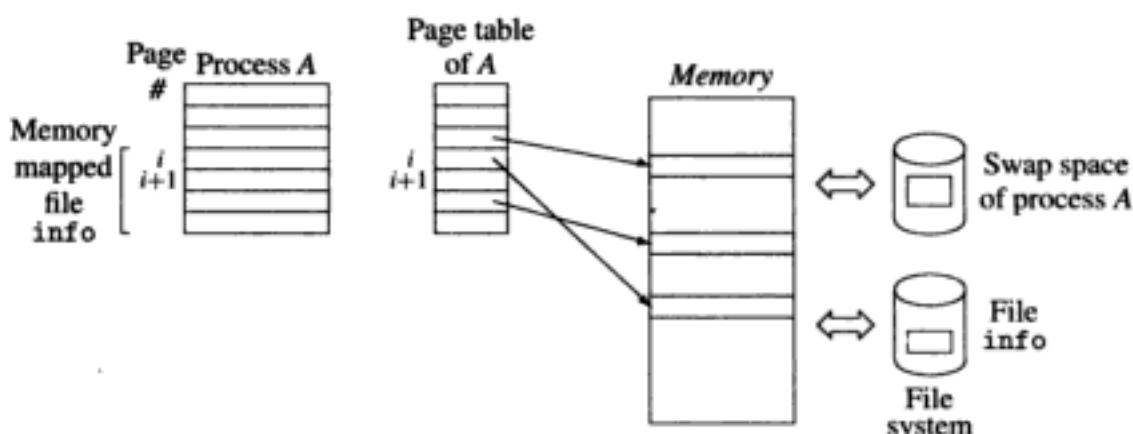


Fig. 6.25 Memory mapping of file `info` by process *A*

Table 6.5 Advantages of memory mapped files

Advantage	Description
File data as pages	Access to file data is looked upon as access to pages, which is inherently more efficient due to virtual memory hardware.
Less memory-to-memory copying	File data is a part of the process address space. The process can exploit this feature by accessing the data <i>in situ</i> , i.e., where it exists in the logical address space, rather than by copying it into another part of its address space as done during normal file processing.
Fewer read/write operations	File data is read in or written out one page at a time, rather than at every file operation, and so a single read/write operation may suffice for several file operations.
Prefetching of data	For sequential reads, data will already be in memory if the page that contains the data was read in during a previous file operation.
Efficient data access	File data can be accessed efficiently irrespective of file organization.

memory area used as a file buffer or disk cache (see Chapter 12). The process now copies the data from the buffer or the disk cache into its own address space (i.e., into some variables) for accessing it. Thus one disk-to-memory copy operation and one memory-to-memory copy operation are performed. When a file is memory mapped, the memory-to-memory copy operation from the buffer to the process address space is not necessary since the data is already a part of the process address space. Similarly, fewer copy operations are performed when file data is modified. Data located in

a page that was read in during a previous file operation can be accessed without disk I/O; so memory mapping reduces the number of I/O operations performed during file processing.

The last advantage, efficient access to data in a file irrespective of its organization, arises from the fact that data in a file is accessed using the virtual memory hardware. Hence any part of the data can be accessed equally efficiently. If some data in the middle of a large sequential file is to be accessed, only the page containing the data is loaded and accessed.

6.7 UNIX VIRTUAL MEMORY

Unix has been ported on computer systems with diverse hardware designs, hence a variety of ingenious schemes have been used to exploit features of paging hardware of a host machine. This Section describes some common features of the Unix virtual memory and interesting techniques used in different Unix versions. Its purpose is to provide a view of the practical issues in virtual memory implementations rather than to study the virtual memory handler of any specific Unix version in detail. Whenever possible, Unix terminology has been replaced by terminology used in previous Sections of this Chapter.

Logical address space and swap space The page table of a process differentiates between three kinds of pages—resident, un-accessed and swapped-out pages. A resident page exists in memory; it must have been loaded on demand at some page fault. An un-accessed page has not been accessed even once during execution of the process, hence it has not never been loaded in memory. It would be loaded when its use in some instruction leads to a page fault. As described later, the page fault handling actions for such a page depend on whether it is a text page or a data page. A swapped-out page exists in the swap space. A reference to it would page fault, and it would be loaded back in memory from its location in the swap space.

An un-accessed page may be a text page or a data page. A text page is loaded from an executable file existing in the file system. Locating such a page in the file system may require reading of several disk blocks in the inode and the file allocation table. To avoid this, the virtual memory handler maintains information about such pages in a separate table and uses it when a page needs to be loaded. As described later, 4.3BSD virtual memory handler maintains this information in the page table entry itself. This information gets overwritten by the page frame number when the page is loaded, hence it is not available if the page gets removed from memory and has to be reloaded. A text page may be removed from memory if it is not used for some time and may be reloaded at a subsequent page fault, hence Unix 4.3BSD writes out a text page into the swap space when it is removed from memory for the first time, and loads it from the swap space on demand. A data page is called a *zero-fill* page; it is filled with zeroes when its first use leads to a page fault. Thereafter, it is either a resident page or a swapped-out page.

A text page may exist in memory even if it is marked non-resident in its page table entry. This situation arises if some other process is using the page (or has used it in the past). An obvious optimization is possible here when a page fault occurs for a text page, the virtual memory handler first checks if the page already exists in memory. If so, it simply puts the page frame information the page table entry of the page and marks the page as resident. This action avoids a page-in operation and also conserves memory.

To conserve disk space, an effort is made to allocate as little swap space as possible. To start with, sufficient swap space is allocated to accommodate the user stack and the data area. Thereafter swap space is allocated in large chunks whenever the stack or data area grows in size. This approach suffers from the problem that swap space in the system may get exhausted when the data area of a process grows. Such a process has to be suspended or canceled during its execution. However, this situation cannot arise at an arbitrary point in the execution of a process; it can arise only when the address space of a process grows.

Copy-on-write The Unix virtual memory handler uses the copy-on-write principle to conserve memory and swap space when a process creates a child process through the *fork* call. Semantics of *fork* require that the child should obtain a copy of the parent's address space. These semantics can be implemented by allocating distinct memory areas and swap space for the child process. However, child processes frequently discard the copy of their parent's address space by using the *exec* call to load some other program for execution. In any case, a child process may not wish to modify much of the parent's data, so both memory and swap space can be optimized by selectively replicating parts of the data space rather than the entire data space. The copy-on-write principle forces a distinct copy of a data page to be created for use by the child process only when the child process tries to modify some part of the page.

Copy-on-write is implemented as follows: When a process is forked, the reference count of all data pages in the parent's address space is incremented by 1. Bits in the access privileges field of the page table entry of a data page are set to make the page read-only. Any attempt at modifying the page raises a protection fault. The virtual memory handler realizes this situation to be an attempt to modify a data page shared by a process and its child. It now reduces the reference count of the page, makes a copy of this page and allocates the read and write privileges to this copy by setting appropriate bits in its page table entry. Thus copies are made only of modified data pages.

Efficient use of page table and paging hardware We live in an imperfect world and try to make the most of it. Nowhere is it more apparent than in the case of virtual memory software. Most computer systems provide inadequate support for good virtual memory management, and it is left to the virtual memory handler to make virtual memory practical and profitable. We discuss some interesting techniques that are used in Unix virtual memory implementations to overcome deficiencies in hardware

support for page replacement.

If a page does not exist in memory, the valid bit of its page table entry is 'off'. Bits in other fields of such an entry, like the *reference* field or the *page frame* field, do not contain any useful information. Hence these bits can be used for some other purposes. Unix 4.3BSD uses these bits to contain address of the disk block in the file system that contains a text page. VAX 11 does not provide a reference bit to collect page reference information. Its absence is compensated by using the valid bit in an interesting manner. Periodically, the valid bit of a page is turned off even if the page exists in memory. The next reference to the page causes a page fault. However, the virtual memory handler knows that this is not a genuine page fault, hence it sets the *valid* bit and returns to execution of the process. In effect, the *valid* bit is used as the reference bit.

In computer systems that do not provide sufficient information concerning last use of a page, the reference bit is manipulated to achieve an equivalent effect. This approach, called the not recently used approach, is as follows: The virtual memory handler turns off the reference bits of pages. Sometime afterwards, the reference bit of the page is examined again. It would be found to be 1 only if the page was referenced after its reference bit was turned off—that is, only if the page was referenced recently. A page whose reference bit is still off has not been referenced since the bit was turned off. Such a page is a candidate for page replacement.

Variations of this theme have been used in different versions of Unix. As described later, the Unix virtual memory handler maintains a free list of pages and tries to keep a certain number of inactive pages always on the free list. When it needs to load a new page, it overwrites the first page in the free list with the new page. When it needs to add more pages to the free list, it uses the reference bit to decide whether a page is inactive. The virtual memory handler scans all pages in memory using one or two pointers. These algorithms are called *clock algorithms* because of the way they use pointers. The page pointed to by a pointer is examined to perform an appropriate action and the pointer is advanced to point at the next page. After examining the last page, the pointer is set to point to the first page once again. In this manner, the pointer moves over the pages over and over the same way the hour or minute hand of a clock moves over all positions marked on a clock dial.

In the *one-handed clock algorithm*, a scan consists of two passes. In the first pass, the virtual memory handler simply resets the reference bit of the page pointed to by the pointer. In the second pass it adds pages whose reference bits are still off to the free list. In the *two-handed clock algorithm*, two pointers are maintained. One pointer is used for resetting the reference bits and the other pointer is used for checking the reference bits. Both pointers are incremented simultaneously. The page to which the checking pointer points is added to the free list if its reference bit is off.

Pageout daemon To facilitate fast page-in operations, Unix virtual memory han-

dlers maintain a list of free page frames and try to keep at least 5 percent of total page frames on this list at all times. A daemon called the *pageout daemon* (process 2) is created for this purpose. It is activated any time the total number of free page frames falls below 5 percent. It tries to add pages to the free list and puts itself to sleep when the free list contains more than 5 percent free page frames. Some versions of Unix use two thresholds—a high threshold and a low threshold—instead of a single threshold at 5 percent. The daemon goes to sleep when it finds that the number of pages in the free list exceed the high threshold. It is activated when this number falls below the low threshold. This arrangement avoids frequent activation and deactivation of the daemon.

Pages in memory can be divided into two broad categories—those that are fixed in memory and those that are not. A process requests fixing of some pages in memory to reduce page fault rates and improve its execution performance. The system permits a process to fix only a certain fraction of its pages in this manner. These pages cannot be removed from memory until they are unfixed by the process. Interestingly, there is no I/O fixing of pages in Unix since I/O operations take place between a disk block and a block in the buffer cache rather than between a disk block and the address space of a process.

The virtual memory handler divides pages that are not fixed in memory into active pages, i.e., pages that are actively in use by a process, and inactive pages, i.e., pages that have not been referenced in the recent past. The virtual memory handler maintains two lists, the active list and the inactive list. Both lists are operated as queues. A page is added to the active list when it becomes active, and to the inactive list when it is deemed to have become inactive. Thus the least recently activated page is at the head of the active list and the oldest inactive page is at the head of the inactive list. A page is moved from the inactive list to the active list when it is referenced. The pageout daemon tries to maintain a certain number of pages, computed as a fraction of total resident pages, in the inactive list. If it reaches the end of the inactive list while adding page frames to the free list, it checks whether the total number of pages in the inactive list is smaller than the expected number. If so, it transfers a sufficient number of pages from the active list to the inactive list.

The pageout daemon is activated when the number of free page frames falls below the low threshold while handling a page fault. The daemon frees page frames in the following order: page frames containing pages of inactive processes, page frames containing inactive pages of active processes and page frames containing active pages of active processes. The daemon finds inactive processes, if any, and swaps them out. It goes back to sleep if the number of free page frames now exceeds the high threshold.

If the number of free page frames after swapping out inactive processes is still below the low threshold, the pageout daemon scans the inactive list and decides whether and when to add page frames occupied by inactive pages to the free list. A page frame containing an inactive page is added to the free list straightaway if the

page is unreferenced and not dirty. If the page is dirty and not already being swapped out, the pageout daemon starts a page-out operation on the page and proceeds to examine the next inactive page. If a page is being swapped out, the daemon merely skips it. The *modify* bit is reset when its page-out operation is completed. Page frames containing such pages would be added to the free list sometime in future when the daemon examines them in a subsequent pass and finds that their page-out operation is complete. The daemon activates the swapper if it cannot add a sufficient number of page frames to the free list. The swapper swaps out one or more active processes to free sufficient number of page frames.

To optimize page traffic, the virtual memory handler writes out dirty pages to the swap space in clusters. When the page daemon finds a dirty page during its scan, it examines adjacent pages to check whether they are dirty. If so, a cluster of dirty pages is written out to the disk in a single I/O operation. Another optimization concerns redundant page-in operations. When a page frame f_i occupied by some clean page p_i is added to the free list, the valid bit of p_i 's page table entry is set to 0. However, the page is not immediately overwritten by loading another page in the page frame. This would happen sometime in future when its entry comes to the head of the free list and it gets allocated to some process. The next reference to p_i would page fault since the valid bit in its page table entry has been set to 0. If it still exists in f_i , i.e., if f_i is still in the free list, then it can be simply taken out of the free list and 'reconnected' to the logical address space of the process. This saves a page-in operation and consequent delays to the page-faulting process.

Swapping Unix virtual memory handler does not use a working set memory allocator because of high overhead of such an allocator. Instead it focuses on maintaining needed pages in memory. A process is swapped out if all its required pages cannot be maintained in memory and conditions resembling thrashing exist in the system. An inactive process, i.e., a process that is blocked for a long time, may also be swapped out in order to maintain a sufficient number of free page frames. When this situation arises and a swap-out becomes necessary, the pageout daemon activates the swapper, which is always process 0 in the system. The swapper finds and swaps out inactive processes. If that does not free sufficient memory, then it is activated again by the pageout daemon. This time it swaps out the process that has been resident the longest amount of time. When swapped out processes exist in the system, the swapper periodically checks whether sufficient free memory exists to swap-in some processes. A swap-in priority—which is a function of when the process was swapped out, when it was last active, its size and its nice value—is used for this purpose. This function ensures that no process remains swapped out indefinitely. In 4.3BSD, a process was swapped-in only if it could be allocated as much memory as it held when it was swapped out. In 4.4BSD this requirement has been relaxed; a process is brought in if enough memory to accommodate its user structure and kernel stack can be allocated to it.

6.8 LINUX VIRTUAL MEMORY

Linux uses a page size of 4 Kbytes. On 64-bit architectures, it uses a three-level page table (see Section 6.2.6). The three levels are the page global directory, the page middle directory and the page table. Accordingly, a logical address consists of four parts, three of these are for the three levels and the fourth one is the byte number within a page.

Linux uses an interesting arrangement to eliminate page-in operations for pages that were loaded previously in memory, but were marked for removal. This is achieved by using the following states for page frames: A *free* page frame is one that has not been allocated to a process, while an *active* page frame is one that is in use by a process to which it has been allocated. An *inactive dirty* page frame was modified by the process to which it was allocated but it is not in use by the process any more. An *inactive laundered* page is one what was *inactive dirty* and is therefore being written out to the disk. An *inactive laundered* page becomes *inactive clean* when its contents are copied to the disk. If a process page faults for a page that is in a page frame marked *inactive clean*, the page frame is once again allocated to the process, and the page is simply marked as present in memory. If the page is in a page frame marked *inactive laundered*, these actions are performed when its disk operation completes. Apart from saving on disk operations, this arrangement also prevents access to a stale copy of a page. An *inactive clean* page can also be allocated to another process straightaway.

Page replacement in Linux is based on a clock algorithm. The kernel tries to maintain a sufficient number of free page frames at all times so that page faults can be quickly serviced using one of the free page frames. It uses two lists called *active list* and *inactive list*, and maintains the size of the active list to two thirds of the size of the inactive list. When the number of free page frames falls below a lower threshold, it executes a loop until a few page frames are freed. In this loop it examines the page frame at the end of the inactive list. If its referenced bit is set, it resets the bit and moves the page frame to the head of the list; otherwise, it frees the page frame. When the balance between the active and inactive lists is to be maintained, it processes a few page frames from the end of the active list in a similar manner and either moves them to the head of the active list, or moves them to the head of the inactive list with their reference bits on. A page frame is moved from the inactive list to the active list if it is referenced by a process.

Linux uses a buddy system allocator for allocating page frames to processes (see Section 5.4.1). This method facilitates performing of I/O operations through older DMA buses that use physical addresses, because such I/O operations require memory to be contiguously allocated (see Section 6.2.4).

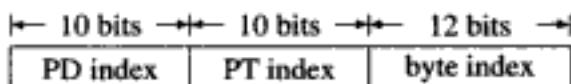
The logical address space of a process can consist of several virtual memory regions; each region can have different characteristics and is handled using separate policies for loading and replacement of pages. A page in a *zero filled memory region* is filled with zeroes at its first use. A *file backed region* facilitates memory mapping

of files. The page table entries of its pages point at the disk buffers used by the file system. This way, any update in a page of such a region is immediately reflected in the file and is visible to concurrent users of the file. A *private memory region* is handled in a different manner. When a new process is forked, the child process is given a copy of the parent's page table. At this time, pages of a private memory region are given a copy-on-write status. When a process modifies such a page, a private copy of the page is made for it.

6.9 VIRTUAL MEMORY IN WINDOWS

Windows operates on several different architectures. Windows supports both 32 bit and 64 bit logical addresses. The address space of a process is either 2 Gbytes or 3 Gbytes. The remainder of the logical address space is reserved for OS use. The kernel is mapped into this part of every process's address space. The page size is 4 Kbytes.

On an X-86 architecture, Windows uses a two-level page table organization similar to the one shown in Figure 6.15. The higher level page table is called a *page directory* (PD). PD contains 1024 entries of 4 bytes each. Each entry in PD points to a *page table*(PT). Each page table contains 1024 page table entries of 4 bytes each. Each 32 bit logical address is split into 3 components as shown below:



To resolve a logical address, the *PD index* field is used to locate a page table. The *PT index* field is used to select the entry for a page. This entry points to a page frame. The *byte index* is concatenated with the address of the page frame to obtain the effective physical address. Each page table entry contains a 20 bit address of the page frame containing a page. The remaining 12 bits are used for the following purposes: five bits contain the protection field, four bits indicate the file that contains a copy of the page and three bits specify the state of the page. If the page is not in memory, the 20 bits specify the offset into the paging file. This address can be used to load the page in memory. If the page contains code, a copy of it exists in a code file. Hence the page need not be included in a paging file before it is loaded for the first time. In this case, one bit indicates the page protection and 28 bits point to a system data structure that indicates the position of the page in a file containing code. On other architectures, Windows uses three-level or four-level page tables and uses different page table entry formats.

A page frame can be in any one of eight states. Some of these states are as follows: *valid* : the page is in active use, *free* : the page is not in active use, *zeroed* : the page is cleaned out and available for immediate use, *standby* : the page has been removed from the working set of the process to which it was allocated, but it can be 'reconnected' to the process if it is referenced again; *modified* : the page is dirty

and yet to be written out, and *bad* : the page cannot be accessed due to a hardware problem.

Two key provisions are made to handle shared pages. A *section* object represents a section of memory that can be shared. Every process that shares the section object has its own *view* of the object; a view controls the part of the object that is visible to the process. A process maps a view of a section into its own address space by making a kernel call with parameters indicating the part of the section object that is to be mapped, i.e., an offset and the number of bytes to be mapped, and the logical address in the address space of the process where the object is to be mapped. When a view is accessed for the first time, the kernel allocates memory to that part of the section that is covered by the view, unless memory is already allocated to it. If the memory section has the attribute *based*, the shared memory has the same virtual address in the logical address space of each sharing process.

A copy-on-write feature is used for sharing the pages. It implies that all sharing processes share the same copy of a page until the page is modified. If a process modifies a page, then a private copy of the page is created for it. Copy-on-write is implemented by setting the protection field of a page to *read only*. A protection exception is raised when a process tries to modify it. The virtual memory manager now makes a private copy of the page for use by the process.

The second provision for sharing concerns a level of indirection to access page table entries for shared pages. In the absence of such indirection, the page would have entries in page tables of all sharing processes, and all these copies would have to be modified when a page is loaded or removed from memory. To avoid this, an indirection bit is set in the PT entries of the page in each sharing process' page table. The page table entry now points to a *prototype page entry*, which points to the actual page frame containing the page. Only the prototype page entry needs to be modified when a shared page is loaded or removed from memory.

6.10 VIRTUAL MEMORY USING SEGMENTATION

In a segmented virtual memory, a process is a collection of segments. Each segment is a significant logical entity in a process. In programming terms, a segment usually consists of a set of procedures or data and forms a module of a software system. Thus a segment groups together procedures and data that possess common attributes for sharing by other procedures. Segmentation is useful while constructing large software systems that share modules with other software systems.

A logical address in a segmented process is viewed as a pair (s_i, b_i) where s_i and b_i are the segment number and byte offset, respectively. During execution, the MMU converts this reference into an effective memory address using the schematic of Figure 6.26. Several similarities with the paged virtual memory can be noted. A special hardware register called the *segment table address register* (STAR) points to the *segment table* of a process. The segment table is accessed with s_i as an index. If the segment is present in memory, address translation is completed using the memory

start address found in its segment table entry; otherwise, a ‘missing segment’ fault is raised.

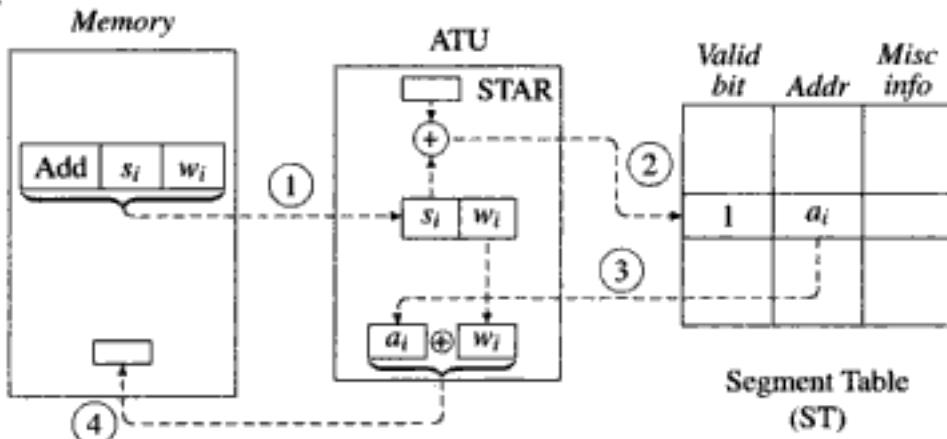


Fig. 6.26 Virtual memory implementation using segmentation

Address translation in a segmented virtual memory system differs from that in a paged virtual memory system in one significant respect. Unlike pages, segments have different lengths, hence physical memory cannot be allocated in units of a fixed size. Each segment table entry therefore contains a complete physical memory address rather than simply a frame number as in a paged virtual memory system. The byte number b_i is added to this address to compute the effective physical address. This step involves an addition cycle rather than mere concatenation as in paging.

Variations exist concerning the manner in which s_i and b_i are indicated in an instruction. One method is to use numeric identifiers for each of them. The logical address thus consists of a segment number and a byte offset, and address translation is performed as described earlier. Alternatively, s_i and b_i could be specified in a symbolic form, i.e., as names. A typical logical address in this case would be $(\text{alpha}, \text{beta})$ where alpha is the name of a segment and beta is the name of a byte in it. During address translation, the MMU has to search and locate the entry of alpha in the segment table to obtain its start address in memory. For each segment a table showing ids and their offsets in the segment would have to be maintained. We will call this the *segment linking table* (SLT), and use the segment name as a subscript. Thus $\text{SLT}_{\text{alpha}}$ is the segment linking table for alpha . $\text{SLT}_{\text{alpha}}$ is searched to find the offset of beta in segment alpha . This offset is added to the start address of alpha to complete the address translation.

Example 6.10 Figure 6.27 illustrates effective address calculation for the logical address $(\text{alpha}, \text{beta})$. Part (a) of the figure shows segment alpha . beta and gamma are the two ids that might be used in logical addresses. These ids are associated with bytes in the segment having offsets 232 and 478, respectively. The segment linking table $\text{SLT}_{\text{alpha}}$ contains entries for beta and gamma , showing their offsets. The segment table entry of alpha indicates that it exists in the memory area with the start address 23480. The offset of beta is 232, so the effective address of $(\text{alpha}, \text{beta})$

would be computed as $23480 + 232 = 23712$.

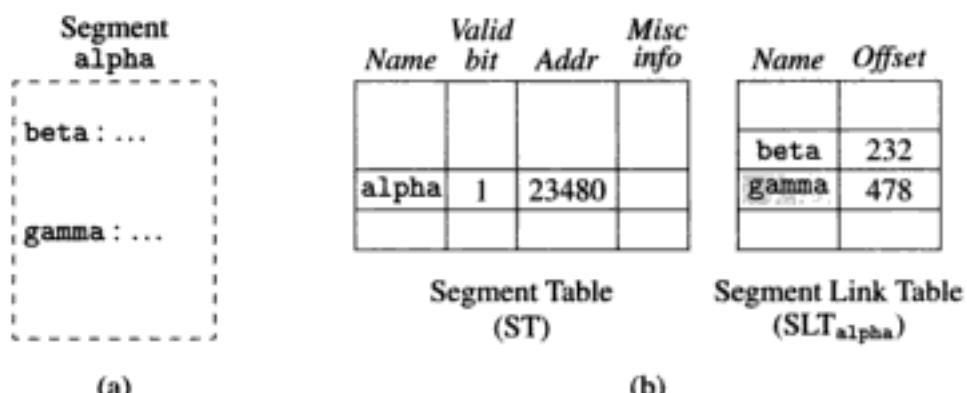


Fig. 6.27 Use of symbolic segment and word ids

Both numeric and symbolic id's have been used in practical segmented virtual memory systems. The MULTICS system used symbolic identifiers. For simplicity, in this Section we will assume the use of numeric identifiers. The logical address space in a segmented virtual memory system is inherently two-dimensional in nature. A logical address lil is specified as a pair (s_i, b_i) . The manner in which (s_i, b_i) is represented determines the maximum size of a segment. If n_1 bits are used to represent s_i and n_2 bits are used for b_i , a process can contain 2^{n_1} segments and the maximum size of a segment is 2^{n_2} bytes. Consider a segmented virtual memory system using numeric segment number and byte offset, and $n_2 = 15$. The maximum size of a segment is 32768. If segment s_i has a size of 2000 bytes, logical addresses $(s_i, 2000), (s_i, 2001) \dots (s_i, 32767)$ do not exist in the process, however the address $(s_{i+1}, 0)$ would exist if segment s_{i+1} exists in the process. Even if s_i had the maximum size, i.e., 32768, the addresses $(s_i, 32767)$ and $(s_{i+1}, 0000)$ are not 'adjoining' addresses in any sense because they belong to different segments of the process.

Contrast this with a paged virtual memory system, in which a logical address lil is specified as a single number. Knowing the number of bits used to represent the page number and byte offset, the MMU extracts p_i and b_i from lil automatically. The logical address space is thus single-dimensional. If a CPU register contains the address of the last byte of a page, adding 1 to the address makes it 'spill over' into the next page! This does not happen in a segmented virtual memory since the last byte of one segment and the first byte of the next segment are not adjoining bytes in the process. This difference has important implications for memory allocation as we shall see in Section 6.10.1.

6.10.1 Management of Physical Memory

Some similarities exist with memory management in paged virtual memory. A segment fault indicates that a required segment is not present in memory. A segment-in operation is performed to load the segment. If sufficient free memory does not

exist, some segment-out operation(s) may have to precede the loading of the segment. Memory allocation for a process can be controlled using the notion of a working set of segments. Segments would be replaced on an LRU basis while servicing segment faults.

Segments do not have a fixed size. This property leads to many differences with memory management in paged virtual memory systems. The memory freed by removing one segment from memory may not suffice for loading another segment, so many segments may have to be removed before a new segment can be loaded. Differences in segment sizes also cause external fragmentation. It can be tackled using memory reuse techniques—either through compaction, or through the first fit or best fit strategies. Compaction is aided by presence of the MMU—only the address field of the segment table entry needs to be modified when a segment is moved in memory. Of course, care should be taken to ensure that segments being moved are not involved in I/O operations. The two-dimensional nature of segmented virtual memory permits a segment to dynamically grow or shrink in size. Dynamic growth can be handled by allocating a larger memory area to a segment and releasing the memory area allocated to it earlier. A segment can be permitted to grow in its present location in memory if the adjoining memory area is free.

6.10.2 Protection and Sharing

Two important issues in protection and sharing of segments are:

1. Protection against exceeding the size of a segment
2. Static and dynamic sharing of segments.

While translating a logical address (s_i, b_i) , it is essential to ensure that b_i lies within segment s_i . The MMU achieves this by comparing b_i with the size of segment s_i , which is stored in an additional field of each segment table entry.

A segment is a convenient unit for sharing because it is a logical entity in a process. If segment ids are numeric, segments must occupy identical positions in logical address spaces of sharing processes. This restriction is analogous to that concerning shared pages in a paged virtual memory system (see Section 6.5 and Figure 6.23). This restriction is not necessary if segment ids are symbolic. Protection in the logical and physical address spaces is analogous to that in paged virtual memory.

6.10.3 Segmentation with Paging

As mentioned in Section 6.10.1, segmented virtual memory systems suffer from the problem of external fragmentation because segment sizes are different. This problem can be addressed by superimposing paging on a segment oriented addressing mechanism. A system using this approach retains the fundamental advantage of segmented virtual memory—the logical address space is two-dimensional, which permits dynamic changes in the size of a segment—while avoiding external fragmentation. Each segment contains an integral number of pages and memory man-

agement is performed in terms of pages. Thus page faults are serviced as in a paged virtual memory system. Working sets are maintained in terms of pages rather than segments. This arrangement may achieve more effective utilization of memory since only required pages of a segment need to be present in memory at any time. However, paging introduces internal fragmentation.

A generalized logical address in such a system has the form (s_i, p_i, w_i) . Each segment consists of a number of pages, so a page table exists for each segment. The entry of a segment in the segment table points to the start of page table for that segment. Figure 6.28 illustrates this arrangement. (The *name* field of segment

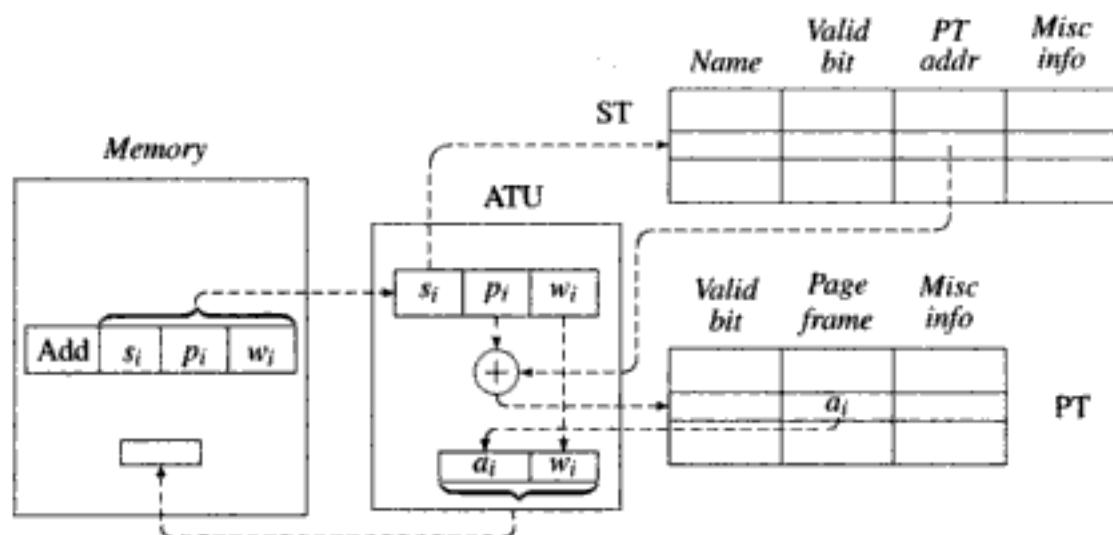


Fig. 6.28 Address translation in segmentation with paging

table is needed only if symbolic segment ids are used.) Address translation now involves two levels of indirection, first through the segment table and then through the page table of the segment. It requires two memory references if the segment and page tables are held in memory. To speed up address translation, address translation buffers would have to be employed for both the segment and page table references. A simple extension to the scheme described earlier in Section 6.2.3 can be used for this purpose. Alternatively, a single set of address translation buffers may be employed, each buffer containing a pair (s_i, p_i) and the corresponding page frame number.

Memory protection can be performed at the level of segments. It can be implemented through the segment table by putting the protection information in each entry of the segment table. Address translation buffers could contain access validation information as copied from the segment table entries. Once access validation is performed at the level of a segment, page level access validation is not necessary.

EXERCISE 6

1. The size of physical memory allocated to a process in a paged virtual memory system is held constant and the page size is varied. (This action varies the number of pages of

- the process in memory.) Draw a graph of page size vs. expected page fault rate.
2. The degree of multiprogramming in a paged virtual memory system is varied by changing the memory allocation for processes. Draw a graph of degree of multiprogramming vs. CPU efficiency. Explain the nature of the graph in the region of high degree of multiprogramming.
 3. Comment on possible loss of protection if contents of the translation look-aside buffer (TLB) are not changed when process scheduling is performed. Explain how this can be prevented.
 4. Page tables are stored in the physical memory, which has an access time of 100 nanoseconds. The translation look-aside buffer can hold 8 page table entries and has an access time of 10 nanoseconds. During execution of a process, it is found that 85% of the time a required page table entry exists in the TLB and only 2% of the references lead to page faults. The average time for page replacement is 2 milliseconds. Compute the average memory access time.
 5. Using the access speeds and hit ratios mentioned in Problem 4, compute the average memory access time in two-level, three-level and four-level page table organizations.
 6. Three approaches to paging of the kernel in a virtual memory system are:
 - (a) Make the kernel permanently memory resident.
 - (b) Page the kernel in a manner analogous to the paging of user processes.
 - (c) Make the kernel a compulsory part of the logical address space of every process in the system and manage its pages as shared pages.

Which approach would you recommend? Give reasons.

7. Performance of a process in a paged virtual memory system depends on the locality of reference displayed during its execution. Develop a set of guidelines that a programmer can follow to obtain good performance of a process. Describe the rationale behind each guideline. (*Hint:* Consider array references occurring in nested loops!)
8. It is observed that when a process is executed with $alloc = 5$, it produces more page faults when the LRU page replacement policy is used than when the optimal page replacement policy is used. Give a sample page reference string for such a process.
9. Can the FIFO page replacement policy perform better than the second-chance algorithm for a process? If so, describe the conditions under which this can happen and construct a page reference string for which the FIFO policy would perform better.
10. A process makes r page references during its execution. The page reference string of the process contains d distinct page numbers in it. The size of the process is p pages and it is allocated f page frames all through its execution.
 - (a) What is the least number of page faults that can occur during its execution?
 - (b) What is the maximum number of page faults that can occur during its execution?
11. Show validity of the following statement if the page replacement policy uses a fixed memory allocation and local page replacement: "If a process does not modify any of its pages, then it is optimal to replace the page whose next reference is farthest in the page reference string."Show that this policy may not lead to the minimum number of page-in and page-out operations if the process modifies its pages.
12. What is Belady's anomaly? Show that a page replacement algorithm that possesses

- the stack property cannot exhibit Belady's anomaly.
13. Show that the LRU page replacement policy possesses the stack property.
 14. Does the optimal page replacement policy possess the stack property?
 15. A working set allocator is used for a page reference string with two values of Δ , $\Delta_1 < \Delta_2$. pfr_1 and pfr_2 are page fault rates when Δ_1 and Δ_2 are used. Is $pfr_1 \geq pfr_2$ if working sets are recomputed (a) after every instruction, (b) after every n instructions for some n ?
 16. Describe actions of a virtual memory handler using a working set memory allocator when it decides to reduce the degree of multiprogramming. Clearly indicate how it uses and manipulates its data structures for this purpose.
 17. For the page reference string (6.5),
 - (a) Show the working set at each time instant if the size of the working set window is (i) 3 instructions, (ii) 4 instructions.
 - (b) Compare operation and performance of the working set allocator with FIFO and LRU allocators.
 18. Explain, with the help of examples, why the working set size of a process may increase or decrease during its execution.
 19. A virtual memory handler uses the following page replacement policy: When a combination of a high page fault rate and low CPU efficiency is noticed, reduce the allocation for each process and load/activate one more process for execution. Comment on effectiveness of this policy.
 20. Justify the following statement: "Thrashing can arise in a paged virtual memory system using a working set memory allocator; however, it cannot last for long."
 21. Explain why the two-handed clock algorithm for page replacement is superior to the one-handed clock algorithm (see Section 6.7).
 22. A virtual memory handler uses dynamic sharing of pages. Describe the housekeeping actions performed by it in the following situations
 - (a) When a page fault occurs.
 - (b) When a shared page drops out of the working set of one of the sharing processes.
 23. In the context of management of shared pages, discuss advantages and drawbacks of
 - (a) Protection in the logical address space
 - (b) Protection in the physical address space.
 24. A memory allocator uses the notion of 'instructions in the past' during execution of a process. The most recently executed instruction is said to be '1 instruction in the past' of the process, the instruction before it is said to be '2 instructions in the past', etc. It refers to the page reference in the instruction that is i instructions in the past as the i page reference. It uses a parameter w , and applies the following rules in the order (a)-(d) for memory allocation and replacement:
 - (a) Do nothing if the next page reference matches the w page reference.
 - (b) If the next page reference matches the i page reference for some $i < w$, do the following: if the w page reference does not match with the j page reference for some $j < w$, then reduce the memory allocation for the process by one page frame and remove the least recently used page; otherwise, do nothing.

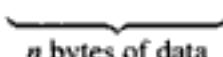
- (c) If the next page reference causes a page fault and the w page reference does not match with the page reference in the j instruction for some $j < w$, then perform a page replacement using the LRU page replacement policy.
- (d) Increase the memory allocation for the process by one page frame and load the page contained in the next page reference.

Show that the actions of the memory allocator are equivalent to actions of the working set memory allocator with $\Delta = w$.

A working set allocator is used for a page reference string with two values of Δ , $\Delta_1 < \Delta_2$. pfr_1 and pfr_2 are page fault rates when Δ_1 and Δ_2 are used. Use the equivalence of these rules with the behavior of a working set allocator to show that $pfr_1 \geq pfr_2$.

25. Write a short note on the advantages and drawbacks of using symbolic segment id's and byte id's in a segmented virtual memory.
26. Compare the following memory management proposals in a virtual memory system using segmentation with paging
 - (a) Use LRU policy within a process.
 - (b) Use LRU policy within a segment.
27. Comment on the validity of the following statement: "In a segmented virtual memory system using segmentation-and-paging, the role of segmentation is limited to sharing. It does not play any role in memory management."
28. An I/O operation consists of the execution of a sequence of I/O commands. A *self-describing* I/O operation is an I/O operation, some of whose I/O commands are read in by a previous I/O command of the same I/O operation. For example, consider the I/O operation
 1. Read $d, 6, aaa$
 2. Read $d, count, bbb$.

where d is the id of the I/O device. The first I/O command reads 6 bytes into memory area with address aaa . Let this be the area where the fields containing *count* (2 bytes) and *bbb* (4 bytes) of the second I/O command are stored. Let n and ccc be the values read into fields *count* and *bbb*, respectively, by the first I/O command. After I/O for the first I/O command is completed, the second I/O command reads n bytes into the memory area with address ccc . The data for this I/O operation would be

$n, ccc,$ 
 n bytes of data

Can the methods of performing I/O in a paged virtual memory system described in Section 6.2.3 handle self-describing I/O operations correctly? Clearly justify your answer.

In a simplified form of self-describing I/O, the first I/O command reads in only 2 bytes and stores them in the *count* field. Can the methods described in Section 6.2.3 handle such I/O operations correctly?

29. The following approach is suggested for performing I/O operations in a paged environment: While performing a read, data is first read into a temporary area. It is then moved to the area where it was supposed to have been read. While performing a write, data is first moved to a temporary area. It is then written to an I/O device.

Explain which features of I/O operations described in Section 6.2.3 need to be retained

- for use in this approach, and which features can be discarded. Explain advantages and drawbacks of this approach.
30. While initiating a process, the virtual memory handler in a paged virtual memory system copies the code of the process into the swap space reserved for the process. From the swap space, code pages are loaded into memory when needed. Some code pages may not be used during an execution, hence it is redundant to copy them into the swap space.
 To avoid redundant copying, some virtual memory handlers copy a code page into the swap space when it is used for the first time. Describe how Unix and Windows systems implement this optimization. Discuss the advantages and drawbacks of the optimization.
31. Performance of a virtual memory is determined by the interplay of three factors—CPU speed, size of the memory, and peak throughput of the paging device. Possible causes of low or high efficiency of the CPU and the paging disk can be summarized as follows:

	High efficiency	Low efficiency
CPU	Programs are CPU-bound, or CPU is slow	Few programs are CPU-bound, or thrashing present
Paging disk	Thrashing is present, or disk is slow	Overcommitment of memory to programs

Performance of a virtual memory system may improve if one or several of the following changes are made: the CPU is replaced by a faster CPU, the paging disk is replaced by a faster disk, the memory is increased, or the degree of multiprogramming is increased. In each of the following cases, which of the above changes would you recommend for improving system performance?

- (a) Low CPU efficiency and low disk efficiency
- (b) Low CPU efficiency and high disk efficiency
- (c) High CPU efficiency and low disk efficiency
- (d) High CPU efficiency and high disk efficiency.

BIBLIOGRAPHY

Randell (1969) is an early paper on the motivation for virtual memory systems. Ghanem (1975) discusses memory partitioning in VM systems for multiprogramming. Denning (1970) is a survey article on virtual memory. Hatfield (1971) discusses aspects of program performance in a virtual memory system.

Belady (1966) discusses the anomaly that carries his name. Mattson *et al* (1970) discuss stack property of page replacement algorithms. Denning (1968a, 1968b) discusses thrashing and introduced the fundamental working set model. Denning (1980) is a comprehensive discussion on working sets. Smith (1978) is a bibliography on paging and related topics. Wilson *et al* (1995) discuss memory allocation in virtual memory environments. Johnstone and Wilson (1998) discuss the memory fragmentation problem.

Chang and Mergen (1988) describe the inverted page table, while Tanenbaum (2001)

discusses the two-level page tables used in Intel 30386. Jacob and Mudge (1998) compare virtual memory features in MIPS, Pentium and PowerPC architectures. Swanson *et al* (1998) and Navarro *et al* (2002) describe superpages.

Car and Hennessy (1981) discuss the clock algorithm. Bach (1986) and Vahalia (1996) describe Unix virtual memory, Beck *et al* (2002), Bovet and Cesati (2002), Gorman (2004), and Love (2005) discuss Linux virtual memory, Mauro and McDougall (2001) discuss virtual memory in Solaris, while Russinovich and Solomon (2005) discuss Windows virtual memory.

Organick (1972) describes segmented virtual memory in MULTICS.

1. Aho, A. V., P. J. Denning, and J. D. Ullman (1971): "Principles of optimal page replacement," *Journal of ACM*, **18** (1), 80–93.
2. Bach, M. J. (1986): *The Design of the Unix Operating System*, Prentice-Hall, Englewood Cliffs.
3. Belady, L. A. (1966): "A study of replacement algorithms for virtual storage computers," *IBM Systems Journal*, **5** (2), 78–101.
4. Bensoussen, A., C. T. Clingen, and R. C. Daley (1972): "The MULTICS virtual memory—concepts and design," *Communications of the ACM*, **15** (5), 308–318.
5. Bryant, P. (1975): "Predicting working set sizes," *IBM Journal of R and D*, **19** (5), 221–229.
6. Beck, M., H. Bohme, M. Dziadzka, U. Kunitz, R. Magnus, C. Schroter, and D. Verworner (2002): *Linux Kernel Programming* (3rd edition), Pearson Education.
7. Bovet, D. P., and M. Cesati (2002): *Understanding the Linux Kernel*, O'reilly, Sebastopol.
8. Carr, W. R., and J. L. Hennessy (1981): "WSClock—a simple and effective algorithm for virtual memory management," *Proceedings of the ACM Symposium on Operating Systems Principles*, 87–95.
9. Chang, A., and M. Mergen (1988): "801 storage: architecture and programming," *ACM Transactions on Computer Systems*, **6**, 28–50.
10. Daley, R. C., and J. B. Dennis (1968): "Virtual memory, processes and sharing in MULTICS," *Communications of the ACM*, **11** (5), 305–322.
11. Denning, P. J. (1968a): "The working set model for program behavior," *Communications of the ACM*, **11** (5), 323–333.
12. Denning, P.J. (1968b): "Thrashing : Its causes and prevention," *Proceedings of AFIPS FJCC*, **33**, 915–922.
13. Denning, P. J. (1970): "Virtual Memory," *Computing Surveys*, **2** (3), 153–189.
14. Denning, P. J. (1980): "Working sets past and present," *IEEE Transactions on Software Engineering*, **6** (1), 64–84.
15. Ghanem, M. Z. (1975): "Study of memory partitioning for multiprogramming systems with virtual memory," *IBM Journal of R and D*, **19**, 451–457.
16. Gorman, M. (2004): *Understanding the Linux Virtual Memory Manager*, Prentice Hall PTR.
17. Guertin, R.L.(1972): "Programming in a paging environment," *Datamation*, **18** (2), 48–55.
18. Hatfield, D. J., and J. Gerald (1971): "Program restructuring for virtual memory," *IBM Systems Journal*, **10** (3), 169–192.

19. Jacob, B., and T. Mudge (1998): "Virtual memory in contemporary microprocessors," *IEEE Micro Magazine*, **18**, 60–75.
20. Johnstone, M. S., and P. R. Wilson (1998): "The memory fragmentation problem: solved?," *Proceedings of the First International Symposium on Memory Management*, 26–36.
21. Love, R. (2005): *Linux Kernel Development, Second edition*, Novell Press.
22. Mauro, J., and R. McDougall (2001): "Solaris Internals—CoreKernel Architecture," Sun Microsystems, Palo Alto.
23. Mattson, R. L., J. Gecsei, D. R. Slutz, and I. L. Traiger (1970): "Evaluation techniques for storage hierarchies," *IBM Systems Journal*, **9** (2), 78–117.
24. Navarro, J., S. Iyer, P. Druschel, and A. Cox (2002): "Practical, transparent operating system support for superpages," *ACM SIGOPS Operating Systems Review*, **36**, issue SI, 89–104.
25. Organick, E. I. (1972): *The MULTICS System*, MIT Press, Mass.
26. Randell, B. (1969): "A note on storage fragmentation and program segmentation," *Communications of the ACM*, **12** (7), 365–369.
27. Rosell, J. R., and J. P. Dupuy (1973): "The design, implementation and evaluation of a working set dispatcher," *Communications of the ACM*, **16**, 247–253.
28. Russinovich, M. E., and D. A. Solomon (2005): *Microsoft Windows Internals, Fourth edition*, Microsoft Press, Redmond.
29. Smith, A. J. (1978): "Bibliography on paging and related topics," *Operating Systems Review*, **12** (4), 39–56.
30. Swanson, M., L. Stoller, and J. Carter (1998): "Increasing TLB reach using superpages backed by shadow memory," *Proceedings of the 25 th International Symposium on Computer Architecture*, 204–213.
31. Tanenbaum, A. S. (2001): *Modern Operating Systems, Second edition*, Prentice-Hall, Englewood Cliffs.
32. Vahalia, U. (1996): *Unix Internals—The New Frontiers*, Prentice-Hall, Englewood Cliffs.
33. Wilson, P. R., M. S. Johnstone, M. Neely and D. Boles (1995): "Dynamic storage allocation: a survey and critical review," *Proceedings of the International Workshop on Memory Management*, 1—116.

File Systems

Computer users store programs and data in files so that they can be used conveniently and repeatedly. A user has many expectations from a file system. The obvious ones are:

- Convenient and fast access to files
- Reliable storage of files
- Controlled sharing of files with other users of the system.

The resources used for this purpose are I/O devices—their capacity to store data and their speeds of data transfer. Like with other resources, the OS has to ensure efficient use of I/O devices.

In many operating systems the above functions are organized into two components called the File System and the Input Output Control System (IOCS). The file system provides facilities that enable a user to create files, assign meaningful names to them, manipulate them, and specify how they are to be shared with other users of the system. The IOCS implements efficient organization and access of data in files. Thus use of the file system and IOCS conveniently separates file-level concerns from I/O-level concerns.

This chapter discusses the file system. Implementation of file operations using facilities provided by the IOCS is discussed in Chapter 12.

7.1 FILE SYSTEM AND IOCS

The file system and IOCS modules form the hierarchy of layers shown in Figure 7.1. Each layer contains policy and mechanism modules. The mechanisms of a layer are implemented using the policy and mechanism modules of the lower layer. The structure of the IOCS layer, as well as the number of IOCS layers, varies across operating systems. We shall discuss a conventional two-layer structure of the IOCS in Chapter 12.

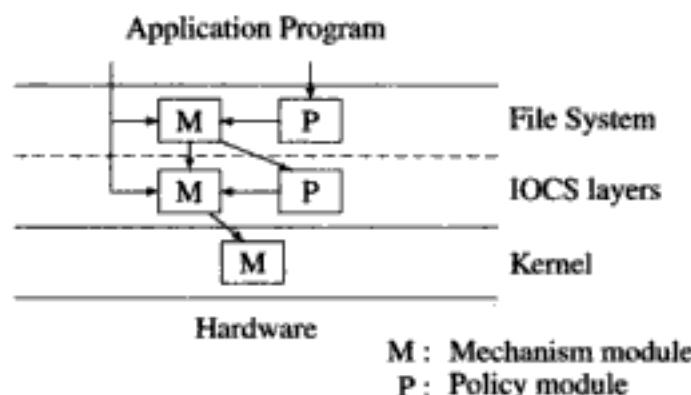


Fig. 7.1 File system and IOCS layers

As discussed in Sections 1.1 and 14.4, the hierarchy of file system and IOCS layers provides a hierarchy of abstractions. The abstraction provided by a layer simplifies design of higher layers. We begin this chapter with an overview of the functions performed by the layers of Figure 7.1.

The kernel interacts with the I/O hardware and provides facilities to handle I/O initiation and completion. The IOCS layer invokes facilities of the kernel through system calls. Using the IOCS mechanisms, it is possible to perform I/O without knowing the intricacies of I/O devices. The IOCS policy modules ensure efficient use of the I/O subsystem and good performance of file processing programs. They invoke the mechanism modules to implement I/O operations. The file system layer uses facilities provided by the IOCS layer to implement its functions. This arrangement hides all details of I/O organization and IOCS module interfaces from the application program; the application program interacts only with the file system.

File system and IOCS facilities The file system views a file as an entity that is *owned* by a user, can be *shared* by a set of authorized users, and has to be *reliably stored* over an extended period of time. As an aspect of ownership, it provides file naming freedom, so that a user can give a desired name to a file without worrying about whether other users have created identically named files, and provides privacy by protecting against interference by other users. The IOCS views a file as a set of records that need to be *accessed speedily*, and stored on an I/O device that needs to be *used efficiently*.

Table 7.1 summarizes the facilities provided by the file system and the IOCS. The file system provides directory structures that enable a user to organize his data into logical groups of files. For example a user may prefer to separate personal data from professional data and structure professional data according to activities. The file system provides protection against illegal file accesses and rules for concurrent sharing of files. It also ensures that data is reliably stored, i.e., data is not lost when system crashes occur. The IOCS provides mechanisms for performing I/O operations and ensuring efficient use of I/O devices. It also provides a set of library modules

Table 7.1 Facilities provided by the file system and the IOCS

File System <ul style="list-style-type: none"> • Directory structures for convenient grouping of files • Protection of files against illegal accesses • File sharing semantics • Reliable storage of files
IOCS <ul style="list-style-type: none"> • Efficient operation of I/O devices • Efficient access to records in a file

that permit a program to process a file efficiently.

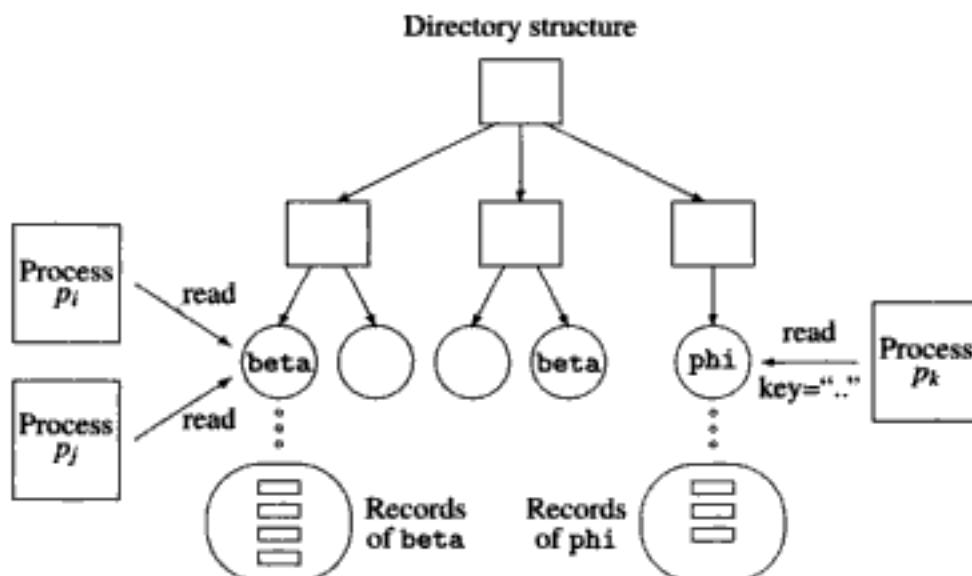


Fig. 7.2 Logical organization in file system

Logical organization in a file system Figure 7.2 shows the logical organization in a file system. The directory structure contains information concerning names, organizations and locations of files. The arrangement of data in a file is dictated by its organization. When a process wishes to manipulate a file, the file is located using the directory structure. Its data is accessed by using commands that are consistent with its organization.

Two files named **beta** exist in the file system. Processes P_i and P_j access one of these files. Which **beta** will be accessed is determined by the directory structure and identities of users who initiated processes P_i and P_j . The nature of sharing of **beta** is determined by file sharing semantics. Files **beta** and **phi** have different organizations (**beta** is a sequential file while **phi** is a direct file—see Section 7.3),

hence the commands used to access them are different.

File processing in a program We use the term *file processing* to imply reading or writing of information in a file. Figure 7.3 illustrates the arrangement used to implement file processing in an application program. The file system and IOCS provide libraries of standard modules. An application program contains declarations of files used in it, which specify values of file attributes that describe the structure and organization of data in a file. While compiling the application program, the compiler selects the appropriate file system and IOCS modules based on attributes of a file. These modules are linked with the application program. File processing actions in the application program are compiled as calls on modules of the file system and the IOCS. The file system modules invoke IOCS modules which in turn invoke the kernel to implement the operation. Programmers with advanced knowledge of IOCS may write application programs that interface directly with the IOCS (see Figure 7.1). This Chapter does not address such usage of IOCS modules.

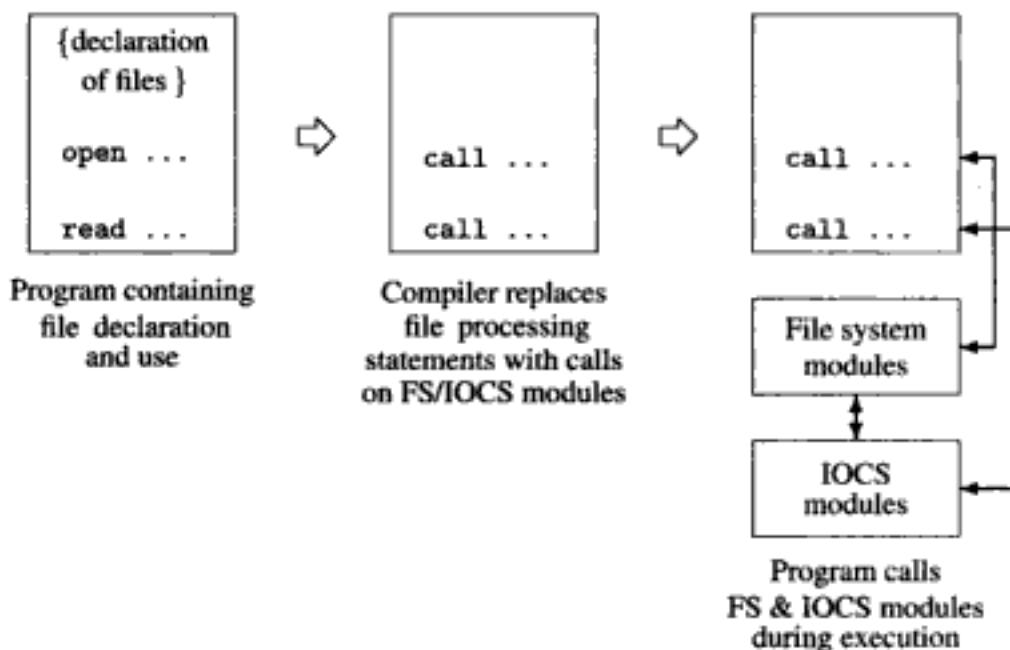


Fig. 7.3 File processing in a program

7.2 FILES AND FILE OPERATIONS

File types A file system contains different types of files, e.g., files containing data, executable programs, object modules, textual information, documents, spreadsheets, photos and video clips. Each of these file types has its own format. These file types can be grouped into two classes:

- Structured files

- Byte-stream files.

A *structured file* is a classical view of a file consisting of records and fields. In this view, a file is a named collection of records, a *record* is a meaningful collection of related fields, and a *field* contains a single data item. A record is also a meaningful unit for processing of data. Each record in a file is assumed to contain a key field. Contents of key fields of all records in a file are unique. Many file types mentioned earlier are structured files. File types used by standard system software like compilers and linkers have a structure determined by the OS designer. Structure of a filetype used by an application such as a spreadsheet program is determined by the application itself. The structure of a data file is decided by the program that creates it.

A *byte-stream file* is ‘flat’. There are no fields and records in it; it is looked upon as a sequence of bytes by the processes that use it. Unix uses byte-stream files to store data and programs.

Example 7.1 Figure 7.4 shows a file `employee.info`. Each record in the file contains information about one employee. A record contains four fields: employee id, name, designation and age. The field containing the employee id is the key field.

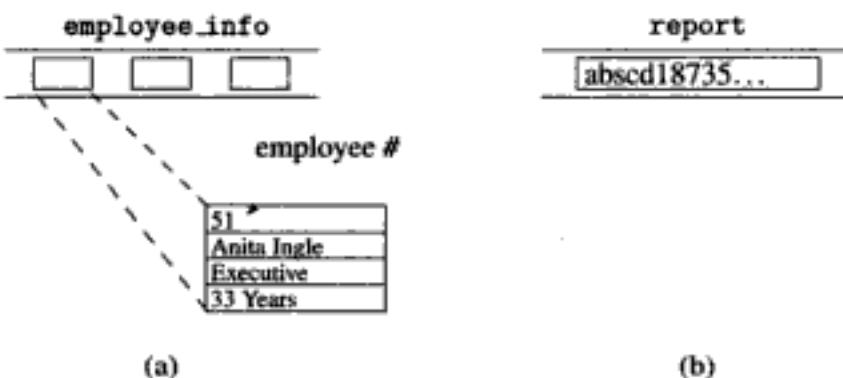


Fig. 7.4 Logical views of (a) A structured file `employee.info`, (b) A byte-stream file `report`

File attributes File attributes are characteristics of a file that are important to its users or to the OS. The common attributes of a file are: type, organization, size, location in the system, access control information that indicates the manner in which different users can access the file, owner name and time of last use.

File operations Table 7.2 describes the operations performed on files. As mentioned earlier, the OS is responsible for creating and maintaining a file, for ensuring that it is accessed by users only in accordance with access privileges specified for it, and for deleting the file when requested by its owner. These functions are performed using the file system interface. Actual access of files, i.e., reading or writing of records, is implemented using the IOCS.

Table 7.2 File operations performed by processes

Operation	Description
Opening a file	A process executes an <code>open</code> statement before performing file processing. The file system locates the file and checks whether the user executing the process has the necessary access privileges. It also performs some housekeeping actions to initiate processing of the file.
Reading or writing a record	The process issues an appropriate command to read or write a record. The file system considers the organization of the file (see Section 7.3) and implements the read/write operation in an appropriate manner.
Closing a file	Execution of a <code>close</code> statement indicates to the file system that processing of the file is completed. The file system updates information concerning the size of the file, i.e., number of records in it, in its data structures.
File creation	A file is created by making a copy of an existing file, or by writing records in a new file. The file system stores information concerning size of the file in its data structures. It also stores information concerning access privileges for the file.
File deletion	The file is deleted from the directory structure of the file system. The secondary storage medium occupied by the file is freed.
File renaming	The file system remembers the new name of the file in its directory structure.
Specifying access privileges	The user can specify or change access privileges of a file created by his process at any time during the file's lifetime.

The next Section describes various ways of organizing data in a structured file. The later Sections discuss file access preliminaries and provisions required to ensure that a process accesses a file only in accordance with its access privileges.

7.3 FUNDAMENTAL FILE ORGANIZATIONS

We informally use the term 'record access pattern' to describe the order in which records in a file are accessed by a process. Two fundamental record access patterns are sequential access, in which records are accessed in the same order in which they exist in a file (or in the reverse order), and random access, in which records are accessed in some order other than the sequential order. A program would execute efficiently if its record access pattern while processing a file can be implemented efficiently in the file system.

A file organization defines two things concerning a file—the arrangement of

records in the file, and the procedure to be used to access the records. Its use provides efficient record access for a specific record access pattern. The file organization determines how efficiently the I/O medium would be used. A file system provides several file organizations, so that a program can select a file organization that best suits its requirements.

We use the following notation while discussing the file processing activity in a process:

- t_p : CPU time required to process the information in a record
- t_w : Wait time per record, i.e., time since the request for a record by a process to the time when the record becomes available for processing

File processing efficiency depends on the value of t_w . A file organization exploits characteristics of an I/O device to provide good file processing efficiency for a specific access pattern. For example, a disk record has a unique address and a read/write operation can be performed on any disk record by specifying its address. This fact can be used to efficiently implement the random access pattern.

This Section describes three fundamental file organizations. Other file organizations used in practice are either variants of these fundamental organizations or are special purpose organizations that suit less commonly used I/O devices. Accesses to files that use a specific file organization are implemented by an IOCS module called an *access method*. We describe functions performed by access methods after discussing the fundamental file organizations.

7.3.1 Sequential File Organization

In the sequential file organization, records are stored in an ascending or descending sequence by the key field. The access pattern of an application is expected to be a subsequence of this sequence. Accordingly, a sequential file supports two kinds of operations: read the next (or previous) record, and skip the next (or previous) record. A sequential file is used in an application if its data can be conveniently presorted into an ascending or descending order.

Most I/O devices can be accessed sequentially, hence sequential files are not crucially dependent on device characteristics. Consequently, a sequential file can be migrated easily to a different kind of device. The sequential file organization is also used for byte-stream files.

Example 7.2 A master file of employee data is organized as a sequential file. The employee number is the key field of an employee record and records are arranged in ascending order by employee number (see Figure 7.5(a)). Each record in the master file contains the bank account number of the employee where all payments are to be deposited. When some special payment for a specific class of employees is to be processed, their employee numbers are sorted in ascending order. To handle special payments, a process reads an employee number, obtains the master file record for that employee and processes the payment for crediting in the employee's bank account.

Since the employee numbers are presorted in an ascending order, it is possible to perform this processing by always reading or skipping the next record.



Fig. 7.5 Records in (a) sequential file, (b) direct file

7.3.2 Direct Access File Organization

The direct file organization provides convenience and efficiency of file processing when records are accessed in a random order. To access a record, a read/write command only needs to mention its key; hence access to a record is independent of which record was accessed prior to it. Contrast this with the sequential file organization. If a process wishes to access the record for employee numbered 125, and if the last record read from the file was for employee numbered 36, it would have to explicitly skip the intervening records if it used the sequential file organization. These actions are avoided in a direct file organization, which makes it both convenient and efficient.

Direct files are recorded on disks. When a process provides a key value, the access method for the direct file organization applies a transformation to the key value to generate a (*track_no*, *record_no*) address. The disk heads are now positioned on track *track_no* before issuing a read or write command on record *record_no*. Consider the master file of employee information used in Example 7.2, this time organized as a direct file. Let *p* records be written on one track of the disk. Assuming the employee numbers and the track and record numbers of the master file to start from 1, the address of the record for employee number *n* is (*track number*(*t_n*), *record number*(*r_n*)) where

$$t_n = \lceil \frac{n}{p} \rceil \quad (7.1)$$

$$r_n = n - (t_n - 1) \times p \quad (7.2)$$

and $\lceil \dots \rceil$ indicates a rounded-up integer value.

The direct file organization provides access efficiency when records are processed randomly; however, it has two drawbacks compared to a sequential file:

- Record address calculation consumes CPU time.
- Poor utilization of the I/O medium results from the need for dummy records. Example 7.3 illustrates this aspect.

Hence sequential processing of records in a direct file is less efficient than processing of records in a sequential file.

Example 7.3 Figure 7.5 shows the arrangement of employee records in sequential and direct file organizations. Employees with the employee numbers 3, 5–9 and 11 have left the organization. However, the direct file needs to contain a record for each of these employees to satisfy the address calculation formulae (7.1)–(7.2). This requirement leads to the need for dummy records in the direct file.

Another practical problem in the use of direct files is excessive device dependence. Characteristics of an I/O device are explicitly assumed and used by the address calculation formulae (7.1)–(7.2). Rewriting the file on another device with different characteristics, e.g., different track capacity, will imply modifying the address calculation formulae.

7.3.3 Index Sequential File Organization

An index helps to determine the location of a record from its key value. In a purely indexed file organization, the index contains entries of the form (key, disk address) for all key values existing in a file. To access a record with key k , the index entry containing k is found by searching the index, and the disk address mentioned in the entry is used to access the record. If an index is smaller than a file, this arrangement provides high access efficiency because search in the index is more efficient than search in the file.

The index sequential file organization is a hybrid organization that uses elements of the indexed and the sequential file organizations to combine some of their advantages and avoid some of their drawbacks. It uses an index to identify a section of the disk surface that may contain a desired record. Records in this section of the disk are searched sequentially to find the record. The search succeeds if the record is present in the file; otherwise, it results in a failure. This arrangement requires a much smaller index than in a purely indexed file because all key values do not exist in the index. It also provides better access efficiency than a sequential file organization while ensuring comparable efficiency of I/O media utilization.

For a large file the index would contain a large number of entries, hence the time required to search through the index would be large. A higher level index can be used to reduce the search time. An entry in the higher level index points to a section of the index. This section of the index is searched to find the section of the disk that may contain a desired record. Example 7.4 illustrates this arrangement.

Example 7.4 Figure 7.6 illustrates a master file of employee information organized as an index-sequential file. Records are stored in ascending order by the key field. Two indices are built to facilitate speedy search. The track index indicates the smallest and largest key value located on each track (see fields named Low and High in Figure 7.6). The higher level index contains entries for groups of tracks containing 3 tracks each. To locate the record with key k , first the higher level index is searched to locate the group of tracks that may contain the desired record. The track index for the tracks of the group is now searched to locate the track that may contain the desired record, and

the selected track is searched sequentially for the record with key k . The search ends unsuccessfully if it fails to find the record on the track.

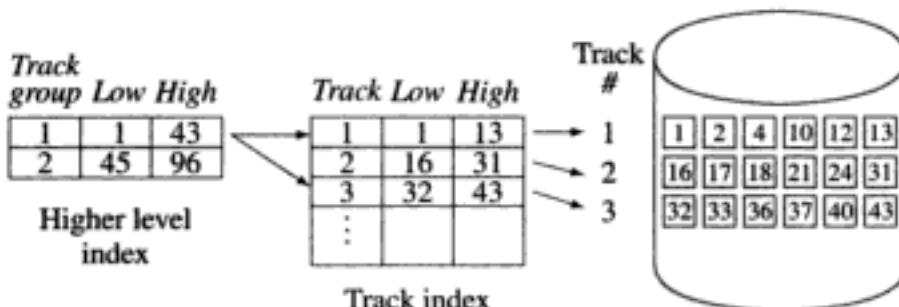


Fig. 7.6 Track index and higher level index in an index sequential file

7.3.4 Access Methods

An access method is a module of the IOCS, which implements accesses to a class of files using a specific file organization. The procedure to be used for accessing records in a file, whether by a sequential search or by address calculation, is determined by the file organization. The access method uses this procedure to access records. It may also use some advanced techniques in I/O programming to make file processing more efficient. Two such techniques are *buffering* and *blocking* of records.

Buffering of records Records of an input file are read ahead of the time when they are needed by a process. The access method holds these records in memory areas called *buffers* until they are required by the process. The purpose of buffering is to reduce or eliminate t_w ; the process faces a wait only when the required record does not already exist in a buffer.

The converse actions are performed for an output file. When the process performs a write operation, the value to be written into the file is copied into a buffer. It is written on the I/O device sometime later and the buffer is released for reuse. The process faces a wait only if a buffer is not available when it performs a write operation.

Blocking of records A large block of data is always read from, or written onto, the I/O medium. The size of this block exceeds the size of a record in the file. This arrangement reduces the total number of I/O operations required for processing a file, which improves file processing efficiency of a process. Blocking also improves utilization of an I/O medium and throughput of a device.

We discuss the techniques of buffering and blocking of records in Chapter 12.

7.4 DIRECTORY STRUCTURES

A file system contains files owned by several users. Two features are important in this context:

- *Naming freedom:* A user's ability to give any desired name to a file, without being constrained by file names used by other users.
- *Sharing of files:* A user's ability to access files created by other users, and ability to permit other users to access his files.

A file system uses directories to provide both these features. A directory contains information about a group of related files. Each entry in it contains information concerning one file—its location, type, and the manner in which it may be accessed by other users in the system. When a process issues a command to open a file, the file system finds the file's entry in a directory and obtains its location.

Figure 7.7 shows fields in a typical directory entry. The *Flags* field is used to differentiate between different kinds of directory entries. We put the value 'D' in this field to indicate that a file is a directory, 'L' to indicate that it is a link, and 'M' to indicate that it is a mounted file system. Later sections describe these uses. The *Misc info* field contains information like owner, time of creation and time of last modification.

<i>File name</i>	<i>Type/size</i>	<i>Location info</i>	<i>Protection info</i>	<i>Flags</i>	<i>Misc info</i>

Fig. 7.7 A typical directory entry

A file system contains several directories. The directory structure of the file system connects a directory to other directories in the system. It governs the manner in which files may be shared by a group of users. We use the pictorial convention that a directory is represented by a rectangle, while a file is represented by a circle. Figure 7.8 shows a simple directory structure containing two kinds of directories. A *user directory* (UD) contains information about files owned by one user; each file is described by one entry in the directory. The *master directory* (MD) contains information about UDs of all registered users of the system. Each entry in the MD contains a pair (user id, UD pointer). Users A and B have both created a file named alpha. These files have entries in the respective UDs. We refer to the directory structure shown in Figure 7.8 as a two-level directory structure.

Use of UDs provides naming freedom. When a process initiated by user A performs the call `open(alpha, ...)`, the file system searches the MD to locate A's UD, and searches for alpha in it. If the call `open(alpha, ...)` was performed by some process executed by B, the file system would have searched B's UD for alpha. This arrangement ensures that the correct file would be accessed even if many files with identical names exist in the system.

Use of UDs has one drawback—it prohibits users from sharing their files with other users. A special syntax may be provided to enable a user to refer to another user's file. For example, a process initiated by user C may perform the `open(A->alpha, ...)` to open A's file alpha. The file system can implement

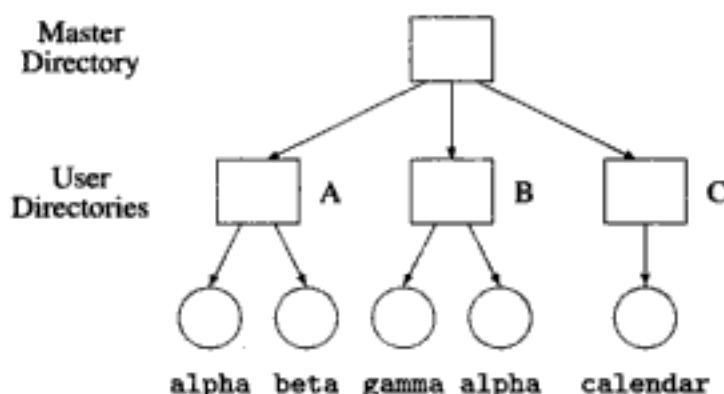


Fig. 7.8 Master and user directories

this simply by using A's UD, rather than C's UD, to search and locate file *alpha*. To implement file protection, the file system must determine whether user C may be permitted to open A's file *alpha*. It checks the *protection info* field of *alpha*'s directory entry for this purpose. Details of file protection are discussed in Section 7.5.

The two-level directory structure of Figure 7.8 can be generalized in many interesting ways to provide more flexibility to users. Table 7.3 summarizes three important generalizations described in the following.

Table 7.3 Generalizations of the two-level directory structure concept

Generalization	Advantages
Multi-level structure	Multiple levels of directories permit a user to structure his files into functionally related levels and sublevels.
Directory as a file	A user can create files and directories within a directory. He can also give a desired name to a directory. Effectively, a user can customize the directory structure.
Generalized syntax for file access	The generalized syntax permits a user to access any file in the file system, subject to the constraints imposed by file protection.

The first generalization concerns the number of levels in the directory structure. The file system may provide a fixed multi-level directory structure in which each user has a UD containing a few directories, and some of these directories contain other directories. A user can utilize this feature to group files according to some meaningful criterion such as the nature of activities to which they pertain. However, this approach lacks flexibility since it provides a fixed number of levels in the hierarchy and a fixed number of directories at each level.

The second generalization provides more flexibility—a directory is a file and it can be created the same way a file is created. A 'D' can be put in the *flags* field of a

file's entry to indicate that it is a directory file. The file system provides a directory called `root` that contains information about UDs of all users. A user creates directory files and data files to structure his information as needed. Figure 7.9 shows the directory tree for user A. The user has created a file called `alpha` and directories called `admin` and `projects` in the UD provided by the file system. The `projects` directory contains a directory `real_time`, which contains a file `main.pgm`. The directory trees of all users together constitute the directory tree of the file system. The root of this tree is the file system root directory `root`.

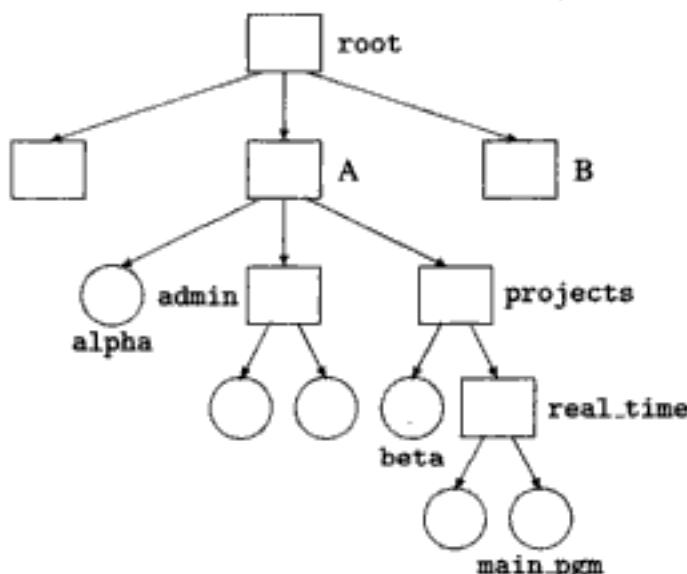


Fig. 7.9 Files of user A

At any moment of time, a user is said to be 'in' some specific directory. This directory is called the *current directory* of the user. When the user wishes to open a file, the filename is searched for in this directory. While registering a user with the OS, some directory in the file system directory hierarchy is specified as his *home directory*. Whenever the user logs in, the OS puts him in the home directory. This action makes home directory the current directory of the user. A user may change the current directory to some other directory through a *change directory* command. To facilitate this, each directory stores information about its parent directory in the directory structure.

The third generalization concerns the syntax for accessing files. Since a filename may not be unique in the file system, to access a file a user or a process must identify it in an unambiguous manner. An *access path* is a path in the directory hierarchy, which provides an unambiguous method of designating a file. A user or a process uses a *path name* to indicate which file is to be accessed. A *path name* is a sequence of one or more path components separated by '/', where each path component is a reference through a directory and the last path component may be the name of a file.

Table 7.4 contains examples of path names. The first path name contains a single path component that is the name of a file. This name is to be searched for in the current directory. In the third path name the notation `-xyz` signifies the home directory of user `xyz`.

Table 7.4 Examples of access paths

Path name	Explanation
<code>alpha</code>	file <code>alpha</code> in the current directory
<code>projects/beta</code>	file <code>beta</code> in the directory <code>projects</code> contained in the current directory
<code>-xyz/alpha</code>	file <code>alpha</code> in the home directory of user <code>xyz</code>
<code>../personal/tax</code>	file <code>tax</code> in directory <code>personal</code> , which exists in the parent of the current directory

Path names starting on the current directory are called *relative path names*. Relative path names are often short and convenient to use, however they can be confusing because a file may have a different relative path name from different directories. For example, in Figure 7.9, file `alpha` has the relative path name `alpha` from directory `A`, whereas it has the relative path names `../alpha` and `.../alpha` from directories `projects` and `real_time`, respectively.

An absolute path name starts on the root directory of the file system directory hierarchy. We will use the convention that the first path component in an absolute path is a null symbol. Thus, in Figure 7.9, the absolute path name of file `alpha` is `/A/alpha`. Identically named files created in different directories differ in their absolute path names.

7.4.1 Tree and Graph Directory Structures

The directory structure illustrated in Figure 7.9 is a tree. In this structure, each file except the root directory (note that a directory is also a file) has exactly one parent directory. The advantage of the tree structure is that it provides total separation of different users' files, thereby providing complete naming freedom to users. However, it makes file sharing rather cumbersome. A user wishing to access another user's files has to do so through two or more directories. For example, in Figure 7.9, user B can access file `beta` using the path name `../A/projects/beta` or `~A/projects/beta`.

Use of the tree structure leads to a fundamental asymmetry in the way different users can access a shared file. The file would exist in some directory belonging to one of the users, who can access it with a shorter path name than other users. This problem can be solved by organizing the directories in an acyclic graph structure. In this structure, a file can have many parent directories, hence a shared file can be pointed to by directories of all users who have access to it. Links can be used to construct acyclic graph structures.

Links A *link* is a directed connection between two existing files in the directory structure. It can be written as a triple (*<from_filename>*, *<to_filename>*, *<link_name>*), where *<from_filename>* is a directory file and *<to_filename>* could be a directory or data file. Once the link is established, the *<to_filename>* can be accessed as if it were a file named *<link_name>* in the directory *<from_filename>*. The fact that *<link_name>* is a link in the directory *<from_filename>* is indicated by putting the value 'L' in its *flags* field. Example 7.5 illustrates how a link is set up.

Example 7.5 Figure 7.10 shows the directory structure after user B creates a link using the command (-B, -A/projects/beta, s_ware). The name of the link is *s_ware*. The link is made in the directory *-B* and it points to the file *-A/projects/beta*. This link permits *-A/projects/beta* to be accessed by the name *-B/s_ware*. Of course, user B should have the necessary access privileges to *beta* in order to set up this link.

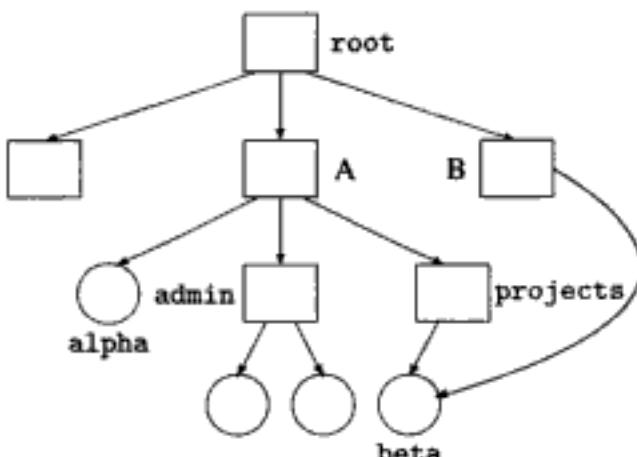


Fig. 7.10 A link in the directory structure

To facilitate file deletion, which we describe later, the file system maintains a count indicating the number of links pointing to a file. An *unlink* command deletes a link and decrements the count. Implementation of the *link* and *unlink* commands therefore involves manipulation of directories that contain files *<from_filename>* and *<to_filename>*. Deadlocks may arise while implementing *link* and *unlink* commands if several processes issue these commands simultaneously. The file system can use some simple deadlock prevention policy to ensure absence of deadlocks (see Chapter 11).

Operations on directories Common operations on directories are maintenance operations like creating or deleting files, updating file entries when a process performs a close operation, listing the directory, and deleting the directory. Most of these operations require modifications to be made in the directory.

The delete operation becomes complicated when directory structures resemble graphs because a file may have multiple parents. The file system should check

whether the file to be deleted has a single parent. If this is the case, the file is deleted and its entry is removed from its parent. Otherwise, the file is not deleted because that would lead to dangling pointers; however, its entry is removed from the parent directory involved in the access path provided in the delete command.

It is cumbersome to check whether a file has multiple parent directories. This task is simplified by maintaining a reference count with each file. The count is set to 1 when the file is created, and it is incremented by 1 whenever a link is set to point to the file. When an attempt is made to delete a file, its reference count is decremented by 1 and the file's entry is deleted from the parent directory involved in its access path provided in the delete command. The file itself is deleted if the new value of its reference count is 0.

This simple strategy is not adequate if the directory structure contains cycles. Cycles develop when a link is set from a directory to one of its ancestor directories, e.g., if a link were set up from directory `real.time` in Figure 7.9 to directory `A`. Deletion of `A` from directory `root` would lead only to deletion of `A`'s entry in `root` because `A`'s new reference count is 1. However, there is no reason to retain directory `A`, and files reachable from it, since `A` is not accessible from the home directory of any user! This problem can be solved either by using a technique to detect cycles not reachable from any home directories, which can be expensive, or by preventing cycles from arising in the directory structure, which is equally expensive.

7.4.2 Mounting of File Systems

A file system is constituted on a logical device, i.e., on a partition of a disk. Files contained in a file system can be accessed only when the file system is mounted. This operation 'connects' the file system to the system's directory structure. An unmount operation disconnects a file system. The mount and unmount operations are performed by the system administrator. This feature provides an element of protection. Mounting of file systems is useful when more than one file system exists in the system (see Section 7.12), or when a user of a distributed system wishes to access files located in a remote machine (see Chapter 19).

A mounted file system is typically considered to be a file in the root directory of the host file system. However, some operating systems generalize the concept of mounting and allow a file system to be mounted at any point in the directory structure of another file system. Such mounting provides an effect analogous to that provided by a link. The difference is that mounting does not permanently alter the directory structure. Its effect lasts until the file system is unmounted or until the system is booted again.

Mounting is implemented as follows: Certain files in the file system hierarchy are designated as *mount points*. A file system can be mounted at a mount point by issuing the command `mount (<FS_name>, <mount_point_name>)`; where `<FS_name>` and `<mount_point_name>`, both of which are path names, designate the root of the file system to be mounted and the mount-point, respectively. After the mount op-

eration has been performed, any file with the relative path name ap_i in directory $<FS_name>$ can be accessed by the path name $<mount_point.name>/ap_i$. Example 7.6 illustrates the effect of executing *amount* command.

Example 7.6 Figure 7.11(b) shows the effect of the mount command `mount (meeting, -A/admin)`; where the file system hierarchies at `meeting` and `admin` are as shown in Figure 7.11(a). File items can now be accessed as `-A/admin/agenda/items`.

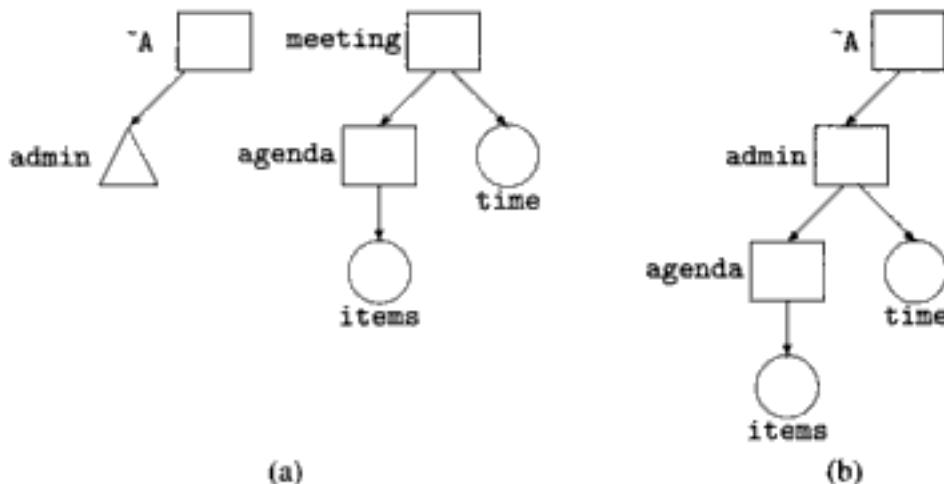


Fig. 7.11 Mounting of a file system

The effect of a mount operation is nullified by a command `umount (<FS_name>, <mount_point.name>)`. The umount operation succeeds only if no files of the mounted file system are currently open. To check this condition easily, the file system keeps a count in the root of the mounted file system to indicate how many files have been opened.

7.5 FILE PROTECTION

The owner of a file can specify the manner in which other users can access his file. Protection information is specified while creating a file, and may be changed by the owner of the file at any time during the file's lifetime. It is stored in the *protection info* field in the directory entry of a file (see Figure 7.7). The file system uses this information to provide controlled sharing of files.

It is convenient to store the protection information in the form of an *access control list* (ACL). Each element of the access control list is an access control pair ($<user.name>$, $<list.of.access.privileges>$). When a process initiated by some user X tries to perform an operation $<opn>$ on file alpha, the file system finds the access control pair with $<user.name> = X$ and checks to see whether $<opn>$ is contained in $<list.of.access.privileges>$. The attempt to access alpha fails if this is not the case. For example, a write attempt by X would fail if the entry for user X in the access control list is (X, read), or if no entry for X exists in ACL.

Size of the access control list of a file depends on the number of users and the number of access privileges defined in the system. Most file systems use three kinds of access privileges—*read*, *write* and *execute*. The *read* and *write* privileges indicate whether the file can be read or modified by a user. A *write* privilege permits existing data in the file to be modified and also permits new data to be added. One can differentiate between these two privileges by defining a new access privilege called *append*, however this would increase the size of the protection information. The *execute* privilege permits a user to execute the program contained in a file.

Access privileges have different meanings for directory files. The *read* privilege for a directory file implies that one can obtain a listing of the directory, while the *write* privilege for a directory implies that one can create new files in the directory. The *execute* privilege for a directory permits an access to be made through the directory—that is, it permits a file existing in the directory to be accessed. A user can use the execute privilege of directories to make a part of his file hierarchy visible to other users.

If a system contains a large number of users, it is not feasible to use an access control pair for every user. To reduce the size of protection information, users can be classified in some convenient manner and an access control pair can be specified for each user class rather than for each individual user. Now an access control list has only as many pairs as the number of user classes. Example 7.7 describes the access control list used in Unix. Chapter 8 discusses file protection in detail.

Example 7.7 The Unix operating system limits the size of the access control list of a file alpha by dividing all users in the system into the following three classes:

- Class 1 : Owner of file alpha
- Class 2 : Users in the same group as the owner of alpha
- Class 3 : All other users in the system

Three access privileges, viz. *read*, *write* and *execute*, are defined. The directory entry of alpha contains the user id of its owner, and access privileges assigned to each user class. Thus there are only three access control pairs in each access control list.

7.6 INTERFACE BETWEEN FILE SYSTEM AND IOCS

The file system uses IOCS mechanisms to implement I/O operations. These mechanisms use I/O programming to handle device level details of I/O initiation and interrupt handling. This arrangement shields the file system from machine dependent features. The interface between the file system and IOCS consists of the file control block and functions that perform I/O operations.

A *file control block* (FCB) contains all information concerning a file processing activity (see Table 7.5). Information in the FCB is derived from a variety of sources. Information concerning the organization of a file is derived from the file declaration contained in an application program. While compiling the program, the compiler creates the FCB in its object code and records this information in the FCB.

Directory information becomes available through joint actions of the file system and the IOCS. The file system decides the location of a file in the system when it is created, and stores this information in the directory entry of the file. IOCS copies this information into the FCB when a file is opened for reading. Information concerning the current state of processing is written into the FCB by IOCS. This information is continuously updated during the processing of a file.

Table 7.5 Fields in the file control block (FCB)

Category	Fields
File organization	File name File type, organization and access method Device type and address Size of a record Size of a block Number of buffers Name/address of IOCS module/access method
Directory information	Address of parent directory's FCB Address of the file map table (FMT) (or the file map table itself)
Current state of processing	Address of the next record to be processed Addresses of buffers.

The file system supports the following operations:

- `open (<fcb_address>, <processing_mode>);`
- `close (<fcb_address>);`
- `read/write (<fcb_address>, <record_info>, <I/O_area_addr>);`

Each operation takes an FCB address as its first operand. The `open` operation takes the processing mode, whether input, output or append, as an optional parameter. The `read/write` operations take the parameter `<record_info>`, which indicates the identity of the record to be read or written. This parameter is required in all file organizations except the sequential file organization (see Section 7.3). The parameter `<I/O_area_addr>` indicates the address of the memory area where data from the record should be read, or which contains the data to be written into the record.

Each file system operation extracts relevant information from the FCB and the directory structure, and invokes an appropriate operation of the IOCS. The `open` operation extracts the file name from the FCB, refers to the directory hierarchy and locates the directory entry for the file. It passes address of the parent directory's FCB and its own parameters to IOCS. IOCS copies information from the file's directory entry into the FCB. The `close` operation passes address of the parent directory's FCB and other parameters to IOCS for updating the directory entry of the file. A `read/write` operation simply passes all its parameters to IOCS.

The IOCS interface supports the following operations:

- `iocs-open (<fcb_address>, <directory_entry_address>, <processing_mode>);`
- `iocs-close (<fcb_address>, <directory_entry_address>);`
- `iocs-read/write (<fcb_address>, <record_info>, <I/O_area_addr>);`

`iocs-open` and `iocs-close` operations are specialized read and write operations that copy information into the FCB from the directory entry or from the FCB into the directory entry. The `iocs-read/write` operations use information concerning current state of file processing found in the FCB to implement a read/write operation. When a write operation requires more disk space, `iocs-write` would invoke a function of the file system to perform disk space allocation (see Section 7.7).

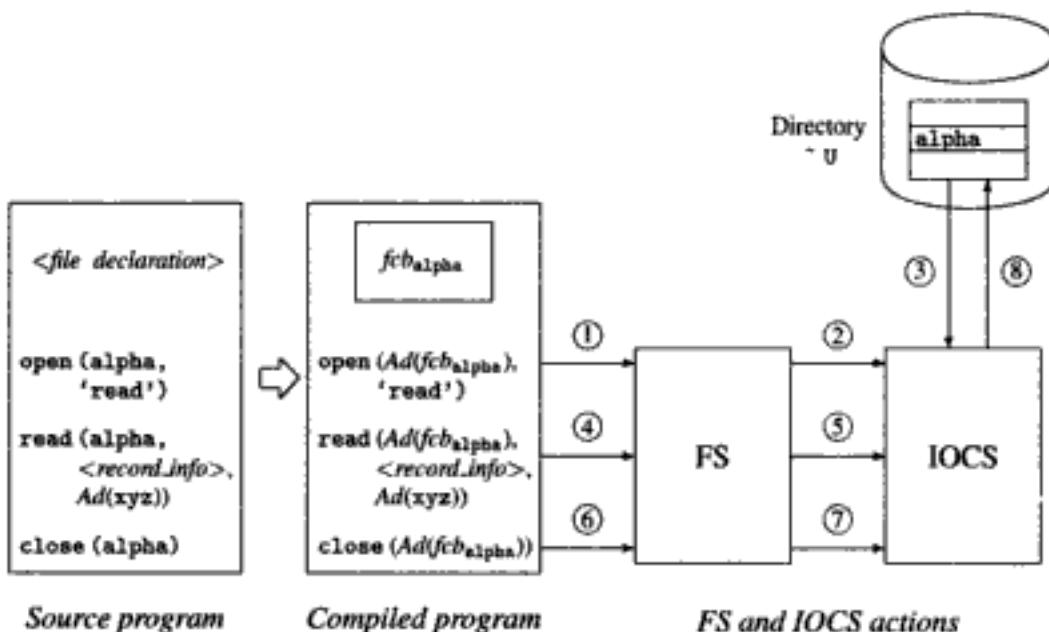


Fig. 7.12 Overview of file operations

Figure 7.12 shows a schematic of the processing of file alpha in a process initiated by some user U. The source program contains a declaration for file alpha. In the compiled version of this program, the compiler creates an FCB for file alpha. We denote it as fcb_{alpha} . The compiler also replaces statements `open`, `read` and `alpha` in the source program by calls on file system operations `open`, `read` and `alpha`, respectively. The first parameter in all these calls is the address of fcb_{alpha} instead of the file name `alpha`. Significant steps in execution of this program are shown by numbered arrows in Figure 7.12; they are described in the following.

1. The process representing execution of this program performs the call `open(Ad(fcb_alpha), 'read')`.

2. The open operation extracts the file name alpha from fcb_{alpha} , locates the directory entry of alpha, and stores its address in fcb_{alpha} for use while closing the file. It now makes a call `iocs-open` with $Ad(fcb_{\text{alpha}})$ and the address of the directory entry of alpha as parameters.
3. IOCS copies information concerning file size and file location, viz. addresses of the first and last byte/block, from the directory entry into fcb_{alpha} .
4. When the process wishes to read a record of alpha into area xyz, it invokes the `read` operation of the file system with $Ad(fcb_{\text{alpha}})$, $<\text{record_info}>$ and $Ad(xyz)$ as parameters.
5. The file location information of alpha is now available in fcb_{alpha} , so it is not necessary to refer to the directory entry of alpha to implement a `read/write` operation; these operations can directly invoke `iocs-read/write` operations. In fact, the process may directly invoke `iocs-read/write`.
6. The process invokes the `close` operation with $Ad(fcb_{\text{alpha}})$ as a parameter.
7. The file system makes a call `iocs-close` with $Ad(fcb_{\text{alpha}})$ as a parameter.
8. IOCS copies information concerning file size and file location, viz. addresses of the first and last byte/block, from fcb_{alpha} into the directory entry of alpha.

7.7 ALLOCATION OF DISK SPACE

In Section 7.6 we mentioned that disk space allocation is performed by the file system. While creating or updating a file, IOCS expects a FS module to supply the address of a disk block in which a record should be written. For simplicity, early file systems adapted the contiguous memory allocation model (see Section 5.5) by allocating a single contiguous disk area to a file. To facilitate this, a process was required to provide an estimate of file size while creating a new file. This approach led to both internal and external fragmentation of disk space.

Contiguous disk space allocation also required complicated arrangements to avoid use of bad disk blocks. The file system would identify bad disk blocks while formatting the disk and note their addresses. It would then allocate alternate disk blocks for the bad ones and build a table showing addresses of bad blocks and their alternates. During a read/write operation, the IOCS would check whether the disk block to be accessed was a bad block. If so, it would obtain address of the alternate disk block and access it.

Contemporary systems avoid these problems by adapting the noncontiguous memory allocation model (see Section 5.6) to disk space allocation. This approach avoids external fragmentation. Fixed sized disk blocks are allocated on demand while creating or updating a file. This reduces the average fragmentation per file to half the size of a disk block. Access to data now requires an equivalent of 'address translation' (see Figure 5.16). Tables storing information about disk blocks allocated to a file are maintained for this purpose.

We discuss two approaches to noncontiguous disk space allocation. Following Section 7.4, it is assumed that information concerning disk blocks allocated to a file can be obtained from the *Loc info* field of its directory entry.

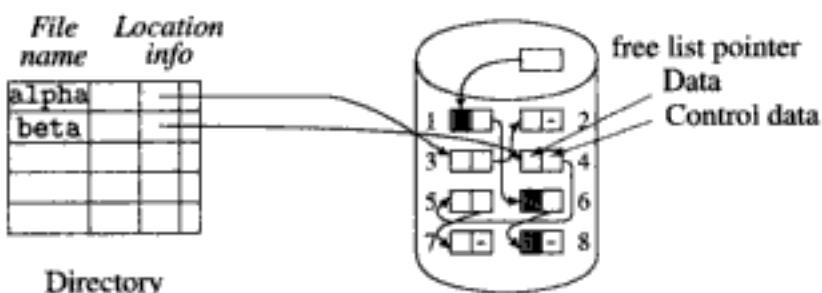


Fig. 7.13 Linked allocation of disk space

Linked allocation Figure 7.13 illustrates linked allocation. A file is represented by a linked list of disk blocks. Each disk block has two fields in it—*data* and *control info*. The *data* field contains the data written into the file, while *control info* contains address of the next disk block allocated to the file. The *Loc info* field of the directory entry of a file points to the first disk block of the file. Other blocks are accessed by following the pointers in the list of disk blocks. Free space on the disk is represented by a *free list*. When a disk block is needed to write a new record of a file, a block is taken off the free list and added to the file's list of disk blocks. To delete a file, the file's list of disk blocks is simply added to the free list. This action saves a lot of processing time during file deletion.

Linked allocation is simple to implement, and incurs a low allocation/deallocation overhead. It also supports sequential files quite efficiently. However, files with non-sequential organizations cannot be accessed efficiently. Reliability is also poor since corruption of the control information field in a disk block may lead to loss of data in the entire file. Similarly operation of the file system may be disrupted if a pointer in the free list is corrupted. These issues are discussed in Section 7.10.

File allocation table (FAT) MS-DOS uses a variant of linked allocation that stores the control data separately from the file data. A *file allocation table* (FAT) of a disk is an array that has one element corresponding to every disk block in the disk. For a disk block that is allocated to a file, the corresponding FAT element contains the address of the next disk block. Thus the disk block and its FAT element together form a pair that contains the same information as the disk block in a classical linked allocation scheme.

The directory entry of a file contains the address of its first disk block. The FAT element corresponding to this disk block contains the address of the second disk block, etc. The FAT element corresponding to the last disk block contains a special code to indicate end of file. Figure 7.14 illustrates the FAT for the disk of Fig. 7.13. The file alpha consists of disk blocks 3 and 2. Hence the directory entry of alpha

contains 3. The FAT entry for disk block 3 contains 2, and the FAT entry for disk block 2 indicates that it is the last block for the file. The file **beta** consists of blocks 4, 5 and 7.

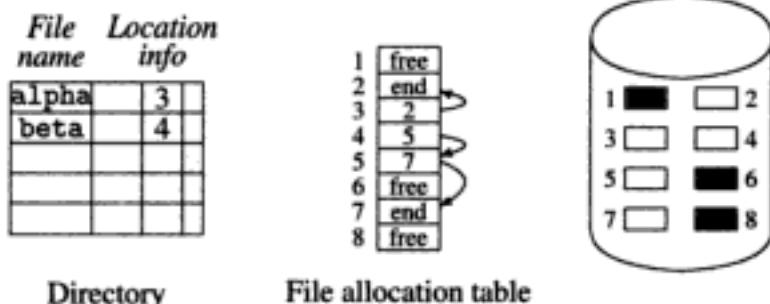


Fig. 7.14 File allocation table (FAT)

An FAT can also be used to store free space information. The list of free disk blocks can be stored as if it were a file, and the address of the first free disk block can be held in a free list pointer. Alternatively, some special code can be stored in the FAT element corresponding to a free disk block, e.g. the code 'free' in Figure 7.14.

Use of an FAT rather than the classical linked allocation involves a performance penalty, since the FAT has to be accessed to obtain the address of the next disk block. To overcome this problem, the FAT is held in memory during file processing. Use of an FAT provides higher reliability than classical linked allocation because corruption of a disk block containing file data leads to limited damage. However, corruption of a disk block used to store an FAT is disastrous.

Indexed allocation In indexed allocation an index called the *file map Table* (FMT) is maintained to note addresses of disk blocks allocated to a file. In its simplest form, FMT can be an array containing disk block addresses. Each disk block contains a single field—the data field. The *loc info* field of a directory entry points to the FMT for a file (see Figure 7.15). In the following discussion we denote the FMT of file **alpha** as fmt_{alpha} .

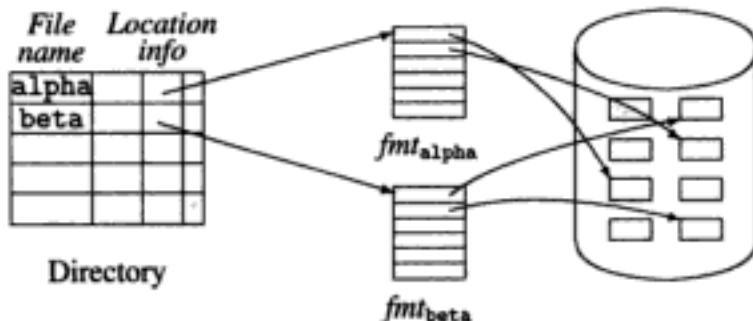


Fig. 7.15 Indexed allocation of disk space

A table called the *disk status map* (DSM) is used to indicate status of disk blocks.

DSM has one entry for each disk block that indicates whether the disk block is free or has been allocated to a file. This information can be maintained in a single bit. Figure 7.16 illustrates a DSM. A '1' in an entry indicates that the corresponding disk block is allocated. DSM is consulted every time a new disk block has to be allocated to a file. An alternative to the use of a DSM is to use a free list of disk blocks as described earlier.

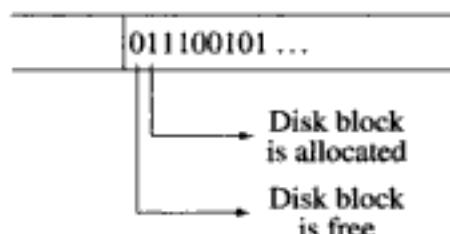


Fig. 7.16 Disk status map (DSM)

Allocation of disk space to a file named *alpha* is performed on demand when *alpha* is created or updated. DSM is searched to locate a free block, and the address of the block is added to fmt_{alpha} . Deallocation is performed when *alpha* is deleted. All disk blocks pointed to by fmt_{alpha} are marked free before fmt_{alpha} and the directory entry of *alpha* are erased.

The reliability problem is less severe in indexed allocation than in linked allocation. This is due to the fact that corruption of an entry in an FMT or DSM leads to limited damage. Compared to linked allocation, access to sequential files is less efficient since the FMT of a file has to be accessed to obtain the address of the next disk block. However, access to records in a direct file is more efficient since the address of the disk block that contains a specific record can be obtained directly from the FMT.

For a small file, the FMT can be stored in the directory entry of the file. This is both convenient and efficient. For a medium or large file the FMT would not fit into the directory entry. The multi-level indexed allocation depicted in Figure 7.17 is popularly used to such FMTs. The directory still contains a part of the FMT. The first few entries in the FMT, say n entries, point to data blocks as in the conventional indexed allocation. Other entries point to special blocks called *index blocks*. An index block does not contain data, it contains pointers to data blocks. Such data blocks need to be accessed through two levels of indirection. The first level of indirection points to an index block. The second level of indirection points to a data block. The advantage of this arrangement is that small files containing n or fewer data blocks continue to be accessible very efficiently as their FMT does not contain index blocks. Medium and large files suffer a marginal degradation of their access performance due to multiple levels of indirection.

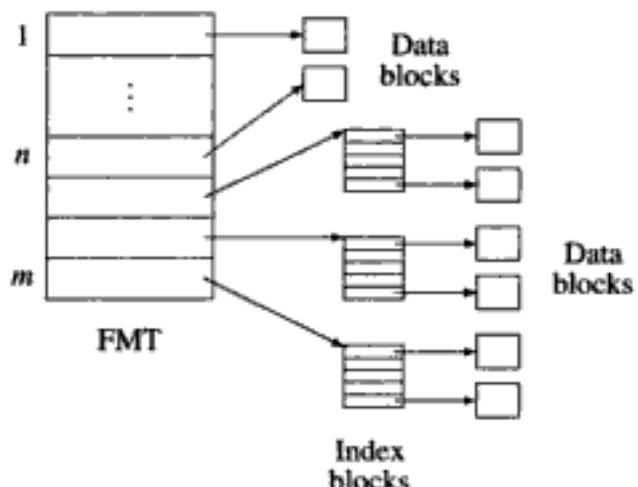


Fig. 7.17 Multi-level indexed allocation

7.8 IMPLEMENTING FILE ACCESS

Use of the directory structure and the file control block (FCB) in implementing a file access has already been discussed in Section 7.6. The scheme discussed there uses an FCB to keep track of the current state of a file processing activity. Address of an FCB is passed as a parameter in every call on a file system operation made by a process, and in every call on an IOCS operation made by the file system. A weakness of this scheme is that an FCB is a part of the address space of a process, which raises a reliability concern because the process could tamper with information in the FCB, thereby affecting operation of the file system. This concern can be addressed by constructing FCBs in the system area. The kernel creates an FCB in its own address space when a process opens a file, and passes its address to the process. The process cannot tamper with the FCB due to memory protection.

A data structure called the *active files table* (AFT) is used to hold FCBs of all open files. When a file is opened, the file system stores its FCB in one entry of the AFT. The offset of this entry in the AFT is called the *internal id* of the file. The internal id is passed back to the process, which uses it as a parameter in all future file system calls. Figure 7.18 shows this arrangement. When a file *alpha* is opened $internal\ id_{alpha}$, which is 6, is passed back to the process.

7.8.1 File System Actions at Open

The purpose of a call `open (<pathname> ..)`, where `<pathname>` is an absolute or relative path name for a file `<filename>`, is to set up the processing of the file. `open` performs the following actions:

1. An FCB for the file `<filename>` is created in the AFT.
2. Internal id of the file `<filename>` is passed back to the process for use in file processing.

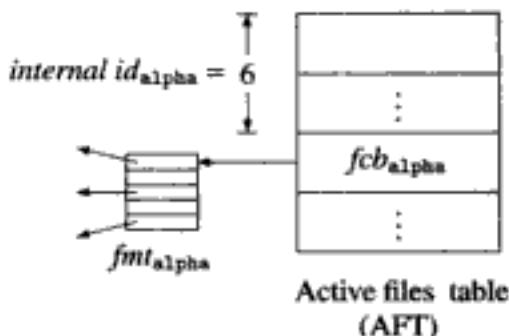


Fig. 7.18 Active files table (AFT)

3. If the file $\langle\text{filename}\rangle$ is being created or updated, provision is made to update its directory entry when a `close` call is made by the process.

To implement action 3, the file system sets up a pointer called the *directory FCB pointer*. This pointer points to the FCB of the directory, which contains an entry for $\langle\text{filename}\rangle$. This directory should be updated when the file is closed, so the file system stores *directory FCB pointer* in FCB of $\langle\text{filename}\rangle$.

These actions are implemented by a procedure called *path name resolution*, which traverses all path components in a path name and checks validity of each component. Path name resolution consists of the following steps:

1. If an absolute path name is used, locate the FCB of the file system root directory in the AFT; otherwise, locate the FCB of the current directory. (This step assumes that the FCBs of these directories have already been created in AFT. If not, they should be created in this step.) Set *directory FCB pointer* to point at this FCB.
2. (a) Search for the next path component in the path name in the directory represented by *directory FCB pointer*. Give an error if the component does not exist or if it is invalid in this directory.
 (b) Create an FCB for the file described by the path component. Store this FCB in a free entry of AFT.
 (c) Set a pointer called *file FCB pointer* to point at this FCB.
 (d) If this is not the last component in the path name, initialize the newly created FCB using information from the directory entry of the file. Set *directory FCB pointer* = *file FCB pointer*, and repeat step 2.
3. (a) If the file already exists, initialize the FCB pointed to by *file FCB pointer* using information from the directory entry of the file. This action includes copying the pointer to the FMT of the file.
 (b) If the file does not already exist, format the FMT of the file. (This action may involve allocating a disk block for the FMT and storing its address in the FCB.)
4. Set *internal id* of the file to the offset of *file FCB pointer* in AFT. Copy the

directory FCB pointer into the FCB of the file. Return *internal id* to the process.

Apart from the actions described above, the file system may perform some other actions in the interest of efficiency. For example, while opening an existing file it may copy part or whole of the file's FMT into memory (see Step 3(a)). This action ensures efficient access to data in the file. Also, only the FCBs pointed to by *directory FCB pointer* and *file FCB pointer* are needed during file processing, so other FCBs created during path name resolution may be destroyed.

Example 7.8 illustrates the data structures built by the file system when a file is opened.

Example 7.8 Figure 7.19 shows the result of the file system actions after executing the call

```
open (/info/alpha, ...);
```

The path name mentioned in the open call is an absolute path name. The file system searches for the name *info* in the root directory, and creates an FCB for *info* in the AFT. It now searches for the name *alpha* in *info* and creates an FCB for *alpha* in the AFT. *directory FCB pointer* points to *fcb_{info}*, and *file FCB pointer* points to *fcb_{alpha}*. Since *alpha* is an existing file, its FMT pointer is copied into *fcb_{alpha}* from the directory entry of *alpha*. The call returns with the internal id of *alpha*, which is 6.

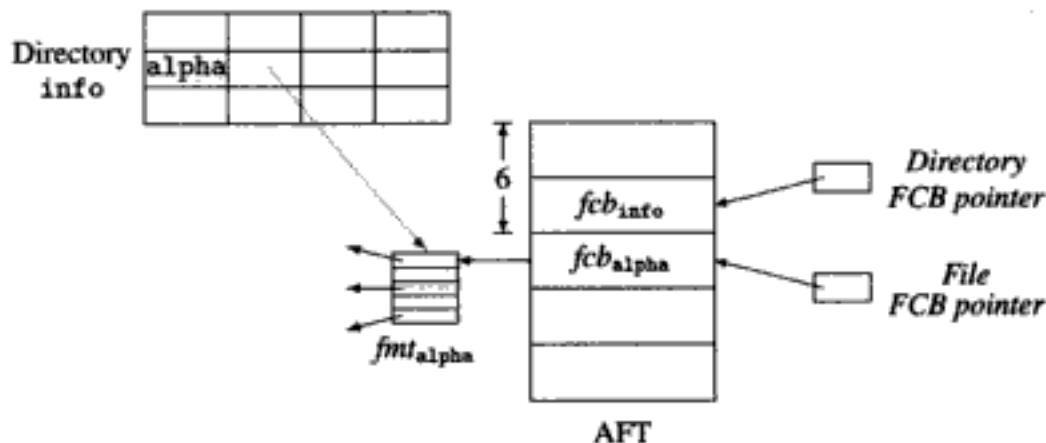


Fig. 7.19 FS actions at open

Accessing mounted files A file system is mounted at a mount point by using the command *mount (<FS_name>, <mount_point_name>)* (see Section 7.4). A simple way to implement mounting is to temporarily change the directory entry of *<mount_point_name>* in its parent directory to point at the directory entry of *<FS_name>*.

The file system has to pay special attention to the crossing of a mount point during path name resolution. For this purpose, it puts the value 'M' in the *flags* field

of the directory entry of $\langle FS_name \rangle$ and maintains a *mount table* to store pairs of the form ($\langle FS_name \rangle$, $\langle mount_point_name \rangle$). During path name resolution, this table is consulted when a mount point is encountered while traversing the directory hierarchy from parent to child (for '/' operator in the path name) or child to parent (for the '..' operator). The file system also has to ensure that disk space allocation performed while processing a mounted file should be in the mounted file system rather than in the host file system.

Example 7.9 When the call `mount (meeting, -A/admin)` of Section 7.4 is executed, the file system changes the directory entry of `admin` to point at `meeting`. The changed directory hierarchy now resembles the hierarchy shown in Figure 7.11(b). The directory entry of `meeting` contains 'M' in its *flags* field, and the mount table contains the pair (`meeting`, `-A/admin`). This information is adequate for path name resolution.

7.8.2 File System Actions at a File Operation

After opening a file $\langle filename \rangle$, a process initiated by user U performs some read or write operations on it. Each such operation is translated into a call

$\langle opn \rangle$ (*internal id*, *record id*, $\langle IO_area\ addr \rangle$);

where *internal id* is the internal id of $\langle filename \rangle$ returned by the `open` call. (Note that *record id* may be absent in the case of a sequential file as an operation is always performed on the next record.) The file system performs the following actions to process this call:

1. Locate the FCB of $\langle filename \rangle$ in the AFT using *internal id*.
2. Search the access control list of $\langle filename \rangle$ for the pair (U, ...). Give an error if $\langle opn \rangle$ does not exist in the list of access privileges for U. This step requires a reference to the access control list of $\langle filename \rangle$. This reference can be performed using the *directory FCB pointer*. Alternatively, the access control list can be copied into the FCB when the file is opened. This way the directory entry of the file need not be accessed for every file operation.
3. Make a call on `iocs-read` or `iocs-write` with the parameters *internal id*, *record id* and $\langle IO_area\ addr \rangle$.

The IOCS module `iocs-read` is called for a read operation. It obtains the FMT from FCB of the file, and converts *record id* into a pair (*disk block id*, *byte offset*) using the FMT. If execution of operation `iocs-write` requires a new record to be written into the file, the IOCS module may have to call an appropriate module of the file system that would allocate a new disk block and add its address to the FMT.

Example 7.10 Following the `open` call of Example 7.8, a statement `read (alpha, 25, ...)` in the process, where 25 is *record id*, would be translated into `iocs-read (6, 25)`. If disk blocks have a size of 1000 bytes each, and a record is 100 bytes in

length, IOCS would convert *record id* into disk block number = 3 and record number in disk block = 5. This gives byte offset = 400. Disk block id of the third disk block allocated to alpha is obtained from the FMT and this block is read to obtain the desired record.

7.8.3 File System Actions at Close

The file system performs the following actions when a process executes the statement `close (internal id, ...)`.

1. If the file has been newly created or updated,
 - (a) If a newly created file, use *directory FCB pointer* to locate the FCB of the directory in which the file is to exist. Create an entry for the file in this directory. Copy the FMT of the new file or its pointer, as the case may be, into this entry. If the directory entry contains a pointer to FMT rather than the FMT itself, the FMT is first written into a disk block and the address of the disk block is entered in the directory entry.
 - (b) If the file has been updated and its size has changed, the directory entry of the file is updated using *directory FCB pointer*.
 - (c) If necessary, repeat steps 1(b)–1(c) to update other directories in the path name after setting *file FCB pointer* := *directory FCB pointer* and *directory FCB pointer* := Address (FCB of the parent of the Directory).
2. The FCB of the file and FCB's of its parent and ancestor directories are erased from the AFT.

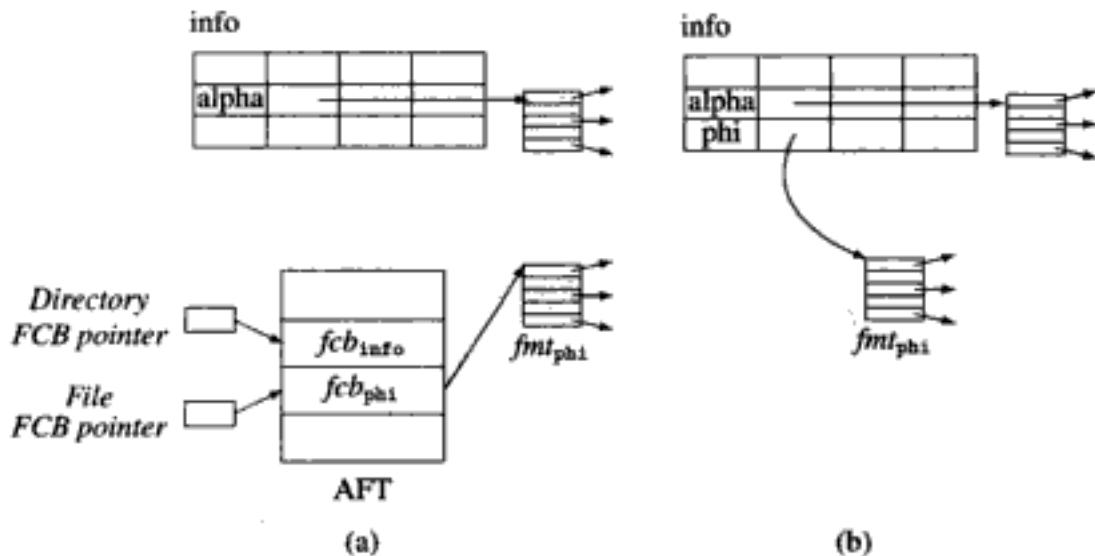


Fig. 7.20 FS actions at close

Example 7.11 Figure 7.20 illustrates the file system actions before and after executing the command `close /info/phi` for a newly created file *phi*. An entry is created

for ϕ_i in directory info and a pointer to $f\text{mt}_{\phi_i}$ is put in the *Loc info* field of this entry. Addition of this entry to info increases the size of info , hence an additional disk block may have to be allocated to info . This will involve updating the FMT of info (see steps 1(b) and 1(c) of actions at *close*).

7.9 FILE SHARING SEMANTICS

As discussed in Section 7.5, the owner of a file can authorize some other users to access the file. Authorized users can read, write or execute the file in accordance with access privileges granted to them. In essence they *share* the files to which they have access.

Two modes in which file sharing occurs are *sequential sharing* and *concurrent sharing*. Sequential sharing occurs when authorized users access a shared file one after another. Clearly, file modifications made by one user, if any, are visible to users who access the file afterwards. Concurrent sharing occurs when two or more users access a file over the same period of time. *File sharing semantics* determine the manner in which results of file manipulations performed by concurrent users are visible to one another.

Sequential sharing of files can be implemented by adding a lock field to each directory entry. This field is a single bit that can contain the values 'set' or 'reset'. The *open* operation on a file succeeds if the lock field in the file's directory entry has the value 'reset', and sets the lock; otherwise, the *open* operation fails and has to be repeated. A *close* operation always resets the lock. This arrangement ensures that only one process can use the file at any time. Creation and destruction of FCBs is performed as discussed in Section 7.8.

Now consider two processes P_1 and P_2 concurrently sharing a sequential file *alpha*. As seen in Section 7.6, address of the next record to be accessed by a process is contained in an FCB for a file. Hence a primary requirement for avoiding interference between these processes is that the system should create a separate FCB for each process. This arrangement can be set up quite easily. The file system can simply follow the procedure of Section 7.8.1 every time file *alpha* is opened. This procedure would ensure that an FCB is created for each process sharing the file. We denote the FCB of *alpha* created for process P_1 as $fcb_{\text{alpha}}^{P_1}$.

Concurrent sharing of a file may be implemented in one of the following three modes:

- *Concurrent sharing using immutable files*: The file being shared cannot be modified by any process.
- *Concurrent sharing using single image mutable files*: An image is a view of a file. All processes concurrently sharing a file 'see' the same image of the file. Thus modifications made by a process are immediately visible to other processes using the file.
- *Concurrent sharing using multiple image mutable files*: Each process accessing the file has its own image of the file. The file system may maintain many

images of a file, or it may reconcile them in some manner to create a single image when processes close the file. Effectively, many versions of the file may exist at any time and updates made by a user may not be visible in some concurrent processes.

These modes of sharing have different implications for the file system and for the users sharing a file.

Sharing immutable files When file alpha is shared as an immutable file, none of the sharing processes can modify it. This form of sharing has the advantage that sharing processes are independent of one another—the order in which records of alpha are processed by P_1 is immaterial to the execution of P_2 . Creation of an fcb_{alpha} for each sharing process is adequate to implement this form of file sharing.

Two important issues arise when files are mutable. These are:

- Visibility of modifications in alpha made by one process to other processes sharing alpha.
- Interference between processes sharing alpha.

Sharing single image mutable files In single image mutable files a single copy of the file is shared by processes accessing it, so changes made by one process are immediately visible to other processes. To implement this form of sharing, it is essential that a single copy of FMT should be used by all processes sharing the file. Hence it is best to keep a pointer to the FMT, rather than the FMT itself, in an FCB.

Figure 7.21 shows concurrent sharing of file alpha using such an arrangement. Two FCBs denoted as fcb_{alpha}^{P1} and fcb_{alpha}^{P2} exist for alpha, both pointing to the same copy of fmt_{alpha} . Each FCB contains the address of the next record to be processed by a process. If the set of records processed by P_1 and P_2 overlap, their modifications would be visible to one another. Race conditions could also arise in such situations, and updates made by processes may be lost. A typical file system does not provide any protection against this problem; the sharing processes would have to evolve their own synchronization conventions for this purpose. The Unix file system provides single image mutable files. We discuss Unix file sharing semantics in Section 7.12.2.

Sharing multiple image mutable files In multiple image mutable files, many processes can concurrently update alpha. Each updating process creates a new version of alpha, which is distinct from versions created by other concurrent processes. In this scheme, a distinct fmt_{alpha} must exist for each FCB, and it must point to an exclusive copy of the file. This requirement is best implemented by making a copy of alpha (and its FMT) for each process concurrently updating it. α^{P1} represents the copy of alpha made for process P_1 .

Figure 7.22 illustrates the arrangement for implementing multiple image mutable files. Processes P_1 and P_2 are engaged in updating alpha. Processing by P_1 uses fcb_{alpha}^{P1} and fmt_{alpha}^{P1} to access α^{P1} , while processing by P_2 uses fcb_{alpha}^{P2} and fmt_{alpha}^{P2} .

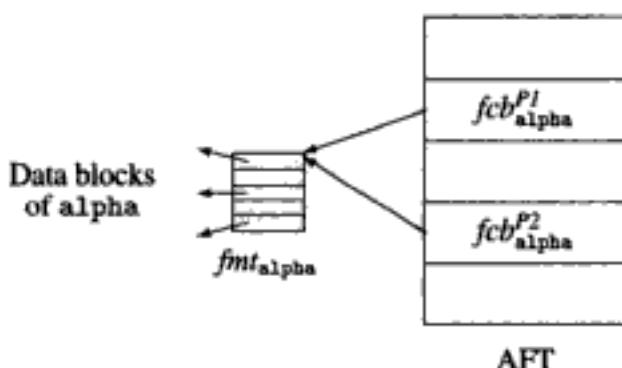


Fig. 7.21 Concurrent sharing of a single image mutable file by processes P_1 and P_2

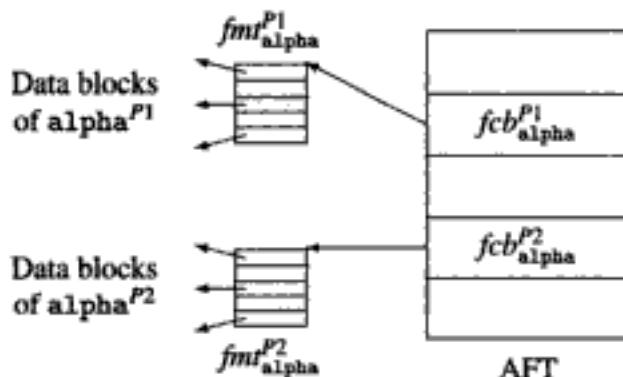


Fig. 7.22 Concurrent sharing of a multiple image mutable file

fmr_{alpha}^{P2} to access α^{P2} . α^{P1} and α^{P2} are thus two versions of α . To arrive at a unique implementation scheme, the file sharing semantics must specify how α would be accessed by processes that only wish to read it, i.e., which version they would access.

Sharing of multiple image mutable files is possible only in applications where existence of multiple versions due to concurrent updates is meaningful. File sharing semantics for multiple image mutable files are very hard to understand and implement. We discuss the *session semantics* used for multiple image mutable files in Section 19.3.

7.10 FILE SYSTEM RELIABILITY

File system reliability concerns ability of a file system to function correctly despite occurrence of faults in the system. Two aspects of file system reliability are:

- Ensuring correctness of file creation, deletion and updates.
- Preventing loss of data in files.

Reliability literature distinguishes between the terms *fault* and *failure*. A fault is a defect in some part of the system. A failure is a system behavior that is erroneous, or

at least different from its expected behavior. Occurrence of a fault causes a failure. Thus corruption of a disk block due to a damaged disk head or a power outage is a fault, whereas inability of the file system to read a block is a failure. In this chapter we shall use the terms fault and failure informally and interchangeably. Chapter 18 uses these terms precisely.

Data corruption in disk blocks and system crashes due to power interruptions are the common failures that lead to reliability problems in file systems. Data corruption causes loss of data in files or loss of file system control data stored on disk. A file system may not be able to continue meaningful operation if the file system control data is either lost or becomes inconsistent. Damage caused by loss of data in a file due to data corruption is comparatively less serious since it is limited to a single file.

7.10.1 Loss of File System Integrity

File system integrity implies correctness and consistency of control data and operations of the file system. Loss of integrity arises if control data of the file system is lost or damaged. It is interesting to see why this happens. To ensure efficient operation, the file system maintains some of its control data in memory. This includes the active files table, which contains file control blocks of open files, parts of the disk status map or free lists of disk blocks, and file map tables of open files. Parts of this data, like the file map table, are written on a disk when a file is closed. In addition, the system may periodically copy its control data, viz. the disk status map and free lists, on the disk. Due to this arrangement, disk copies of control data may not contain up-to-date information during system operation, so control data is lost when power fails. Control data not maintained in memory is lost when a disk crash occurs.

Consider a process that updates a file *alpha*. The following data structures would exist in memory while the process executes: fcb_{α} (which exists in the active files table), part of fmt_{α} and part of the disk status map. This data would be lost if the system crashes. This situation may result in one or more of the following failures:

1. Some data from file *alpha* may be lost.
2. Part of file *alpha* may become inaccessible.
3. Contents of two files may get mixed up.

It is easy to visualize a situation of the first kind. For example, let a failure occur after a new disk block has been added to file *alpha*. The disk copy of fmt_{α} would not contain this block's id, hence data in the newly added block is lost when a failure occurs. Situations of the second and third kind can arise if the file system uses the linked allocation scheme and a failure occurs while a new disk block is being added to a file using Algorithm 7.1.

Algorithm 7.1 (Add block d_j between blocks d_1 and d_2)**Input :**

```

 $d_1, d_2, d_j$  : record
    next : ...; { id of next block }
    data : ...;
end

```

1. $d_j.next := d_1.next;$
2. $d_1.next := address(d_j);$
3. Write d_1 to disk.
4. Write d_j to disk.

Algorithm 7.1 adds a new disk block d_j between blocks d_1 and d_2 of the file. Figure 7.23 illustrates how parts of file alpha may become inaccessible. Figure 7.23(a) shows the file before execution of the algorithm. If a failure occurs between Steps 3 and 4 of Algorithm 7.1, new contents would have been written into disk block d_1 , but not into disk block d_j . Hence $d_1.next$ would point to d_j whereas d_j would not contain correct control information in its *next* field (see Figure 7.23(b)). Disk blocks d_2, d_3, \dots would not be accessible as parts of the file any more.

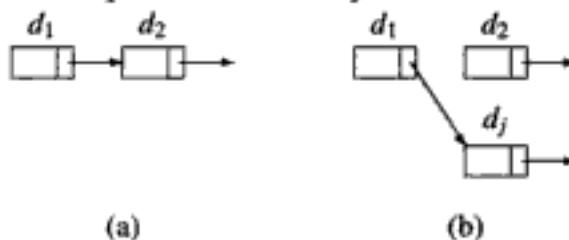


Fig. 7.23 Failure while adding a disk block to a file leads to loss of data

Contents of two files may get mixed up if the file system writes control data to disk only while closing a file, and not after every file operation. Consider the following situation: A process P_1 deletes a disk block d_k from some file *beta*. d_k would be returned to the free list (or would be marked free in the disk status map). Now another process P_2 adds a new record to file *alpha*. The file system allocates a new disk block d_j for this purpose and adds it before disk block d_m in file *alpha* (see Figure 7.24(a)). Now, consider the situation when $d_j = d_k$ and the following events occur in the system:

1. File *alpha* is closed.
2. The file system updates the disk copy of file *alpha* and *fcb_alpha*. This involves adding disk block d_j to *alpha*.
3. A failure occurs.

The disk contains an old copy of *beta*, which contains block d_k , and the new copy of *alpha*, which contains block d_j . Since $d_j = d_k$, *alpha* and *beta* now share disk

block d_j and all other blocks accessible through it (see Figure 7.24(b)). All disk-blocks of file **beta** previously accessible through d_k are now inaccessible. In effect, some data is common to files **alpha** and **beta**, while some data of **beta** has been lost.

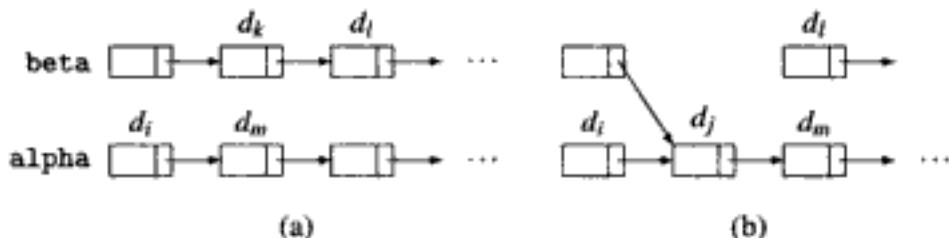


Fig. 7.24 (a) Files alpha and beta, (b) when $d_j = d_k$ and alpha is closed

7.10.2 File System Reliability Techniques

Operating systems use two techniques to ensure that user files are reliably stored over a period of time. *Recovery* is a classic technique which is used when a failure is noticed. It restores the data in files and control data of the file system to some previous consistent state. The file system now resumes its operation from this state. Thus, deviations from correct behavior do occur, but system operation is rectified when deviations are noticed.

Fault tolerance is a technique to guard against loss of integrity of the file system when faults occur. It aims at providing file system operation that is both uninterrupted and correct at all times.

7.10.2.1 Recovery Techniques

The *file system state* at some time instant t_i is the collection of all user and control data in the file system at t_i . A *back-up* of the file system is a recording of the file system state. To support recovery, the file system periodically produces back-ups during its operation. Let t_{lb} represent the time at which the latest back-up was produced. In the event of a failure, say, at time t_f , the file system is restored to the state recorded in its latest back-up. This action recovers all file updates performed prior to t_{lb} . However, updates performed between t_{lb} and t_f are lost, so processes performing these updates must be re-executed following recovery.

Recovery actions in a file system involve two kinds of overhead—overhead of creating back-ups, and overhead of re-processing. The latter is the cost of re-executing a process whose updates were lost. An interesting way to reduce overhead is to use a combination of incremental and full back-ups of a file system. An *incremental back-up* contains copies of only those files and data structures that were modified after the last full or incremental back-up was created. The file system creates full back-ups at large intervals of time, e.g., a few days or a week. Incremental back-ups are created at shorter intervals, e.g., at every file close operation, and are discarded when the

next full back-up is created.

After a crash the system is restored from the latest full back-up. Incremental back-ups are then processed in the same order in which they were created. Thus files whose modification was completed before the failure would be recovered in full. Files that were being modified at the point of failure would not be completely restored. Some reprocessing cost would be incurred to re-execute the processes that were in execution at the time of system failure. The space overhead would also increase because full and incremental back-ups coexist and some files may exist in more than one incremental back-up.

Example 7.12 Figure 7.25 illustrates a system in which full back-ups were taken at time t_1 and t_4 , and incremental back-ups were taken at t_2 and t_3 . Note that the sizes of the incremental back-ups vary depending on the number of files updated since the last full or incremental back-up. If a failure occurs after t_4 , the system would be restored to the state recorded in the back-up taken at t_4 . If a failure occurs between t_3 and t_4 , the system would be first restored to the state recorded in the back-up taken at t_4 . After this is done, the incremental back-ups taken at t_2 and t_3 would be processed in that order. An alternative approach would be to process the incremental and full back-ups in reverse order, that is, process the incremental back-up taken at t_3 , followed by the incremental back-up taken at t_2 , followed by the full back-up taken at t_1 . Care should be taken not to restore a file that has been already restored from a later back-up.

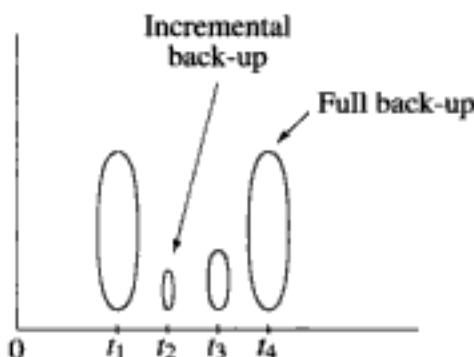


Fig. 7.25 Full and incremental back-ups for recovery

7.10.2.2 Fault Tolerance Techniques

File system reliability can be improved by taking two precautions—prevent loss of data due to device malfunction, and prevent inconsistency of control data due to failures. These precautions are implemented using the fault tolerance techniques of stable storage and atomic actions, respectively.

Stable storage The *stable storage* technique proposed by Lampson provides reliability of a data record in the presence of a single failure. This is achieved using redundancy. Two copies of a record, called its *primary* and *secondary* copy, are maintained on a disk. A write operation updates both copies—the primary copy is

updated first, followed by the secondary copy. For a read operation, the disk block containing the primary copy is accessed. If it is unreadable, the block containing the secondary copy is accessed. Since only single failures are assumed to occur, one of the blocks is sure to contain readable data.

Figure 7.26 illustrates operation of the stable storage technique if failures occur at times t_1, t_2, t_3 , or t_4 , respectively, while a process P is executing an update operation on some data D. Parts (a)–(d) show timing charts and values in the primary and secondary copies of D when failures occur. In Part (a), a failure occurs at time t_1 , i.e., before the primary copy is updated, so the primary copy, which contains the old value of the data, is accessible after a failure. In Part (b), a failure occurs while updating the primary copy, making it unreadable. The old value of data is accessible from the secondary copy. In Part (c), failure occurs after updating the primary copy but before updating the secondary copy. New data is accessible in the primary copy after the failure. In Part (d), failure occurs after both copies have been updated, so both copies remain accessible.

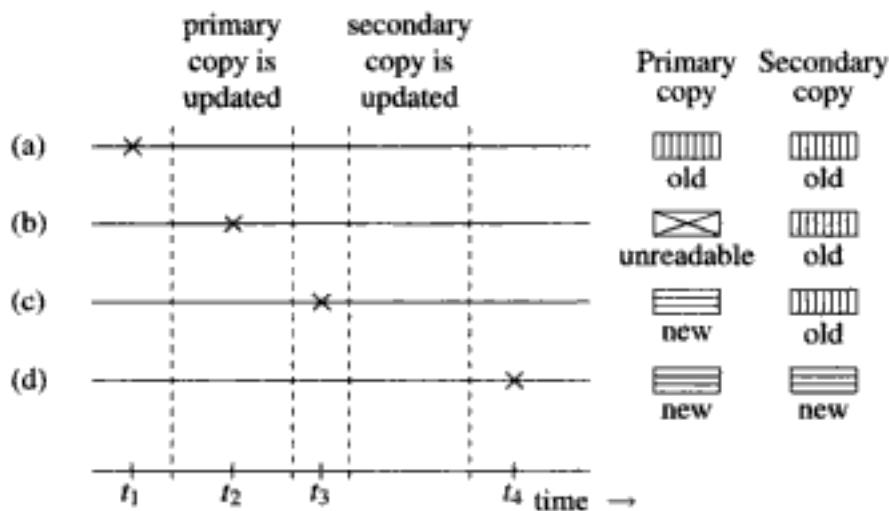


Fig. 7.26 Fault tolerance using the stable storage technique

The stable storage technique can be applied to entire files. Lampson called this technique disk mirroring (however it is different from the disk mirroring technique used in RAID—see Section 12.3). Disk mirroring is very expensive for general use in a file system; processes may selectively use it to protect their data. Also, while disk mirroring guarantees that one copy of data will survive a single failure, it cannot indicate whether this value is old or new (see Parts (a), (d) of Figure 7.26), so the user does not know whether to re-execute the update operation in P when system operation is restored. An atomic action overcomes this problem.

Atomic actions An action may involve manipulation of many data structures. The data structures may become inconsistent if a failure interrupts its execution. For

example, consider a banking application that transfers funds from one account to another. If a failure interrupts its execution, funds may have been debited from one account but not credited to the other account, or vice versa. Atomic actions aim at avoiding such ill-effects of failures.

Definition 7.1 (Atomic action) An action A_i consisting of a set of sub-actions $\{a_{ij}\}$ is an atomic action if for every execution of A_i either

1. Effects of all sub-actions in $\{a_{ij}\}$ are realized, or
2. Effects of none of the sub-actions are realized.

An atomic action A_i has an *all-or-nothing* property. This property avoids data inconsistency when failures occur. Figure 7.27 shows Algorithm 7.1 coded as an atomic action named *add_a_block*. It differs from Algorithm 7.1 only in the use of the statements **begin atomic action** and **end atomic action**. The execution of an atomic action begins with the **begin atomic action** statement. It can end in two ways—it can either fail or succeed. An atomic action fails if an **abort** statement is executed, or if a failure occurs before the statement **end atomic action** is executed. If it fails, the state of each file and each variable used by the atomic action should be as it was prior to execution of the **begin atomic action** statement. An atomic action succeeds when it executes the **end atomic action** statement. It is said to *commit* at this point. All updates made by the atomic action are guaranteed to survive any failures after it commits.

```
begin atomic action add_a_block;
    dj.next := d1.next;
    d1.next := address(dj);
    write d1;
    write dj;
end atomic action add_a_block;
```

Fig. 7.27 Atomic action *add_a_block*

We thus have the guarantee that execution of the atomic action *add_a_block* of Figure 7.27 would lead to one of two possibilities. If the atomic action commits, disk block d_j is added to file *alpha*, which now consists of disk blocks $\dots d_1, d_j, d_2, \dots$. If the atomic action fails, disk block d_j is not added to file *alpha*, i.e., *alpha* continues to consist of disk blocks $\dots d_1, d_2, \dots$. Consistency of the file system control data can be preserved by updating all file system data structures using atomic actions. Data base systems use atomic actions to ensure certain other properties as well; this discussion is restricted to file system reliability only.

Atomic actions can be implemented in several ways. In one implementation approach, files are not updated during execution of the atomic action. Instead details of updates to be made are noted in a list that we will call the *intentions list*. The intentions list contains pairs of the form (*<disk block id>*, *<new contents>*) indicating that *<new contents>* should be written in the disk block with the id *<disk block*

id>. Information in the intentions list is used to update the files when the atomic action commits. This is called *commit processing*.

This arrangement automatically tolerates failures that occur before an atomic action commits since no updates would have been made in the files. Thus it implements the ‘nothing’ part of the all-or-nothing property. While making updates, it is necessary to ensure that all updates would be completed even if failures occur during commit processing. This feature of commit processing would implement the ‘all’ part of the all-or-nothing property.

A *commit flag* is associated with each atomic action. It contains two fields *transaction id* and *value*. This flag is created when the statement **begin atomic action** of an atomic action A_i is executed, and its fields are initialized to A_i and ‘not committed’. The commit flag is changed to ‘committed’ when **end atomic action** is executed. The flag is destroyed after all updates have been carried out.

Both the intentions list and the commit flag are maintained in stable storage to protect them against data corruption and loss due to a failure. When a system recovers after a failure, it checks for presence of commit flags. If a commit flag exists for A_i and has the value ‘not committed’, it is simply destroyed and atomic action A_i is executed again starting with the statement **begin atomic action**. Existence of a commit flag for A_i with the value ‘committed’ implies that commit processing of A_i was in progress when the failure occurred. Since it is not known if any entries of the intentions list were processed before the failure, the entire commit processing is now repeated. Note that in this arrangement an update may be carried out more than once if a failure occurs during commit processing.

To ensure that repeated processing of an intentions list does not pose any problems of data consistency, an update operation must be *idempotent*. That is, executing the update operation more than once must give the same result as executing it once. If this were not the case, files may contain different values if an update operation is repeated due to a failure during commit processing. (Imagine somebody’s age being incremented by more than one on 31 December!). Writing *<new contents>* in the disk block with the id *<disk block id>* is an idempotent operation, so problems concerning data consistency do not arise even if failures occur during commit processing. Algorithm 7.2 summarizes all actions concerning implementation of an atomic action.

Algorithm 7.2 (Implementation of an atomic action)

1. *Execution of an atomic action:*

- (a) Perform the following actions when the statement **begin atomic action** of atomic action A_i is executed:

commit flag := (A_i , ‘not committed’);

intentions list := ‘empty’;

Both *commit flag* and *intentions list* are maintained in stable storage.

- (b) For every file update made by a sub-action, add a pair (d, v) to the intentions list, where d is a disk block id and v are its new contents.
- (c) When the statement **end atomic action** is executed, set value of A_i 's *commit flag* to 'committed' and perform Step 2.
2. *Commit processing*:
 - (a) For every pair (d, v) in the intentions list, write v in the disk block with the id d .
 - (b) Erase the commit flag and the intentions list.
3. *On recovering after a failure*:
- If the commit flag for atomic action A_i exists,
- (a) If the value in commit flag is 'not committed': Erase the commit flag and the intentions list. Re-execute atomic action A_i .
 - (b) Perform Step 2 if value in commit flag is 'committed'.

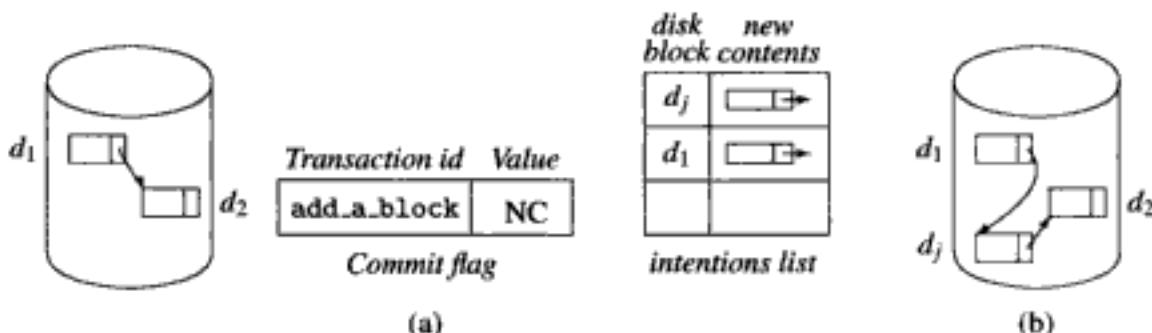


Fig. 7.28 (a) Before, and (b) after commit processing (note: NC implies *not committed*)

Example 7.13 Figure 7.28(a) shows file alpha, the commit flag and the intentions list when Algorithm 7.2 is applied to the atomic action of Figure 7.27. New contents of disk blocks d_j and d_1 are kept in the intentions list until commit processing. Atomicity of the action is ensured as follows: If a failure occurs during Step 1 of the algorithm, none of the file updates are reflected on the disk. Hence the file contains the original sequence of disk blocks d_1, d_2, \dots . A failure in Step 2 can corrupt some data, however the intentions list and the commit flag are not affected since they exist in stable storage. Failures in Step 2 lead to repeated commit processing. This does not interfere with data consistency since new contents are simply written into disk blocks during commit processing. Thus, as shown in Figure 7.28(b), the file contains the sequence of disk blocks $d_1, d_j, d_2 \dots$ at the end of commit processing.

7.11 VIRTUAL FILE SYSTEM

Users have diverse requirements of a file system, such as convenience, high reliability, fast response and access to files on other computer systems. A single file system cannot provide all these features, so an operating system provides a *virtual file sys-*

tem (VFS), which facilitates simultaneous operation of several file systems. This way each user gets to use the file system he prefers.

A virtual file system (VFS) is an abstraction that supports a generic file model. The abstraction is implemented by a VFS layer that is situated between a process and a file system (see Figure 7.29). The VFS layer has two interfaces—an interface with the file systems, and an interface with processes. Any file system that conforms to the specification of the VFS–file system interface can be installed to work under it. This feature makes it easy to add a new file system. The VFS–process interface provides functionalities to perform generic open, close, read and write operations on files, and mount, unmount operations on file systems. These functionalities are invoked through system calls. The VFS determines which file system a file actually belongs to and invokes the open, close, read and write functionalities of the specific file system through the VFS–file system interface. It also invokes functions of the specific file system to implement mount and unmount operations.

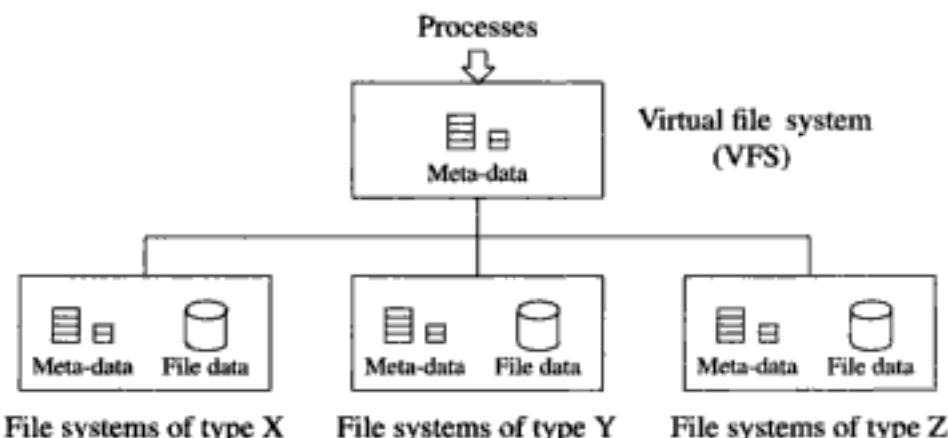


Fig. 7.29 Virtual file system

All file systems operating under the VFS are available for use simultaneously. In the system of Fig. 7.29, one process may use a file system of type X while another process simultaneously uses a file system of type Y. The virtual file system can also be used to compose a heterogeneous file system. For example, a user can mount a file system of type X in a directory of a file system of type Y. This feature is useful with removable media like CDs; it permits a user to mount the file system that exists in a CD in his current directory and access its files without any concern for the fact that file data is recorded in a different format. This feature is also useful in a distributed environment for mounting a remote file system into a file system of a different type. It is described in Section 19.6.1.

As shown in the schematic diagram of Figure 7.29, the virtual file system does not contain any file data. It merely contains data structures that constitute VFS meta-data. Each file system contains its own meta-data and file data. The key data structure used by the virtual file system is the virtual node, popularly called *vnode*. The *vnode*

is a representation of a file in the kernel. It can be looked upon as a file object with the following three parts:

- File system independent data such as a file id that is unique within the domain of the VFS, which may be the individual computer system or a network; the file type, e.g., directory, data file, or a special file; and other fields such as an open count, lock and flags.
- File system specific data such as the file map table.
- Addresses of functions in the file system which contains this file. These functions implement the open, close, read and write operations on files of this file type.

Operating systems have provided virtual file systems since the 1990's. Sun OS, Unix System V version 4, Unix 4.2 BSD, and Linux provide a virtual file system.

7.12 UNIX FILE SYSTEM

Versions of Unix differ in features of the file system. In this section we describe important features common to many Unix versions. The file system of Unix is greatly influenced by the Multics file system. The directory structure of Unix is analogous to the directory structure discussed in Section 7.4. Directory hierarchies are formed by treating directories themselves as files. For protection purposes, three user groups are defined as discussed in Example 7.7. Unix supports only byte-stream files, i.e., files without any structure. Files are considered to be streams of characters and are accessed sequentially. The generic arrangement concerning file processing described in Section 7.4 and 7.6 centers around the use of directory entries, FCBs and internal ids. The arrangement used in Unix centers around the following data structures:

- Inodes (a short form of *index nodes*),
- File descriptors
- File structures

7.12.1 Inodes, File Descriptors and File Structures

Information in the directory entry of Section 7.4 is split between the directory entry and the inode of the file. A directory entry contains only the file name. Bulk of the information concerning a file is contained in the inode. The directory in System V contains entries of 16 bytes each. The first two bytes contain the inode number, while the remaining bytes contain the file name, which can be upto 14 bytes long. In Unix version 4.2, the file name can be upto 255 bytes long. The file structure contains two fields—current position in an open file, which is in the form of an offset from start of the file, and a pointer to the inode for the file. A file descriptor points to a file structure. Its use resembles that of the internal id of a file in the generic arrangement of Sections 7.4 and 7.6. A directory lookup cache holds information concerning a

few files on an LRU basis. This cache is searched at file open time. A successful search avoids expensive directory lookups.

The inode data structure is maintained on disk. Some of its fields contain the following information:

- File type, whether directory, link, special file, etc.
- Number of links to the file
- File size
- id of the device on which the file is stored
- Inode serial number
- User and group id's of the owner
- Access permissions
- Allocation information.

The allocation information is analogous to the FMT described in Section 7.7. Note the similarity between fields of the inode and those of the FCB (see Table 7.5). The system administrator can specify a disk quota for each user to prevent a user from occupying too much disk space.

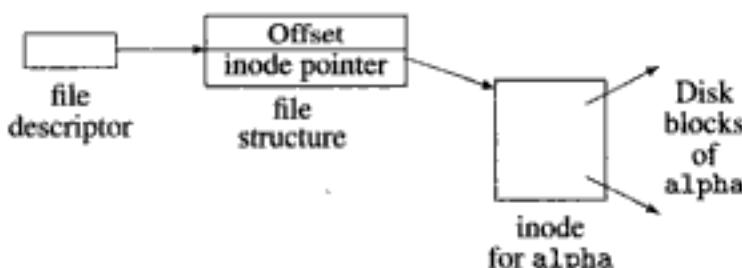


Fig. 7.30 Unix data structures

Figure 7.30 illustrates the arrangement of file descriptor, file structure and inode in memory during the processing of a file. When a process opens a file, a file descriptor and a file structure is created for it and the file descriptor is set to point to the file structure. The file structure contains two fields—offset of the next byte in the file, and the inode pointer. The inode of the file is copied into the memory, unless the memory already contains such a copy, and the file structure is made to point to the inode. Thus the inode and the file structure together contain all information necessary to access a file.

File descriptors are stored in a table of open files. This table resembles the active files table (AFT) described in earlier sections, with one difference. It is a per process table rather than a global table. The file descriptor points to the file structure. When a file is opened, the file descriptor is passed to the process that opened the file. When a process creates a child process, a table of descriptors is created for the child process

and file descriptors of the parent process are copied into it. Thus many file descriptors may share the same file structure. Processes owning the descriptors share the file offset.

7.12.2 File Sharing Semantics

Unix provides single image mutable file semantics for concurrent file sharing. As shown in Figure 7.30, every process that opens a file points to the copy of its inode through the file descriptor and file structure. Thus, all processes sharing a file use the same copy of the file; changes made by one process are immediately visible to other processes sharing the file. To avoid race conditions while accessing an inode, a lock field is provided in the memory copy of an inode. A process trying to access an inode must sleep if the lock is set by some other process. Processes concurrently using a file must make their own arrangements to avoid race conditions on data contained in the file.

7.12.3 Disk Space Allocation

Unix uses indexed disk space allocation, with a disk block size of 4 K bytes. Each file has a *file allocation table* analogous to FMT, which is maintained in its inode. The allocation table contains 15 entries (see Figure 7.31). Twelve of these entries directly point to data blocks of the file. The next entry in the allocation table points to an indirect block, i.e., a block that itself contains pointers to data blocks. The next two entries point to double and triple indirect blocks. In this manner, the total file size can be as large as 2^{42} bytes. However, the file size information is stored in a 32-bit word of the inode, so file size is limited to 2^{32} 1 bytes, for which the direct, single and double indirect blocks of the allocation table are adequate.

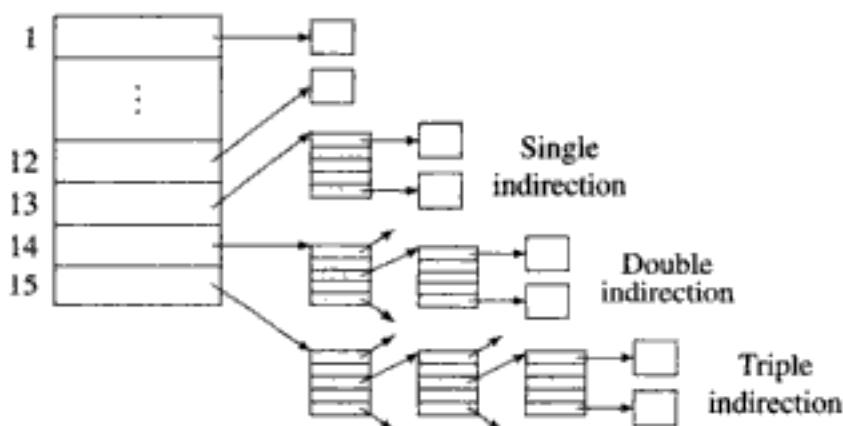


Fig. 7.31 Unix file allocation table

For file sizes smaller than 48 K bytes, this arrangement is as efficient as the flat FMT arrangement discussed in Section 7.7. Such files also have a small allocation table that can fit into the inode itself. The indirect blocks permit files to grow to

very large sizes, although their access involves traversing the indirection in the file allocation table.

The file system maintains a free list of disk blocks. An interesting technique is employed to store the free list. In its simplest form, a free list would have been stored using an arrangement similar to linked allocation—the list would consist of a set of free disk blocks, with each block pointing to the next block in the list. However, Unix uses an indexed allocation scheme, so each disk block that was allocated to a deleted file would have to be individually linked to the free list. To avoid this, the Unix free list consists of a list of blocks in which each block is similar to an indirect block in FMT. Thus each block in the free list contains addresses of free disk blocks, and the id of the next block in the free list. This arrangement minimizes the overhead of adding disk blocks to the free list when a file is deleted. Only marginal processing is required for files smaller than 48 Kbytes in size.

A lock field is associated with the free list to avoid race conditions when disk blocks are added and deleted from it. A file system program named `mkfs` is used to form the free list when a new file system is created. `mkfs` lists the free blocks in ascending order by block number while forming the free list. This ordering is lost as disk blocks are added to and deleted from the free list during the file system operation. The file system makes no effort to restore this order. Thus blocks allocated to a file may be haphazardly distributed on a disk, which reduces the access efficiency of a file. To address this issue, BSD Unix uses the concept of *cylinder groups*. A cylinder group is a set of consecutive cylinders on a disk. To minimize disk head movement, the file system ensures that all disk blocks allocated to a file belong to the same cylinder group.

7.12.4 Multiple File Systems

Unix maintains the root of every file system, called the *super block*, in main memory in the interest of efficiency. Inodes of all open files are also copied into the memory. The super block contains the size of the file system, the free list and the size of the inode list. The super block is copied onto the disk periodically. Some part of the file system state is lost if the system crashes after the superblock is modified but before it is copied to the disk. The file system can reconstruct some of the lost state information, e.g., the free list, by analyzing the disk status. This is done as a part of the system booting procedure.

Many file systems can exist in a Unix system. Each file system consists of a super block, an inode list and data blocks. A file system exists on a single logical disk device, hence files cannot span across different logical disks. A logical disk contains exactly one file system. A physical disk can be partitioned into many logical disks and a file system can be constructed on each of them. This partitioning provides some protection, and also prevents a file system from occupying too much disk space. A file system has to be mounted before being accessed. Only a super user, typically the system administrator, can mount a file system.

Mounting and unmounting of file systems works as follows: A logical disk containing a file system is given a device special file name. A file system can be mounted in any directory by specifying its path name in a `mount` command, e.g.,

```
mount (<FS_name>, <mount_point_name>, ...);
```

where `<FS_name>` is the device special file name of the file system to be mounted, and `<mount_point_name>` is the path name of the directory in which the file system is to be mounted. Once mounted, the root of the file system has the name given by the path name specified for `<mount_point_name>`. Other details of mounting are as described in Section 7.4. When a file system is mounted, the superblock of the mounted file system is loaded in memory. Disk block allocation for a file in the mounted file system must now be performed within the logical disk device on which the file system exists. Files in a mounted file system are accessed as described in Section 7.8.1.

A file open call in Unix specifies three parameters—path name, flags and mode. Flags indicate the kind of operations for which the file is being opened. The possible values of this flag are `read`, `write` or `read/write`. The mode parameter is provided when a new file is being created. It specifies the access privileges to be associated with the file. This information is typically copied from the file creation mask of the user. The owner of a file can change the file protection information any time through a `chmod` command.

7.12.5 Other Features

Two kinds of links are supported in Unix. Links as described in Section 7.4 are called *hard links*. The file system maintains a reference count with each file to indicate the number of links pointing to it. A file can be deleted only if its reference count becomes 0. Thus, if a file has many links to it, the file may not be deleted even if its owner performs a delete operation. In fact, the file would be included in the owner's quota and the owner would continue to be charged for the disk space occupied by it! A *symbolic link* is a file that contains a pathname of another file. Thus if a file `alpha` is created as a symbolic link to file `beta`, file `alpha` is created and the path name of `beta` provided in the `link` command is stored in it. The directory entry of `alpha` is marked as a symbolic link. This way the file system knows how to interpret its contents. The reference count of `beta` is not incremented when a symbolic link is set up. Thus, `beta`'s deletion remains under control of its owner. However, symbolic links can create dangling references when files are deleted.

Buffers are not allocated at the level of a file or a process. All buffers and their contents are shared across processes. This arrangement helps to implement single image mutable file semantics (see Section 7.12.2). It also reduces the number of disk accesses when a file is processed concurrently by two or more processes (see Section 12.10).

7.13 LINUX FILE SYSTEM

Linux provides a virtual file system (VFS) which supports a common file model that resembles the Unix file model. This file model is implemented using Unix-like data structures such as superblocks and inodes. When a file is opened, the VFS transforms its directory entry into a dentry object. This dentry object is cached so that the overhead of building it from the directory entry is avoided if the file is opened repeatedly during a computing session. The standard file system of Linux is called ext2. The file system ext3 incorporates journaling, which provides integrity of file data and meta-data and fast booting after an unclean shutdown.

Ext2 provides a variety of file locks for process synchronization. *Advisory* locks are those that are supposed to be heeded by processes to ensure mutual exclusion; however, the file system does not enforce their use. Unix file locks belong to this category of locks. *Mandatory* locks are those that are checked by the file system; if a process tries to access data that is protected by a mandatory lock, the process is blocked until the lock is reset by its holder. A *lease* is a special kind of file lock which is valid for a specific amount of time after another process tries to access the data protected by it. It is implemented as follows: If a process accesses some data that is protected by a lease, the holder of the lease is intimated by the file system. It now has a stipulated interval of time to finish accessing the file and release the lease. If it does not do so, its lease is broken and awarded to the process that tried to access the data protected by it.

Design of ext2 was influenced by Unix BSD's fast file system. Ext2 uses the notion of a *block group*, which is a set of consecutive disk blocks, to reduce the movement of disk heads when a file is opened and its data is accessed. It uses a bitmap to keep track of free disk blocks in a block group. When a file is created, it tries to allocate disk space for the inode of the file within the same block group that contains its parent directory, and also accommodates the file data within the same block group. Every time a file is extended through addition of new data, it searches the bitmap of the block group to find a free disk block that is close to a target disk block. If such a disk block is found, it checks whether a few adjoining disk blocks are also free and preallocates a few of these to the file. If such a free disk block is not found, it preallocates a few contiguous disk blocks located elsewhere in the block group. This way it is possible to read large sections of data without having to move the disk head. When the file is closed, preallocated but unused disk blocks are freed. This strategy of disk space allocation ensures use of contiguous disk blocks for contiguous sections of file data even when files are created and deleted at a high rate; it contributes to high file access performance.

7.14 WINDOWS FILE SYSTEM

NTFS is the file system used in Windows. NTFS is designed to meet the file system requirements for servers and workstations, so it provides support for high-end appli-

cations like network applications for large systems of corporate entities and client-server applications for file and database servers. A key feature of NTFS is recoverability of the file system. We discuss this feature in Section 12.11.

NTFS disk space management is designed to be independent of sector sizes on different disks. It uses the notion of a *cluster*, which is a collection of contiguous sectors, for disk space allocation. The number of sectors in a cluster is a power of 2. A *volume* is a logical partition on a disk. A bitmap file is used to indicate which clusters in a volume are allocated and which are free. A bad clusters file keeps track of clusters that are unusable due to hardware problems. Large files that exceed the capacity of a partition are supported using the notion of a *volume set*. A volume set can contain upto 32 volumes.

An NTFS volume contains a boot sector, a *master file table* (MFT), some system files and user files. Existence of the boot sector makes every volume bootable. The MFT contains information about all files and folders on the volume. It also contains information about unused areas on the volume. Thus, the MFT resembles the file allocation table (FAT) discussed in Section 7.7.

MFT entries are given 16 bit entry numbers. Each file has a 16 bit *file number*, which is simply the entry number of the MFT entry occupied by it. A file also has a 48 bit *sequence number*. This is a count of the number of times the MFT entry has been used. The *file number* and *sequence number* together form a *file reference* that is unique within one volume. The sequence number is used to prevent mix-ups between two files that have used the same MFT entry at different times.

Each file has a set of attributes. Each attribute is considered to be an independent byte stream. Some standard attributes are defined for all files in the system. In addition, some files may have special attributes. The data in a file is also considered to be an attribute. An MFT entry contains the file reference of a file, the size of the file, and the last time it was updated. It also stores the attributes of the file. If an attribute is large in size, it is stored as a non-resident attribute. Non-resident attributes are stored elsewhere in an extent of the volume and a pointer to the extent is stored in the MFT entry. This arrangement permits the data in a small file to be stored in the MFT entry itself.

A *folder* is a directory. NTFS implements it using an index file. The directory hierarchy is formed by permitting folders to contains other folders. A directory is organized as a B+ tree with files as its leaf nodes. The B+ tree data structure has the property that the length of each path in the tree is the same, which facilitates efficient search for a file in a directory.

NTFS achieves robustness by safeguarding consistency of its data structures in the event of failures in the system. This is achieved by treating every modification of the data structures as an atomic transaction. While recovering from a failure, it checks whether a transaction was in progress at the time of failure. If so, it completes the transaction before resuming operation. This action ensures that the file system data structures are consistent whenever the file system is in operation. However, this

recovery approach is not extended to user files due to high overhead of using atomic transactions, so user files may lose data due to failures. A RAID architecture is used to avoid loss of user files.

Atomic transactions are implemented using a write-ahead log file. This approach is analogous to the implementation of atomic actions discussed in Section 7.10.2. To modify its own data structures, NTFS first writes the steps involved in the modification, i.e., its intentions, into the log file. The log file is then flushed to a disk to ensure that log records are not lost if a crash occurs. After this, the required modification of the data structures is performed. The log can now be discarded. If a crash occurs any time after the log is written to the disk, the log file is used to complete the transaction.

Atomic transactions cannot avoid loss of file system data due to a crash. To avoid such loss of data, the log is not discarded immediately after completing an update. Instead, NTFS takes a checkpoint every 5 seconds and discards the log file at that time. In the event of a crash, the file system can be restored by copying the checkpoint and processing records, if any, in the log file.

7.15 PERFORMANCE OF FILE SYSTEMS

Table 7.6 summarizes techniques used to ensure good performance of a file system. Most of these techniques concern fast access to meta-data of a file system, such as directory entries and file map tables, and file data. Other techniques address efficiency of the data structures and algorithms used in the file system. Arrangements like hash tables and B+ trees are used to make directory searches more efficient.

Table 7.6 Issues in file system performance

Issue	Techniques used to address the issue
Directory access	Directory cache
Directory search	Hash tables, B+ trees
Accessing file map table	File map table cache in memory
Accessing a disk block	Disk block cache in memory, Disk scheduling, Cylinder groups and extents, Disk block cache in device
Accessing data	Buffering and blocking of data

The techniques of *caching* and *buffering* are used to speed up accesses to meta-data and file data. Caching is the fundamental technique of speeding up access to information in a memory hierarchy (see Section 2.1.1). It involves keeping accessed information in memory to speed up repeated accesses to it. Buffering loads information in memory in anticipation of future references. Thus it speeds up the first access to information. Subsequent accesses may be speeded up by caching the information. A buffering action, though, may be wasted if buffered information is not accessed as anticipated.

Directories are cached in memory when accessed for the first time. Thus a direc-

tory used to resolve a path name is retained in a memory cache to speed up future references to files located in it. This cache is called a *directory names cache*. A file map table is buffered in memory. It is loaded in memory when the file is opened. This is done in anticipation of accesses to it. It may be cached following its first access. Buffering may not be feasible if a file map table is large in size. In that case, parts of it may be cached in memory when first referenced.

Three of the techniques mentioned in Table 7.6 are discussed later in Chapter 12. *Disk scheduling* is used to reduce disk head movements and the average wait time for I/O operations. A *disk cache* stores disk blocks in memory following their first use. It reduces the number of I/O operations on a disk. Hit ratios of 0.9 or above in the disk cache are achievable. An access method uses the techniques of *buffering* and *blocking* of file data to reduce the wait time involved in an I/O operation.

As technology advances, the trend is for speed up techniques that were developed in software to become implemented in the hardware. Modern I/O device technology incorporates some of the techniques mentioned in Table 7.6. Thus SCSI disks provide disk scheduling in the device itself. RAID units contain a disk block buffer, which can be used to both buffer and cache disk blocks. Both these technologies are discussed later in Chapter 12.

7.15.1 Log-Structured File System

Disk caching reduces the number of read operations directed at a disk. Hence disk usage is dominated by disk head movement and write operations. Disk head movement can be reduced through disk scheduling and through the use of *cylinder groups* in disk space allocation for files. However, these techniques are less effective when files located in different parts of a disk are processed simultaneously, which is the case most of the time in a shared computer system. For example, in a Unix system, write operations to a disk consume only about ten percent of the disk time; rest of the time is spent in disk head movement. This leads to poor throughput of a disk.

A *log-structured file system* reduces disk head movement through a radically different file organization. It writes file data of *all* files together in a single sequential structure that resembles a journal. We call it the *log file*. When an update or write operation is performed on any file, the new data is simply added to the end of the log file. Hence little disk head movement is involved in this operation. The file system writes special index blocks into the log file to contain meta-data about the location of each file's data in the log file. The index blocks are used when file data has to be read off the disk. Thus, little disk head movement is required for reading data that was written into a file recently; however, more disk head movement is involved for older data. Performance studies on the Sprite log-structured file system showed that disk head movement accounted for only 30 percent of the disk time consumed during file processing, and its performance was superior to the conventional file system for frequent small writes. Example 7.14 illustrates operation of a log-structured file system.

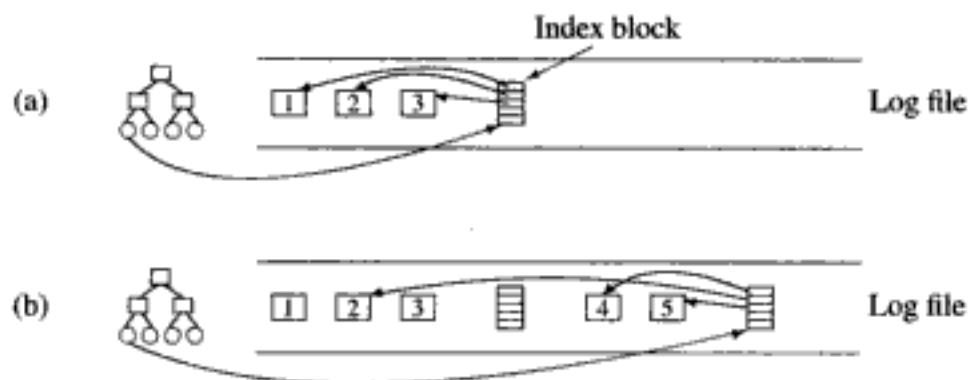


Fig. 7.32 File update in a log-structured file system

Example 7.14 Figure 7.32(a) is a schematic diagram of the arrangement used in a log-structured file system. For simplicity, it shows the meta-data and file data of a single file in the log file. The data blocks in the log file are numbered for convenience. The directory entry of a file points to an index block in the log file; we assume the index block contains the FMT of the file. When file data residing in block 1 is updated, the new values are written into a new disk block, i.e., block 4. Similarly some file data is written into disk block 5 when the data in block 3 is updated. The file system now writes a new index block that contains the updated FMT of the file and sets the FMT pointer in the directory of the file to point to the new index block. The new FMT contains pointers to the two new data blocks and to data block 2 that has not been modified (see Figure 7.32(b)). The old index block and disk blocks 1 and 3 are now free.

Since the log file is written as a sequential-access file, the file system has to ensure that a large-enough disk area is always available to write it. It achieves this by moving data blocks around on the disk to make a large free area available for the log file. This operation is analogous to memory compaction (see Section 5.5.1). It involves considerable disk head movement that now dominates the disk usage; however, compaction is performed as a background activity so it does not delay file processing activities in processes.

EXERCISE 7

1. A file named `data` is frequently accessed by users in a system. The following alternatives are proposed to simplify access to `data`.
 - (a) Set up links from every user's home directory to `data`.
 - (b) Copy `data` into every user's home directory.
 Compare advantages and drawbacks of these approaches.
2. An index sequential file contains 10,000 records. Its index contains 100 entries. Each index entry describes an area of the file that contains 100 records. If all records in the file have the same probability of being accessed, calculate the average number of disk

operations involved in accessing a record. Compare this number with the number of disk operations required if the same records were stored in a sequential file.

3. Consider the index sequential file of Figure 7.6. An interesting problem arises when a new record, say record for employee numbered 8 (we will call it record 8), is added to it. There is no space to store the new record, so the access method takes out record 13 from the track and shifts records 10 and 12 to make space for the new record. Record 13 is now put into an *overflowarea*. A new field called *overflowarea pointer* is added to each entry in the track index. This pointer in the first index entry is set to point to record 13 in the overflow area. If more records overflow out of the first track, they are put into a linked list and the overflow area pointer of the first index entry points to the head of the list. Similar linked lists may be formed for several tracks over a period of time.

If all records in the index sequential file have the same probability of being accessed, show that access efficiency of the file will be affected by presence of records in the overflow area. Can access efficiency be restored by rewriting the file as a new file that does not contain any overflow records?

4. Write a short note on actions to be performed during a file deletion operation if links exist in the directory structure.
5. The Amoeba distributed operating system uses contiguous allocation of disk space. It does not permit updating of files. An equivalent effect is achieved by writing an updated file as a new file and deleting its old copy.
Comment on advantages and drawbacks of this approach.
6. Does noncontiguous allocation of disk space influence feasibility and effectiveness of the fundamental file organizations discussed in Section 7.3?
7. A file system uses multi-level indexed disk space allocation (see Figure 7.17). The size of each disk block is 4 K bytes and each disk block address is 4 bytes in length. The size of FMT is one disk block. It contains 12 pointers to data blocks. All other pointers point to index blocks.

A sequential file `info` contains 5000 records, each of size 4 K bytes. Characteristics of the disk and of a process accessing file `info` are as follows:

$$\begin{array}{ll} \text{Average time to read a disk block} & = 3 \text{ msec} \\ \text{Average time to process a record} & = 5 \text{ msec} \end{array}$$

Calculate the elapsed time of a process that reads and processes all records in the file, under the following conditions:

- (a) The file system keeps the FMT in memory, but does not keep any index blocks in memory while processing `info`.
- (b) The file system keeps FMT and one index block of `info` in memory.
8. A new record is to be added to file `info` of Problem 7. Write an algorithm to perform this operation.
9. A file system uses contiguous allocation of disk space. The sequential access method handles bad blocks on a disk as follows: If an error occurs while reading/writing a block, it consults the bad blocks table, which is itself stored on the disk, and accesses the alternate disk block assigned to the bad block. Assuming all disk accesses to require identical access times, calculate degradation in file access performance if 2 percent of disk blocks allocated to a file are bad blocks. Suggest a method to improve

the access performance.

10. To reduce the overhead of file access validation (see Step 2 of Section 7.8.2), an OS designer proposes to perform validation only at file 'open' time. The open statement specifies the kind of accesses that will be made to the file, e.g., open (abc, 'read').
Is a single access validation check at file open time adequate? If not, explain why. In either case, suggest an implementation outline.
11. Step 2 of Section 7.8.1 creates an FCB for every directory appearing in a path name.
 - (a) Is this arrangement adequate when a relative path name is used?
 - (b) The Unix file system has an additional field in each directory, which contains the address of its parent directory. Can the information in this field be used to reduce the number of FCB's in memory?
12. Comment on implementation of the following issues relating to mounting of file systems:
 - (a) *Cascaded mounts*: Directory C contains a file D. The file system hierarchy rooted at C is mounted at mount point X/B. Later, the file system hierarchy rooted at X is mounted in directory Y/A. Can file D be accessed as ..Y/A/B/D?
 - (b) *Multiple mounts*: The file system hierarchy rooted at C is mounted at many mount points simultaneously.
13. Comment on validity of the following statement: "A memory mapped file provides advantages of both sequential and direct files."
14. An *audit trail* is a technique used to facilitate file recovery following a failure. Every time a file is updated, a record is written into the audit file containing (i) record id of the updated record, and (ii) new contents of the record. Indicate how the audit file can be used in conjunction with full and incremental back-ups to increase the reliability of a file system.
15. Discuss how the stable storage technique can be used to prevent loss of file system integrity.
16. Comment on validity of the following statement: "Mix-up between contents of files as shown in Figure 7.23 cannot occur when indexed allocation is used for files; however, a disk block may occur in more than one file if a failure occurs."
17. Algorithm 7.1 is rewritten as follows:
 1. $d_j.next := d_1.next;$
 2. $d_1.next := \text{address}(d_j);$
 3. Write d_j to disk.
 4. Write d_1 to disk.

Does this modified algorithm prevent mix-up between files if a failure occurs?

18. Explain why a full system back-up cannot be taken while file processing activities are in progress. Describe how an incremental back-up should be taken.
19. Explain how the byte offset into a Unix file can be converted into the pair (disk block id, offset in block).
20. By default, Unix assigns the files *infile* and *outfile* to the keyboard and terminal, respectively. Redirection operators '<' and '>' are used to override the default assignments, and use some other files for input and output. The 'redirect and append' opera-

- tor '>>' appends the output of a process to the end of an existing file. These features can be implemented by permanently associating FCB's for *infile* and *outfile* with each process.
- Indicate the file system actions involved in implementing the default assignments for *infile* and *outfile* and the redirection operators '<' and '>'.
 - Indicate the file system actions involved in implementing the '>>' operator.
21. Disk blocks allocated to a file are added to the free list when the file is deleted. Write an algorithm to perform this operation in Unix.
22. The Unix file system associates a lock field with the free list (see Section 7.12). Comment on validity of the following statement: "Locking of the free list is necessary due to the nature of Unix processes. Such locking is unnecessary in an OS using the conventional process model."

BIBLIOGRAPHY

Organick (1972) is historically the most important paper on directory structures, since the MULTICS directory structure has influenced most contemporary file systems like Unix, Linux, Solaris and Windows. USENIX (1992) contains proceedings of a file system workshop. Grosshans (1986), Weiderhold (1987) and Livadas (1990) discuss file organizations and file systems.

McKusick *et al* (1990) describe a memory-based file system, which provides memory mapped files and directory structures implemented in pageable memory. Levy and Silberschatz (1990) discuss file sharing semantics. Lampson (1981) describes the stable storage technique for reliability of disk data, while Svobodova (1984) surveys how atomic actions are performed in various file servers. Florido (2000) discusses design of journaling file systems. Kleiman (1986) describes the virtual file system design for Sun Unix. Vahalia (1996) describes the Unix virtual file system interface. Rosenblum and Ousterhout (1992) discuss design of the Sprite log-structured file system, while Matthews *et al* (1997) discuss adaptive methods for improving the performance of log-structured file systems. McKusick *et al* (1996) discuss the log-structured file system of Unix 4.4 BSD.

Bach (1986) and Vahalia (1996) describe the Unix file system. Kowalski (1978) describes the Unix program used to check file system integrity. This program looks through every file system data structure on disk. Bina and Emrath (1989) discuss how the file system integrity checks can be speeded up in the Unix file system. Beck *et al* (2002) and Bovet and Cesati (2003) discuss the ext2 file system of Linux. Mauro and McDougall (2001) discuss the Solaris file system. Nagar (1997) and Russinovich and Solomon (2005) describe the NTFS file system of Windows.

1. Bach, M. J. (1986): *The Design of the Unix Operating System*, Prentice-Hall, Englewood Cliffs.
2. Beck, M., H. Bohme, M. Dziadzka, U. Kunitz, R. Magnus, C. Schroter, and D. Verworner (2002): *Linux Kernel Programming*, Pearson Education,
3. Bina, E. J., and P. A. Emrath (1989): "A faster *fsck* for BSD UNIX," *Proceedings of the Winter 1989 USENIX Technical Conference*, 173–185.
4. Bovet, D. P., and M. Cesati (2003): *Understanding the Linux Kernel*, O'reilly, Sebastopol.

5. Burrows, M., C. Jerian, B. Lampson, and T. Mann (1992): "On-line data compression in a log-structured file system," *ACM Sigplan Notices*, **27**, 9, 2–9.
6. Florido, J. I. S. (2000): "Journal file systems," *Linux Gazette*, issue 55.
7. Grosshans, D. (1986): *File Systems: Design and Implementation*, Prentice Hall, Englewood Cliffs.
8. Kleiman, S. R. (1986): "Vnodes: an architecture for multiple file system types in Sun Unix," *Proceedings of the Summer 1986 USENIX Technical Conference*, 238–247.
9. Kowalski, T. (1978): "Fsck—the Unix system check program," Bell Laboratory, Murray Hill.
10. Lampson, B. W. (1981): "Atomic transactions," in *Distributed systems—Architecture and Implementation: An Advanced Course*, Goos, G., and J. Hartmanis (Eds), Springer Verlag, Berlin, 246–265.
11. Levy, H. M., and A. Silberschatz (1990): "Distributed file systems: concepts and examples," *ACM Computing Surveys*, **22**, 4, 321–374.
12. Livadas, P. (1990): *File Structures: Theory and Practice*, Prentice Hall, Englewood Cliffs.
13. Love, R. (2005): *Linux Kernel Development, Second edition*, Novell Press.
14. Matthews, J. N., D. Roselli, A. M. Costello, R. Y. Wang, and T. E. Anderson (1997): "Improving the performance of log-structured file systems with adaptive methods," *Proceedings of Sixteenth Symposium on Operating Systems Principles*, 238–251.
15. Mauro, J., and R. McDougall (2001): *Solaris Internals—Core Kernel Architecture*, Prentice-Hall.
16. McKusick, M. K., K. Bostic, M. Karels, and J. S. Quarterman (1996): *The Design and Implementation of the 4.4BSD Operating System*, Addison Wesley, Reading.
17. McKusick, M. K., M. Karels, and K. Bostic (1990): "A pageable memory based filesystem," *Proceedings of the Summer 1990 USENIX Technical Conference*, 137–144.
18. Nagar, R. (1997): *Windows NT File System Internals*, O'Reilly, Sebastopol.
19. Organick, E. I. (1972): *The MULTICS System*, MIT Press, Mass.
20. Rosenblum, M., and J. K. Ousterhout (1992): "The design and implementation of a log-structured file system," *ACM Transactions on Computer Systems*, **10**, 2, 26–52.
21. Russinovich, M. E., and D. A. Solomon (2005): *Microsoft Windows Internals, Fourth edition*, Microsoft Press, Redmond.
22. Svobodova, L. (1984): "File servers for network-based distributed systems," *ACM Computing Surveys*, **16**, 4, 353–398.
23. USENIX (1992): *Proceedings of the File Systems Workshop*, Ann Arbor, May 1992.
24. Vahalia, U. (1996): *Unix Internals: The New Frontiers*, Prentice Hall, Englewood Cliffs.
25. Weiderhold, G. (1987): *File Organization for Database Design*, McGraw-Hill, New York.

Security and Protection

Security and protection measures prevent interference with use of logical or physical resources in a system. When applied to information, these measures ensure that information is used by only authorized users and only in a desired manner, and that it is neither damaged nor destroyed. Security deals with threats to information that are external to a computer system, while protection deals with threats that are internal.

Security is implemented through authentication using passwords. It thwarts attempts by external entities to masquerade as users of a system. The technique of encryption is used to ensure confidentiality of passwords.

A user needs to share data and programs stored in files with co-workers. An *access privilege* is a specification of the manner in which a user may access a file. The owner of a file provides the OS with information concerning access privileges to the file. The protection function of an operating system stores this information and uses it to ensure that all accesses to the file are strictly in accordance with access privileges.

We start this chapter with a discussion of different kinds of security attacks and how they are perpetrated using *Trojan horses*, *viruses*, and *worms*. It is followed by a discussion of *encryption techniques* and their use in password security. We then describe three popular protection structures called *access control lists*, *capability lists*, and *protection domains*, and examine the degree of control provided by them over sharing of files.

8.1 OVERVIEW OF SECURITY AND PROTECTION

Interference in access of resources by authorized users is a serious threat in an OS. The nature of the threat depends on the nature of a resource and the manner in which it is used. In this Chapter we discuss threats to use of information stored in files. Some techniques used to counter these threats are useful for other resources as well.

Operating systems use two sets of techniques to counter threats to use of information:

- *Security* involves guarding a user's data and programs against interference by entities or persons external to a system, e.g., non-users.
- *Protection* involves guarding a user's data and programs against interference by other users of the system.

Table 8.1 describes two key methods used by operating systems to implement security and protection. Authentication is the method of verifying the identity of a person. Physical verification of identity is not feasible in contemporary computing environments, hence computer-based authentication is based on a set of assumptions. One common assumption is that a person is the user he claims to be if he knows some thing or things that only the user is expected to know. This is called authentication by knowledge. The other method is to assume a person to be a claimed user if he possesses something that only the user is expected to posses. Examples of these approaches are password-based authentication and biometric authentication, i.e., authentication based on some unique and inalterable biological features of a user such as fingerprints, retina or iris, respectively. Authorization is the act of determining the privileges of a user. The privileges are used to implement protection.

Table 8.1 Terminology used in security and protection of information

Term	Explanation
Authentication	Verifying the identity of a user. Operating systems most often perform authentication <i>by knowledge</i> . That is, a person claiming to be some user X is called upon to exhibit some knowledge shared only between the OS and user X, such as a password.
Authorization	Verifying a user's right to access a resource in a specific manner.

Figure 8.1 shows a generic scheme for implementing security and protection in an operating system. The security set up consists of the authentication service and the authentication data base. The protection set up consists of the authorization service, authorization data base and the service and resource manager. The authentication data base contains a pair of the form (login id, validating information) for every registered user of the operating system. The validating information is used to authenticate the user. To prove his identity to the operating system, a person submits his login id and some authentication information expected by the system, such as a password. The authentication service transforms the information in some standard manner and compares it with the validating information for the user. The person is a registered user if the two match.

The authentication service generates an authentication token after it has verified the identity of a user. It passes this token to the authorization service. The authorization service uses an authorization data base, which contains a pair of the form

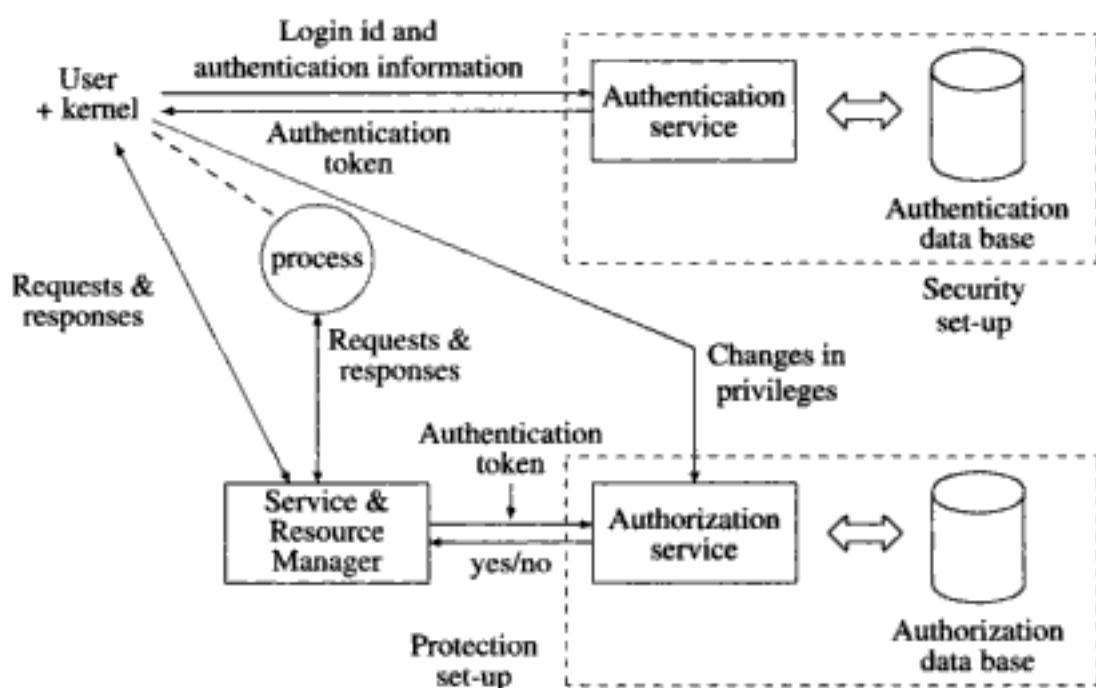


Fig. 8.1 Generic security and protection set-ups in an operating system

(authentication token, privileges) for every registered user of the system. It consults this data base to find out the privileges granted to the user and passes this information to the service and resource manager. Whenever the user or his process makes a request for a service or resource, the kernel associates the user's authentication token with it. The service and resource manager checks whether the user has been authorized to use the service or resource. It grants the request if it is consistent with the user's privileges.

For reasons that are discussed in a later section, in most operating systems the validation information for a user is an encrypted form of his password. The authentication token for a user is a user id assigned by the operating system while registering the user. The operating system remembers the user id of a user until the user logs off. It uses this information whenever a user or a process makes a request for a resource or a service. An authenticated user is allowed to use most resources of the system, except that he can use a file only if the owner of the file has explicitly authorized him to do so. This information is typically maintained and used by the file system, so authorization is not explicitly performed and no authorization data base is maintained.

The distinction between protection and security provides a neat separation of concerns for the OS. In a conventional operating system, the security concern is limited to ensuring that only registered users can use an OS. When a person logs in, a security check is performed to determine whether the person is a user of the OS and, if so, obtain his user id. Following this check, all threats to information stored

in the system are protection concerns; the OS uses the user id of a person to decide whether he can access a specific file in the OS. In a distributed system, however, security concerns are more complex due to presence of the networking component (see Chapter 20). We confine the discussion in this Chapter to conventional operating systems only.

Mechanisms and policies Table 8.2 describes mechanisms and policies in security and protection. Security policies specify whether a person should be allowed to use a system. Protection policies specify whether a user should be allowed to access a specific file. Both these policies are applied outside the OS domain. A system administrator decides whether a person should be allowed to become a user of a system. Similarly, while creating a new file a user specifies the set of users who are permitted to access it. Security and protection mechanisms are used to implement these policies by making specific checks during system operation.

Table 8.2 Policies and mechanisms in security and protection

Security <ul style="list-style-type: none"> • <i>Policy:</i> Whether a person can become a user of the system. The system administrator employs the policy while registering new users. • <i>Mechanisms:</i> Add or delete users, Verify whether a person is a registered user (i.e. perform authentication), Perform encryption to ensure confidentiality of data.
Protection <ul style="list-style-type: none"> • <i>Policy:</i> The file owner specifies the authorization policy for a file. It decides which user can access a file and in what manner. • <i>Mechanisms:</i> Set or change authorization information for a file. Check whether a file processing request conforms to the user's privileges.

The security mechanisms add new users or verify whether a person is an authorized user of the system. The latter mechanism is called *authentication*; it is invoked whenever a person wishes to log in to an OS. Protection mechanisms set protection information for a file or check whether a user can be allowed to access a file. They are invoked when a process wishes to access some file or when the owner of a file wishes to alter the list of users who may access it.

8.2 GOALS OF SECURITY AND PROTECTION

The goals of security and protection are to ensure secrecy, privacy, authenticity and integrity of information. Table 8.3 provides descriptions of these goals.

Secrecy is a security concern because it is threatened by entities outside an operating system. An OS addresses it using the authentication service. Privacy is a protection concern. An OS addresses privacy through the authorization service and the service and resource manager. The authorization service determines privileges of

Table 8.3 Goals of computer security and protection

Goal	Description
Secrecy	Only authorized users should be able to access information. This goal is also called confidentiality.
Privacy	Information should be used only for the purposes for which it is intended and shared.
Authenticity	It should be possible to verify the source or sender of information, and also verify that the information is preserved in the form in which it was created or sent.
Integrity	It should not be possible to destroy or corrupt information.

a user, and the service and resource manager disallows requests that exceed a user's privileges. It is up to the users to ensure privacy of their information using this set up. A user who wishes to share his programs and data with a few other users should set the authorization for his information accordingly. We call it *controlled sharing* of information. It is based on the need-to-know principle.

Secrecy, authenticity and integrity are both protection and security concerns. These concerns are easy to satisfy as protection concerns because the identity of a user has already been verified and the authorization and validation of a request are a part of the protection set up shown in Figure 8.1. However, elaborate arrangements are needed to satisfy secrecy, authenticity and integrity as security concerns. These are discussed in Chapter 20.

It is interesting to see how protection and security threats arise in an OS. First consider a conventional OS. It uses authentication to ensure that only authorized users can log in to the system and initiate processes. Hence it knows which user has initiated a specific process; it can readily check whether a process is allowed to use certain resources. When processes communicate with other processes, OS actions concerning communication are also confined within the same node.

The situation is different when a system has an Internet connection and a user downloads data or programs from the Internet. An entity external to the OS may be able to corrupt the data and programs being downloaded. Threats raised by such data and programs are security threats.

Security threats can arise more easily in a distributed OS. An interprocess message may cross boundaries between nodes as it travels between a sender and a receiver. Communication between nodes takes place over open communication links, including public links. Hence it is possible for an external entity to tamper with messages.

8.3 SECURITY ATTACKS

Attempts of a person or entity to breach the security of a system are called *security attacks*, and such a person is called an *intruder* or an *adversary*. Communication is a vulnerable component of a computing environment, so many attacks are launched through the communication component. Eavesdropping and tampering of messages are two obvious security attacks. These attacks primarily occur in distributed operating systems; we discuss them in Chapter 20.

Table 8.4 Classes of security attacks

Attack	Description
Masquerading	An intruder is able to impersonate a registered user of the system.
Denial of service	Some aspect of an operating system's operation is disrupted so that it cannot support some services as expected.

Table 8.4 describes two common forms of security attack in non-distributed systems. In the *masquerading* attack, an intruder could access resources of a registered user of a system, or, what is worse, he could corrupt or destroy information belonging to the user. The obvious way to launch an masquerading attack is to break the password of a user and use this knowledge to pass the authentication test at log in time. Another approach is to masquerade in a more subtle manner through programs that are imported into a software environment. We discuss this approach in Section 8.3.1.

A *denial of service* attack, also called the DoS attack, is launched by exploiting some vulnerability in the design or operation of an OS. A DoS attack can be launched through several means; some of these means can be employed only by users of a system, while others may be employed by intruders located in other systems. Many DoS attacks can be launched through legitimate means, which makes it easy to launch them and hard for an OS to detect and prevent them.

A DoS attack can be launched by corrupting a program that offers some service, or by destroying some configuration information within a kernel, e.g., use of an I/O device can be denied by changing its entry in the physical device table of the kernel. Another class of DoS attacks are launched by overloading a resource through phantom means to such an extent that genuine users of the resource are denied its use. If the kernel of an OS limits the total number of processes that can be created due to pressure on kernel data structures, a user may create a large number of processes so that no other users can create processes. Use of network sockets may be similarly denied by opening a large number of sockets. A network DoS attack may be launched by flooding the network with messages intended for a particular server so that network bandwidth is denied to genuine messages, and the server is so busy receiving messages that it cannot get around to responding to any messages. A *distributed*

DoS attack is one that is launched by a few intruders located in different hosts in the network; it is even harder to detect and prevent.

8.3.1 Trojan Horses, Viruses and Worms

Trojan horses, viruses and worms are novel ways to cause havoc in a computer system by planting programs or code that can perpetrate a variety of security attacks. Table 8.5 summarizes their features.

Table 8.5 Security threats through Trojan horses, viruses and worms

Threat	Description
Trojan horse	A program that performs a legitimate function that is known to an OS or its users, and also has a hidden component that can be used for nefarious purposes like attacks on message security or impersonation.
Virus	A piece of code that can attach itself to other programs in the system and spread to other systems when programs are copied or transferred.
Worm	A program that spreads to other computer systems by exploiting security holes like weaknesses in facilities for creation of remote processes.

In the case of a Trojan horse or a virus, a threat to computer security is planted into a computer system through subterfuge. Both enter a system when an unsuspecting user downloads a program over the Internet or loads it from a CD. On the contrary, a worm does not enter a computer system through explicit program transfers. A worm existing in one computer system spreads to other computer systems by itself. When activated, the hidden program in a Trojan horse, or the code in a virus or worm, causes security violations.

A *Trojan horse* is a program written with the intention of causing havoc in a computer system. For example, it can erase a hard disk in the computer, which is a violation of the integrity requirement, or force a system to crash or slow down, which amounts to denial of service. It can also monitor traffic between the user and other processes to collect information for masquerading, or initiate a spurious conversation with the same intent. A typical example is a spoof login program that provides a fake login prompt to fool a program into revealing password information. Since a Trojan horse is loaded explicitly by a user, it is not difficult to track its authorship or origin.

A *Virus* is a piece of code that can attach itself to other programs in the system and spread to other systems when programs are copied or transferred. While attaching itself to a program, a virus puts its own address as the execution start address of the program. This way it gets control when the program is activated and infects some programs existing on a disk in the system by attaching itself to them. After that, it passes control to the genuine program for execution. The infection step does not

consume much CPU time, hence a user has no way of knowing that the program being executed carries a virus. The way a virus attaches itself to another program makes it far more difficult to track than a Trojan horse. Apart from infecting other programs, a virus typically remains dormant until some event like a specific date or time activates it. It then launches one of the attacks mentioned earlier. An *e-mail virus* enters a computer system through an e-mail and sends spurious mails to users whose e-mail ids can be found in address-books in the computer system. It may cause mailboxes of users to overflow thus causing denial of service.

A virus cannot replicate or launch a security attack unless a virus-infected program is executed. Hence virus designers created a class of viruses called *boot-sector viruses*. These viruses plant themselves in the boot sector of a hard or floppy disk. Such a virus gets an opportunity to execute when the system is booted, and gets an opportunity to replicate when a new disk is made.

A *worm* is a program that replicates itself by spreading to other computer systems by exploiting holes in their security set-up. Worms are known to replicate at unimaginably high rates, thus loading the network and consuming CPU time during replication. Due to its self-replicating nature, a worm is even more difficult to track than a virus.

The security attacks launched through Trojan horses, viruses or worms can be foiled by doing one or more of the following:

- Exercise extreme care while loading new programs into a computer
- Use anti-virus programs
- Plug security holes as they are discovered or reported.

Loading programs from original disks on which they are supplied by a vendor can eliminate a primary source of Trojan horses or viruses. This approach is particularly effective since the introduction of the compact disk (CD) technology. Since such disks cannot be modified, a genuine program cannot be replaced by a Trojan horse, or a vendor-supplied disk cannot be infected by a virus.

Anti-virus programs analyse each program on a disk to see whether it contains any features analogous to any of the known viruses. The fundamental feature it looks for is whether the execution start address of the program has been modified or whether the first few bytes of a program perform actions resembling replication, e.g., whether they attach code to any programs on a disk.

OS vendors post information about security vulnerabilities of their operating systems on their web sites periodically and provide security patches that seal these loopholes. A user utilizing a single-user machine should check such postings and apply security patches periodically. It would foil security attacks launched through worms.

8.4 FORMAL AND PRACTICAL ASPECTS OF SECURITY

An intruder may employ a variety of ingenuous attacks to breach security, so one needs to formally prove the ability of the system to resist all forms of attacks. A for-

mal proof requires a security model comprising security policies and mechanisms, a list of threats, a list of fundamental attacks, and a proof methodology. The list of attacks must be provably complete in the sense that any conceivable attack amounting to a threat in the list of threats can be obtained through a combination of the fundamental attacks. The proof methodology should be capable of unambiguously ascertaining whether the security model can withstand certain forms of attack.

Early work in security was performed along these lines. In the take-grant model of computer security (Landwehr (1981)), processes were given privileges to objects and to other processes. A privilege for an object entitled the holder of the privilege to access the object in a specific manner. A privilege for another process entitled the holder to take an access privilege possessed by the other process (*tak*e operation), or to transfer an access privilege from itself to the other process (*agr*ant operation). The proof took the form of ascertaining whether a specific process could obtain a specific access privilege for a specific object through a series of take and grant operations.

Example 8.1 In an organization employing military-like security, all documents are classified into three security levels—*unclassified*, *confidential* and *secret*. Persons working in the organization are given security clearances called *U* (*unclassified*), *C* (*confidential*) and *S* (*secret*) with the proviso that a person can access all documents at his level of security classification and at lower levels of classification. Thus, a person with *C* classification can access *confidential* and *unclassified* documents, but not *secret* documents.

The organization uses a Unix system and persons in the organization use Unix features to access files containing documents. To check whether document security is foolproof, all operations in the system are modeled and checked to see if a person can access a document that is at a higher level of classification than his security clearance. It is found that an unrestricted use of the *setuid* feature of Unix can lead to security violations because it permits a user to execute a program with the privileges of the program's owner, rather than with his own privileges (see Section 8.8.4).

The formal approach is hard to use in practice because it needs a list of fundamental attacks; it is not possible to develop such a list for modern operating systems. The approach also requires a clear statement of security policies. This requirement is hard to meet because most security policies consist of rules that are informally stated so that everyone in an organization can understand them.

8.4.1 Practical Aspects of Security

The security component of an OS consists of an arrangement of mechanisms and policies that permits an authorized user to perform operations like sharing of information and exchange of messages with other authorized users in the system. This arrangement has to foil security attacks by an intruder.

Figure 8.2 shows an arrangement of security policies and mechanisms. It uses two basic mechanisms—*authentication* and *encryption*. Authentication has been described earlier. Encryption is used to implement secrecy of the authentication data base. We discuss encryption in Section 8.5.

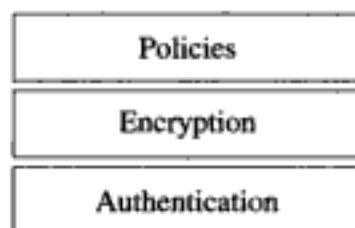


Fig. 8.2 Security mechanisms and policies

8.5 ENCRYPTION

Encryption is a technique for protecting confidentiality of data. Protection and security mechanisms use it to guard information concerning users and their resources. Encryption can also be used to guard information belonging to users. The branch of science dealing with encryption techniques is called *cryptography*. Table 8.6 summarizes key terms and definitions used in cryptography.

Encryption is performed by applying an algorithmic transformation to data. The original form of data is called the *plaintext* form and the transformed form of data is called the *encrypted* or *ciphertext* form. We use the following notation:

$$\begin{aligned} P_d &: \text{Plaintext form of data } d \\ C_d &: \text{Ciphertext form of data } d \end{aligned}$$

where $P_d \equiv d$. Confidentiality of data is protected by storing information in a ciphertext form rather than in its plaintext form. Encryption is performed by applying an encryption algorithm E with a specific encryption key k , to data. In the simplest form of encryption called *symmetric encryption*, a ciphertext is decrypted using a decryption algorithm D with the same key k to obtain its plaintext form. In advanced encryption techniques called *asymmetric encryption*, a different key k' is used to decrypt a ciphertext. Figure 8.3 illustrates symmetric encryption.

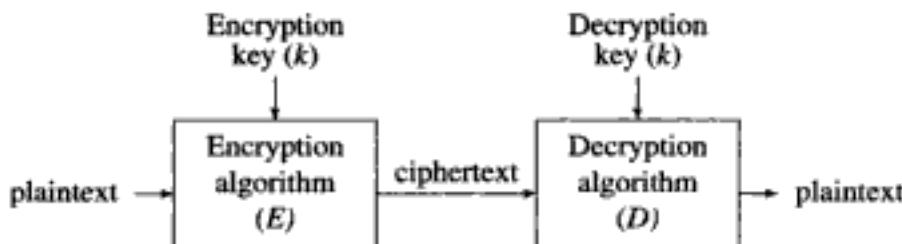


Fig. 8.3 Encryption and decryption of data

We represent encryption and decryption of data using algorithms E and D with key k as application of functions E_k and D_k , respectively. Thus,

$$\begin{aligned} C_d &= E_k(d) \\ P_d &= D_k(C_d) \end{aligned}$$

Table 8.6 Cryptography terms and definitions

Term	Description
Encryption	Encryption is application of an algorithmic transformation E_k to data, where E is an <i>encryption algorithm</i> and k is an <i>encryption key</i> . It is used to protect confidentiality of data. The original data is recovered by applying a transformation $D_{k'}$, where k' is a <i>decryption key</i> . A scheme using $k = k'$ is called <i>symmetric encryption</i> , and one using $k \neq k'$ is called <i>asymmetric encryption</i> .
Plaintext	Data to be encrypted.
Ciphertext	Encrypted form of plaintext.
Confusion	Shannon's principle of confusion requires that changes caused in a ciphertext due to a change in a plaintext should not be easy to find.
Diffusion	Shannon's principle of diffusion requires that the effect of a small substring in the plaintext should be spread widely in the ciphertext.
Attacks on cryptographic systems	An attack is a series of attempts by an intruder to find a decryption function D_k . In a <i>ciphertext only</i> attack, the intruder can examine only a set of ciphertexts to determine D_k . In a <i>known plaintext</i> attack, the intruder has an opportunity to examine the plaintext and ciphertext form of some data, whereas in a <i>chosen plaintext</i> attack the intruder can choose a plaintext and obtain its ciphertext form to perform the attack.
One-way function	A function, computation of whose inverse is expensive enough to be considered impractical. Its use as an encryption function makes cryptographic attacks difficult.
Block cipher	A block cipher technique substitutes fixed-sized blocks of plaintext by blocks of ciphertext. It introduces some confusion, but does not introduce sufficient diffusion.
Stream cipher	Both a plaintext and the encryption key are considered to be bit streams. A few bits in the plaintext are encrypted using an equal number of bits in the encryption key. A stream cipher does not introduce confusion and introduces limited diffusion; however, its variants can introduce a high level of diffusion.
DES	The Data Encryption Standard of the National Bureau of Standards, adopted in 1976, uses a block cipher technique and provides cipher block chaining as an option. It contains 16 iterations which perform complex transformations on the plaintext or the intermediate cipher text.
AES	The Advanced Encryption Standard is the new standard adopted by the National Bureau of Standards in 2001. It performs between 10 and 14 rounds of operations, each involving only substitutions and permutations, on plaintext blocks of 128, 192 or 256 bits.

Obviously functions E_k and D_k must satisfy the relation

$$D_k(E_k(d)) = d, \text{ for all } d.$$

Thus a process must be able to perform the transformation D_k in order to read or manipulate encrypted data.

Use of the encryption technique is based on the assumption that it is impractical for an intruder to determine the encryption key through trial and error. (In a later Section we see how this guarantee holds only probabilistically and not in an absolute sense.) Given this assumption, encryption can be seen to foil attacks aimed at leakage of message contents through eavesdropping. It also foils attempts at tampering of messages as follows: A message contains the ciphertext form of the information being exchanged by processes. An intruder could alter messages or fabricate new messages as there is no write protection on messages in transit. However, decryption of such a message would yield unintelligible data. Hence the recipient can differentiate between a genuine message and a tampered one. Decryption is said to be unsuccessful if it yields unintelligible data.

8.5.1 Attacks on Cryptographic Systems

An attack on a cryptographic system consists of a series of attempts to find the decryption function D_k . Since $D_k(E_k(d)) = d$, D_k is the inverse of E_k . Hence an attack implies finding the inverse of E_k . If we define the quality of encryption to mean its ability to withstand attacks, the aim of an encryption technique is to perform high quality encryption at a low cost. The encryption quality is best if function E_k is a *one-way function*, i.e., computation of its inverse through an attack involves an impractical amount of effort and time.

An intruder, who can be an internal or external entity, can launch a variety of attacks on a cryptographic system. The nature of an attack depends on the position that an intruder can occupy within the system. If an intruder cannot invoke the encryption function and can only examine data in the ciphertext form, he has to depend on guess-work. This is a trial-and-error approach in which function D_k is guessed repeatedly until its application to a ciphertext produces intelligible output. This attack is called an *exhaustive attack* because all possibilities for D_k may have to be tried out. It involves a very large number of trials. The attacks described below involve fewer trials.

Ciphertext only attack An intruder uses some information that is extraneous to a ciphertext in order to guess D_k . For example, the frequency with which each letter of the alphabet appears in English text is known. If it is known that E_k replaces each letter in a message by another letter of the alphabet (this is called a substitution cipher), an intruder may use this information to guess D_k . A ciphertext only attack is more efficient than an exhaustive attack if a suitable extraneous feature of ciphertext can be identified.

Known plaintext attack An intruder knows the plaintext corresponding to a ciphertext. This attack is possible if an intruder can gain a position within the OS from which both a plaintext and the corresponding ciphertext can be observed. Collecting a sufficient number of (plaintext, ciphertext) pairs makes it easier to determine D_k .

Chosen plaintext attack An intruder is able to supply a plaintext and observe its encrypted form. In effect one can choose a d and observe $E_k(d)$. This possibility permits the intruder to systematically build a collection of (plaintext, ciphertext) pairs to support guessing and refinement of guesses during the attack.

We see examples of these attacks when we discuss password security in Section 8.6.1. Quality of encryption is believed to improve with an increase in the number of bits in key k . For example, 2^{55} trials would be needed to break an encryption scheme employing a 56 bit key using an exhaustive attack. The large number of trials was believed to make such a scheme computationally secure from exhaustive attacks. However, powerful mathematical techniques like differential analysis can be employed to guess D_k much more easily than in an exhaustive attack. It is therefore estimated that keys will need to be several thousand bits in length to ensure that encryption functions are one-way functions.

8.5.2 Encryption Techniques

Encryption techniques differ in the way they try to defeat intruder attempts at guessing D_k . The fundamental approach is to mask the features of a plaintext, i.e., ensure that a ciphertext does not reveal features of the corresponding plaintext, without incurring a high cost of encryption.

The simplest encryption technique is the classical *substitution cipher* in which each letter in a plaintext is replaced by some other letter of the alphabet. This technique does not mask many features of plaintexts, so it is vulnerable to an attack based on analysis of the ciphertext. For example, one could guess the code for the space separating two words. Starting with this clue, it may be possible to find single letter words. There are only three such words in English—a, i and o—so it would be possible to guess the codes for these three letters. In fact, a simpler check can be used to obtain valuable clues for guessing D_k : Letters of the alphabet can be arranged in the order of decreasing frequency of usage in a ciphertext. A comparison of this list with a similar list for the English language can be used to guess D_k . To defeat such attempts, encryption techniques try to obscure the positions of letters in a plaintext.

Shannon (1949) formulated two principles for design of high quality encryption techniques. These principles are called *confusion* and *diffusion*. The confusion principle advocates that changes caused in the ciphertext due to a change in a plaintext should not be easy to find. This principle makes it difficult for an intruder to find correlations between a plaintext and the corresponding ciphertext. The diffusion principle directs that the effect of a small substring in the plaintext should be spread across the ciphertext as much as possible. When a high level of diffusion exists, a

small change in a plaintext changes many parts of the ciphertext. This feature makes it difficult for an intruder to find useful patterns in a ciphertext, hence more data is needed for a frequency analysis based attack. We describe three encryption schemes and discuss their confusion and diffusion properties.

Block cipher The block cipher is an extension of the classical substitution cipher. A block cipher technique performs substitution of fixed-sized blocks of a plaintext by ciphertext blocks of equal size. For example, a block consisting of, say, n bits is encrypted with a key k to obtain an n bit block of the ciphertext (see Figure 8.4). These blocks are assembled to obtain the ciphertext. The block cipher technique is

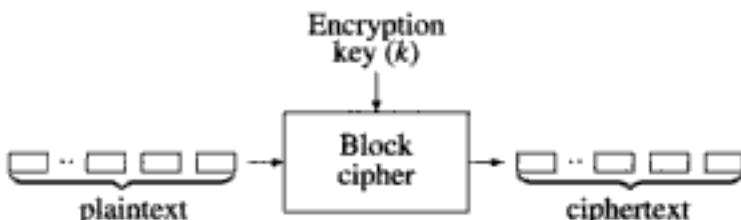


Fig. 8.4 Block cipher

simple to implement. It introduces some confusion, however it does not introduce sufficient diffusion, so identical blocks in a plaintext yield identical blocks in the ciphertext. This feature makes it vulnerable to an attack based on frequency analysis and known or chosen plaintext attacks. The severity of such attacks can be reduced by using larger values of n .

Stream cipher A stream cipher considers a plaintext as well as the encryption key to be streams of bits. Encryption is performed using a transformation that involves a few bits of the plaintext and an equal number of bits of the encryption key. A popular choice of the transformation is a bit-by-bit transformation of a plaintext, typically by performing an operation like exclusive-or on a bit of the plaintext and a bit of the encryption key.

A stream cipher is faster than a block cipher. It does not provide confusion or diffusion when a bit-by-bit transformation is used. However, it can be used to simulate the one-time pad that is famous for its use in encoding during the Second world war. Such a cipher is called the *vernam cipher*. It uses a random stream of bits whose size exactly matches the size of the plaintext as the key stream, so identical substrings in a plaintext do not lead to identical substrings in the ciphertext. The key stream is used to encode only one plaintext. Such a cipher provides a high level of security.

A variant of stream cipher variously called a *ciphertext autokey cipher*, an *asynchronous stream cipher* or a *self-synchronizing cipher* introduces diffusion. It employs a keystream generator that uses a function of the key stream and the last few bits of the ciphertext stream generated so far (see Figure 8.5). It operates as follows:

The key stream is used to encrypt the first few bits of the plaintext. The ciphertext corresponding to these bits of the plaintext is then used as a key stream to encrypt the next few bits of the plaintext and so on until the complete plaintext is encrypted. Thus a substring in the plaintext influences encryption of the rest of the plaintext.

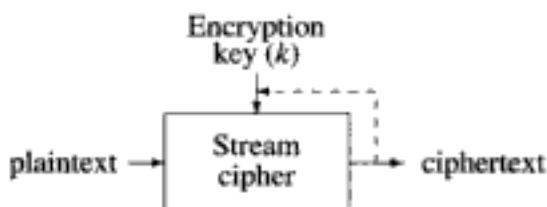


Fig. 8.5 Stream cipher

DES The Data Encryption Standard was developed by IBM for the National Bureau of Standards. It uses a 56 bit key to encrypt 64 bit data blocks. Thus, it is a block cipher. However, to overcome the problem of poor diffusion, DES provides a *cipher block chaining* (CBC) mode. In this mode, the first block of plaintext is combined with an initial vector using an exclusive-or operation and then enciphered. The resulting ciphertext is then combined with the second block of the plaintext using an exclusive-or operation and then enciphered, and so on.

DES contains explicit steps that incorporate diffusion and confusion. Diffusion is introduced using permutation of the plaintext. Confusion is provided through substitution of an m bit number by an n bit number by selectively omitting some bits, and then using the n bit number in the encryption process. These steps obscure the features of a plaintext and the encryption process sufficiently such that an intruder would have to resort to a variant of the exhaustive attack to break the cipher. An exhaustive attack would require 2^{56} trials, so it is considered impractical. (However, a message encrypted using DES was decoded through trial-and-error search jointly conducted by many users. The number of trials required in this effort was only about 25 percent of the theoretical maximum.)

DES consists of three steps—an initial permutation step, the transformation step and the final permutation step. The transformation step consists of 16 iterations. In each iteration the string input to the iteration is subjected to a complex transformation. Decryption is performed by applying the same steps in the reverse order.

Figure 8.6 illustrates operations performed in each iteration. In the first iteration, the string input to it is the plaintext. In all other iterations, the input string is the output of the previous iteration. The input string is split into two halves of 32 bits each. The right half of the input string becomes the left half of the result string. The right half is also subjected to a transformation using a key K_i that is derived by permuting the encryption key k using the iteration number i . The result of this operation is combined with the left half of the input string using an exclusive-or operation to obtain the right half of the result string.

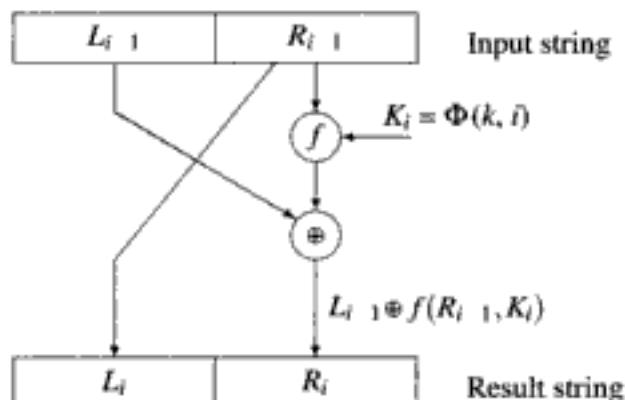


Fig. 8.6 An iteration in DES

Transformation of the right half of the input string consists of following steps: The right half is first expanded to 48 bits by permuting its bits and duplicating some of them. The expanded form is combined with key K_i using an exclusive-or operation (see function f in Figure 8.6). The result of this operation is split into eight groups of six bits each. Each six bit group is input to an S-box for substitution by a four bit group. The S-box performs a nonlinear substitution. The results of substitution are concatenated to obtain a 32 bit string that is permuted to obtain another 32 bit string. This string is combined with the left half of the input string using an exclusive-or operation to obtain right half of the result string.

One criticism of DES was that it used a small key length, which made it vulnerable to successful attacks using contemporary technology. In 1998, a message encrypted through DES was broken in less than three days using a specially designed computer. In 1999, another message was broken in less than a day using 100,000 PCs on the Internet. The *triple DES* algorithm was then endorsed as an interim standard until a new standard was adopted. It contained three iterations, where each iteration applied the DES algorithm using a different key derived from the encryption key—the first and third iterations performed encryption using their keys, while the second iteration performed decryption using its key. Effectively it could use keys upto 168 bits in length, which was considered to make it secure against attacks for a few years. The new standard called the *Advanced Encryption Standard* (AES) was adopted in 2001.

Advanced encryption standard (AES) AES is a variant of *Rijndael*, which is a compact and fast encryption algorithm using only substitutions and permutations. AES uses a block size of 128 bits and keys of 128, 192 or 256 bits, whereas Rijndael can use any key and block sizes in the range 128 bits–256 bits that are multiples of 32 bits. AES uses an array of 4×4 bytes, which is called a *state*. This is a block of the plaintext on which several rounds of operations are performed. The number of rounds depends on the key length—10 rounds are performed for 128 bit keys, 12

round for 192 bit keys and 14 rounds for 256 bit keys. Each round consists of the following operations:

1. *Byte substitution*: Each byte of the state is subjected to a nonlinear transformation applied by an S-box.
2. *Shifting of rows*: Rows in the state are shifted cyclically by 0, 1, 2 and 3 bytes, respectively.
3. *Mixing of columns*: The four bytes in a column are replaced such that each result byte is a function of all the four bytes in the column.
4. *Key addition*: A sub-key, whose size is the same as the size of the state, is derived from the encryption key using a key schedule. The sub-key and the state are viewed as bit strings and combined using the exclusive-OR operation. If this is the last round, the result of the exclusive-OR operation is a block of ciphertext; otherwise, it is used as the state for the next round of encryption.

To enable both encryption and decryption to be performed using the same sequence of steps, a key addition is performed before starting the first round, and the step of mixing of columns is skipped in the last round.

8.6 AUTHENTICATION AND PASSWORD SECURITY

Authentication by knowledge foils masquerading attacks provided authorized users do not reveal the relevant item of information to others, and provided this information cannot be guessed by a third party. Authentication is typically performed through passwords using the scheme shown in Figure 8.1. The system stores validating information of the form (login id, $\langle password_info \rangle$) in a passwords table that acts as the authentication data base, where $\langle password_info \rangle = E_k(\text{password})$. When a user performs a login, he is required to provide the correct password. The system encrypts it using E_k and compares its result with the string stored in the passwords table. The user is considered to be authentic if the two match.

If an intruder has access to the passwords table, he can launch one of the attacks described earlier in Section 8.5.1 to determine E_k . Alternatively, the intruder may launch an attack to crack the password of an individual user. In the scheme described above, if two users use identical passwords, the encrypted forms of their passwords would also be identical, which would facilitate an intruder's attempts at cracking of a password if the passwords table is visible to him. Hence the encryption function E takes two parameters. One parameter is the encryption key k , and the other parameter is a string derived from the user's login id. Now, identical passwords yield distinct encrypted strings.

Biometrics In future, operating systems may use biometric means to verify the identity of a user. The key to this approach is the use of a unique and unalterable biological feature of a user for identification. Fingerprints, retina and iris are some of the likely contenders. Widespread use of biometric techniques will however depend on availability of inexpensive and reliable equipment for biometric identification.

8.6.1 Password Security

Security of a computer system can be breached by breaking a password of some account in it. Password breaking programs depend on the fact that users may use passwords based on personal or pet names, dictionary words and vehicle numbers, or use simple keyboard sequences as their passwords. While guidelines for selecting good passwords are available on the Internet, users continue to use passwords that are not difficult to guess. Sometimes users use simple passwords that are easy to remember for infrequently used accounts, the common refrain being that they do not have many important files in that account. However, a password is the proverbial weakest link in the security chain. Any password that is broken provides an intruder with opportunities for launching further security attacks. A large number of security problems relate to use of poor passwords.

Table 8.7 OS techniques for defeating attacks on passwords

Technique	Description
Password aging	Encourage or force users to change their passwords frequently, at least once in six months. It limits the exposure of a password to intruder attacks.
Encryption of passwords	The encrypted form of passwords is stored in a system file. The file itself may be visible to all users in the system. However, only the ciphertext form of passwords is visible. An intruder who is a registered user can use the chosen plaintext attack by changing his password and viewing its encrypted form. If the encrypted passwords are visible over the Internet, which is not likely, an intruder who is not a user can use a ciphertext only attack. Else he would have to use an exhaustive attack through repeated login attempts.
Encrypt and hide password information	The encrypted file is not visible to any person within or outside the system. Hence an intruder, who may be a user or an outsider, is forced to use an exhaustive attack through repeated login attempts.

Table 8.7 summarizes techniques used to defeat attacks on passwords. *Password aging* limits the exposure of passwords to intruders, which may make passwords more secure. Another way to make passwords more secure is to let the system administrator assign 'good' passwords to users, and prevent users from changing their passwords. These passwords cannot be broken by simple techniques like looking for parts of names or dictionary words in the passwords, so an intruder would have to launch an exhaustive attack that involves trying every conceivable string as a password. The time and effort involved in this approach makes it impractical.

Encryption is a standard technique to protect the authentication data base. An intruder would have to analyze the encrypted passwords file to break passwords.

He can use one of the attacks described in Section 8.5.1. A registered user can try a chosen plaintext attack by changing his own password repeatedly and viewing its encrypted form. An outsider would have to use an exhaustive attack by trying to login with different passwords. If the encrypted password file is not visible to anybody other than the root, any intruder within or outside the system would have to use an exhaustive attack through repeated login attempts.

Unix and Linux perform encryption of passwords. Unix uses DES encryption. Linux uses a message digest technique, which is a one-way hash function that generates a 128 or 160 bit a hash value from a password. This technique has variants called MD2, MD4 and MD5. Linux uses MD5. Both Unix and Linux provide a shadow passwords file option. When this option is chosen, the ciphertext form of passwords is stored in a shadow file that is accessible only to the root. As mentioned earlier, this arrangement forces an intruder to use an exhaustive attack.

8.7 ACCESS DESCRIPTORS AND THE ACCESS CONTROL MATRIX

A user may be permitted to access a set of files and a file may be accessible by a set of users. A *protection structure* contains information indicating which users can access which files, and in what manner. It is the classical name for the authorization data base discussed in Section 8.1.

An *access privilege* for a file is a right to make a specific form of access to the file, e.g., a read or write access. An *access descriptor* describes access privileges for a file. We use obvious notation like *r*, *w* and *x* to represent access privileges to read, write and execute the data or program in a file. An access descriptor is represented as a set, e.g., the descriptor {*r*, *w*} indicates privileges to read and write a file. *Access control information* for a file is a collection of access descriptors for access privileges held by various users.

An *access control matrix* (ACM) stores protection information consisting of access privileges of users and access control information for files. A row in ACM describes the access privileges held by one user. A column describes access control information for a file. Thus, $\text{ACM}(u_i, f_j) = a_{ij}$ implies that user u_i can access file f_j in accordance with access privileges a_{ij} . Figure 8.7 shows an ACM. User Jay has {read, write} access privileges for beta but only {read} privilege for alpha. When a process P_i tries to perform an access a_k to a file f_j , the kernel obtains user id of the user who initiated process P_i . Let this be u_i . The kernel now uses u_i and f_j to obtain the entry $\text{ACM}(u_i, f_j)$. The access made by the process is legal if a_k is included in the list of access privileges contained in $\text{ACM}(u_i, f_j)$.

Access descriptors can be bit-encoded for memory and access efficiency. One bit in such an access descriptor indicates the presence or absence of one access right. In an OS using only three access rights *r*, *w* and *x*, a bit encoded access descriptor would contain 3 bits indicating the presence or absence of each access right. In a bit-encoded form, access descriptor for the access privileges {read, write} may look like the string 110... where the 1s indicate that the read and write privileges are granted,

		Files -->		
		alpha	beta	gamma
Users ↓	Jay	{r}	{r,w}	
	Anita	{r,w,x}		{r}
	Sheila			{r}

Fig. 8.7 Access control matrix (ACM)

and the 0 indicates that an execute privilege is not granted.

Granularity of Protection *Granularity of protection* signifies the degree of control over protection of files. Three levels of granularity are described in Table 8.8. An access control matrix possesses medium granularity. It stores access privileges for each (user, file) pair. Thus, users may possess different access privileges for same file. Coarse granularity exists if the owner of a file is unable to specify access privileges for each user in the system. Typically this happens when users are divided into user groups and access privileges are specified for a group of users. Fine granularity exists if access privileges are specified for processes or execution phases of processes. This way, different processes created by the same user may possess different access privileges for a file, or the same process may possess different access privileges at different times. Fine granularity provides users with much more discrimination than coarse or medium-grained protection. We discuss fine granularity of protection in Section 8.8.3.

Table 8.8 Granularity of protection

Coarse-grained	The owner of a file specifies access privileges to groups of users. Each user in a group has identical access privileges to the file.
Medium-grained	Access privileges can be specified individually for each user in the system.
Fine-grained	Access privileges can be specified for a process, or for a phase of execution of a process.

Disadvantages of the access control matrix Use of an ACM provides simplicity and efficiency of access to the protection information—access privileges of user u_i for file f_i can be obtained from $\text{ACM}(u_i, f_i)$. However, ACM is large in size. If an OS contains n_u users and n_f files, the size of ACM is $n_u \times n_f$. Since both n_u and n_f would be large, large areas of memory have to be committed to hold the ACM, or a part of it, during system operation. A typical user possesses access privileges for few files, hence most entries in ACM contain null information. Thus, the amount of access control information is much smaller than the size of ACM.

The size of access control information can be reduced in two ways—by reducing the number of rows in ACM or by eliminating null information. The number of rows

are reduced by assigning access privileges to groups of users rather than to individual users. Now ACM needs to contain only as many rows as the number of user groups; however, reducing the number of rows compromises granularity of protection.

The size of access control information can be reduced by eliminating null information. Typically, the information is stored in the form of lists rather than a matrix. This approach does not affect the granularity of protection but reduces the size of protection information since only non-null entries of an ACM need to be present in a list. Two list structures are used in practice:

1. Access control lists (ACLs)
2. Capability lists (C-lists).

An access control list stores the non-null information from a column of the ACM. Thus it contains access control information for one file in the system. When a user attempts to use a file, the access control list of that file is searched to locate a pair involving that user. An access is invalid if a pair involving the user does not exist in the list or the attempted form of access is not consistent with access privileges of the user. A capability-list stores the non-null information from a row of the ACM. It describes all access privileges held by a user. The C-list of a user is searched when a user attempts to access a file.

Access control lists are widely used for file protection. Capability lists can be used to protect files as well as resources like programs or data objects in memory. We discuss such use of C-lists in Section 8.9.

8.8 PROTECTION STRUCTURES

A protection structure is a method of storing and using access control information. The nature and use of the information determines granularity of protection. The access control list can provide coarse or medium grained protection, the capability list provides medium grained protection while protection domains provide fine grained protection.

8.8.1 Access Control Lists

The access control list of a file indicates which users can access the file, and in what manner. It is typically stored in the directory entry of the file. Whenever a process wishes to open a file, OS finds the id of the user who initiated the process and checks whether the user's access privileges permit the intended mode of access. The process is aborted if this is not the case. (Actually, the file system of the OS makes this check. However, for simplicity, we will assume that the OS makes all protection checks.)

When a user creates a file, he registers the file's access control information with the system. The access control list for the file stores this information in the form of a set of pairs $\{(user_id, access_privileges), \dots\}$. Figure 8.8 shows access control lists for three files. ACL for alpha is $\{(Jay, \{read\}), (Anita, \{read, write, execute\})\}$, which indicates that user Jay can only read file alpha while Anita can read, write

or execute the file. User Sheila is not permitted any kind of access to alpha, since alpha's ACL does not contain an entry for Sheila. In most file systems the owner gets certain access privileges by default. In Unix, these access privileges are controlled by the user mask.

<i>File name</i>	<i>Location info</i>	<i>Protection info</i>	<i>Flags</i>
alpha		$\{(Jay, \{r\}), (Anita, \{r,w,x\})\}$	
beta		$\{(Jay, \{r,w\})\}$	
gamma		$\{(Anita, \{r\}), (Sheila, \{r\})\}$	

Fig. 8.8 Access control lists

Steps involved in the access of file gamma by user Sheila are as follows:

1. User Sheila performs a login.
2. The system authenticates Sheila by asking for her password.
3. A process initiated by Sheila issues the call `open(gamma, 'write')` for modifying file gamma.
4. OS locates the pair involving sheila in the ACL of file gamma, and checks whether it contains the 'write' access privilege for Sheila.

Note that masquerading is not possible during file access since Sheila has been authenticated by the system in step 2.

Use of access control lists faces some practical problems. Presence of a large number of users in a system leads to large ACL sizes, and thereby to large space overhead in the file system. The time overhead is also high due to ACL searches for validating a file access. Both memory and CPU time can be conserved at the cost of a coarse granularity of protection by specifying and storing protection information for groups of users rather than for individual users. Section 8.10 describes how Unix stores and searches an ACL in an efficient manner.

8.8.2 Capability Lists (C-lists)

A capability represents access privileges of a user for one file. Information concerning capabilities possessed by a user is stored in a *capability list* (C-list). A C-list is thus a set of pairs $\{(file_id, access_privileges), \dots\}$. Figure 8.9 shows the C-list $\{(alpha, \{read, write, execute\}), (gamma, \{read\})\}$ for user Anita. Anita can read, write or execute file alpha and can read file gamma. Anita has no access to file beta, since no entry for beta exists in the C-list. When Anita wishes to access some file, the OS implements the access only if Anita's capability list contains the necessary access privilege for the file. As in the case of ACLs, masquerading possibilities do not exist since a user is authenticated by the system at login time and the user id is used to locate the correct C-list. C-lists are usually small in size. This

feature limits the space and time overhead in using them to control file access.

(alpha, {r, w, x})
(gamma, {r})

Fig. 8.9 Capability list for user Anita

8.8.3 Protection Domain

As mentioned in Section 8.1, operating systems typically address the secrecy aspect of protection and leave the privacy aspect to users and their processes. It is interesting to see why this is so. Use of an ACM, or its variants like ACLs and C-lists, confers access privileges on users. One peculiarity of this arrangement is that every process created by a user has the same access privileges! The arrangement serves the secrecy concern in protection because only authorized users can access a file; however, it does not adequately address the privacy concern because it does not differentiate between different processes created by a user.

Violations of the privacy requirement may arise as follows: An access privilege is granted to a user because *some* process initiated by the user requires it. However, any process created by the user can use the access privilege, so some process initiated by the user may put the information to an unintended or wrong use. Such usage violates the privacy requirement.

Possibility of unintended use of a file by processes raises a major reliability concern as the correctness of data would depend not only on correct manipulation by processes that are supposed to access it, but on harmlessness of actions performed by processes that are not supposed to access it! Example 8.2 illustrates how privacy of information may be jeopardized.

Example 8.2 A user u_i has an execute privilege for a program `invest` owned by another user u_j . When u_i executes `invest`, `invest` executes as a process initiated by user u_i . It can access any file for which user u_i holds an access privilege, including files that have nothing to do with investments! If u_i so wishes, (s)he can code `invest` such that `invest` can access confidential files of u_j and make copies or even modify them.

The notion of a protection domain addresses the privacy concern. We can think of a protection domain as a conceptual ‘execution environment’. Access privileges are granted to a protection domain rather than to a user. A process operates in a protection domain. By default, the initial execution domain of a process does not possess any access privileges. This way a process cannot access any resources while it executes in this domain, not even resources owned by the user who initiated it. In order to access any specific resource, it must ‘enter’ a protection domain

that possesses access privileges for the resource. The kernel provides system calls through which a process may request entry into a protection domain. A set of conditions would be defined for legality of such a request. The kernel would apply the conditions and either honor the request for change of protection domain or abort the process for making an illegal request.

The protection arrangement involving use of protection domains facilitates implementation of the need-to-know principle with a fine granularity. Only processes that need to access a resource are now granted access to it. Example 8.3 illustrates how this approach provides privacy of information and thereby improves data reliability.

		Files →				
		personal	finance	memos	notes	project
Domains ↓	D_1	{r,w}	{r,w}			
	D_2		{r}			
	D_3			{r,w}	{r,w}	{r}

Fig. 8.10 Protection domains

Example 8.3 Figure 8.10 shows three protection domains. Domain D_1 has read and write access rights for files `personal` and `finance` while D_2 possesses only a read access right for `finance`. Domain D_3 has read and write access rights to files `memos` and `notes` and a read access to file `project`. Thus domains D_1 and D_2 overlap while domain D_3 is disjoint with both of them. User u_i executes three computations named `self`, `invest` and `job_related` in domains D_1 , D_2 and D_3 , respectively. Thus `invest` can access only file `finance`, and can only read it.

In an OS that does not use protection domains, user u_i would need read and write access rights to files `personal`, `finance`, `memos` and `notes` and a read access right to file `project`. When user u executes program `invest` that is owned by user u_j , `invest` will be able to modify many files accessible to u !

8.8.4 Change of Protection Domain

A process may be permitted to change its protection domain during execution. This facility may be used to make some resources accessible only during specific phases of execution of a process, thus enhancing privacy of information. We discuss two practical examples of change of protection domain.

Unix A Unix process has two distinct *running* states—user running and kernel running (see Section 3.5.1). While in the user running state, it has access to its own memory space, to files in the file system according to the id of the user who created it, and to other resources allocated to it. The process makes a transition to the kernel running state through a system call. In this state, it can access kernel data structures and also contents of the entire memory. It returns to the user running state when it returns from the system call. Now once again it has only those access privileges that it held before the system call.

The *setuid* feature provides another method of changing protection domains dynamically. The access privileges of a process are determined by its uid. When the kernel creates a process, it sets uid of the process to the id of the user who created it. The *setuid* feature provides a method of temporarily changing the uid of a process. *setuid* can be used as a system call by a process or it can be used when a process performs an *exec* in order to execute a program. The process makes a *setuid* system call with its own id in order to revert to its original uid.

Let a program P be stored in a file named P. If P is to be executed by invoking the *setuid* feature, the *setuid* bit in the inode of file P is set. When P is *exec*'ed, the kernel notices that the *setuid* bit of file P is set, and changes the uid of the process executing P temporarily to the uid of the owner of P. This action effectively puts the process into a protection domain whose access privileges are identical with access privileges of the P's owner. Using this feature, the situation discussed in Example 8.2 can be avoided as follows: User u_j sets the *setuid* bit of program *invest*. User u_i provides u_j with a read access to file *finance* before invoking *invest*. Now, program *invest* can access user u_i 's file *finance*, but it cannot access any other files owned by u_i .

MULTICS MULTICS provides 64 protection domains that are organized as concentric rings. The rings are numbered from the innermost to the outermost (see Figure 8.11). They provide a hierarchy of protection domains—that is, the access privileges of a domain include access privileges of all higher numbered domains. In addition, it may have a few other access privileges. Each procedure of a program is assigned to a protection domain and can be executed only by a process that is in the same protection domain.

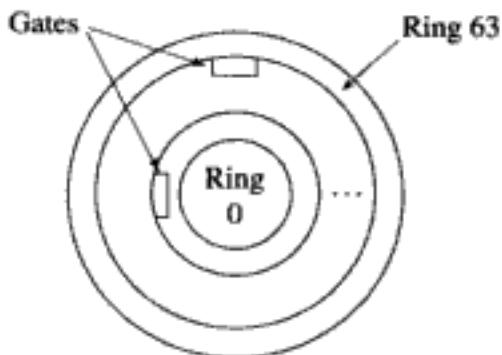


Fig. 8.11 Multics protection rings

The code component of a process may consist of procedures in different protection domains. An interrupt is raised when a process executing in protection domain D_i invokes a procedure that is assigned to a protection domain D_j , where $D_j \neq D_i$. To execute the procedure, the protection domain of the process should be changed to D_j . The kernel checks if this is permissible by rules for change of protection domain. A simplified view of these rules is as follows: Change of protection domain is per-

mitted if a process running in some domain D_i invokes a procedure that exists in a higher numbered domain. However, to enter a lower numbered domain a process must invoke a specially designated procedure called a *gate*. An attempt to invoke any other procedure in a lower numbered layer fails and the process is aborted. If a call satisfies one of these rules, the protection domain of the process is temporarily changed to the domain in which the invoked procedure exists. The invoked procedure executes in this protection domain and accesses resources according to its access privileges. At return, the protection domain of the process is reset to its earlier value, i.e., to D_i .

The MULTICS protection structure is complex and incurs substantial execution overhead due to checks made at a procedure call. Its hierarchical nature dictates that an access privilege possessed by a protection domain should include access privileges of all higher numbered protection domains. This requirement does not permit protection domains whose access privileges are disjoint. For example, domains D_1, D_2 and D_3 of Figure 8.10 cannot be implemented in MULTICS since domain D_3 is disjoint with domains D_1 and D_2 . This feature restricts users' freedom in specifying protection requirements.

8.9 CAPABILITIES

The concept of a capability [Dennis, Van Horn, 1966] was proposed as a general mechanism for sharing and protection. Capabilities in a C-list used for file protection (see Section 8.8.2) are an adaptation of this general mechanism.

Definition 8.1 (Capability) A capability is a token representing access privileges for an object.

A capability contains the id of an object and a list of access descriptors for it. An object is any hardware or software entity in the system, e.g., a laser printer, a CPU, a file, a program, or a data structure of a program. A capability is possessed by a process. It gives the process a right to access the object in a manner that is consistent with the access privileges.

When some process P_i creates an object O_i , the OS forms a capability for O_i that contains the entire set of access privileges, and passes this capability to P_i . Process P_i can make copies of this capability, or it can create *subset capabilities* that contain fewer access privileges, and pass these capabilities to other processes. Thus, many capabilities for O_i may exist in the system. Use of these capabilities by processes leads to sharing of O_i .

Each process possesses capabilities for the objects it owns, and some capabilities passed to it by other processes. All capabilities in a C-list are obtained through legal means—none can be stolen or fraudulently created by a process. This is why a capability is often described as an unforgeable token that confers access privileges onto its holder.

The Format of a capability is illustrated in Figure 8.12. The capability consists of two fields—*object id* and *access privileges*. Each object has an unique object id in the system. The access privileges field typically contains a bit-encoded access descriptor. Thus, each bit in the access privileges field represents the presence or absence of one specific access privilege.

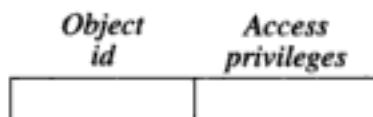


Fig. 8.12 Format of a capability

We use the notation $\text{Cap}_k(\text{obj}_i)$ to refer to a capability for obj_i . The subscript of Cap is used simply to distinguish between different capabilities for an object. It does not have any other significance. For simplicity, we omit the subscript in contexts where a single capability of an object is involved.

8.9.1 Capability Based Computer Systems

A capability based computer system is a system whose architecture implements capability based addressing and protection for all objects in the system. Objects include long-life objects like files and short-life objects like data structures and copies of programs in memory. Many capability based systems were built for research and experimentation. The Intel iapx-432 is an example of a well known commercial processor with a capability based architecture.

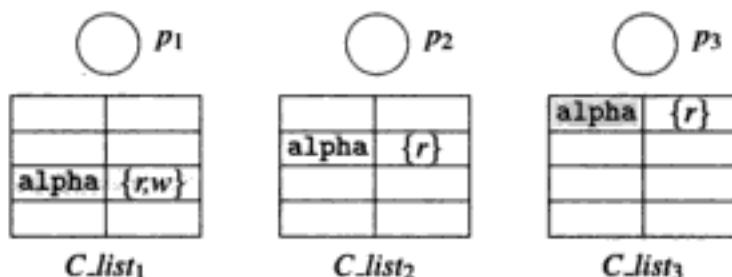


Fig. 8.13 Schematic of a capability based system

Figure 8.13 shows a schematic of a capability-based system. Each process has a C-list describing its access privileges. Processes P_2 and P_3 contain identical capabilities of alpha that confer the read privilege for alpha , while process P_1 holds a capability that confers the read and write privileges for alpha . A capability based system differs from a conventional computer system in the following respects:

1. The system does not explicitly associate ‘memory’ with processes; it associates C-lists with processes.
2. The capabilities in a C-list may be used to access objects existing anywhere in the system (i.e., in memory or on disk); the location of an object is immaterial

to a process.

3. Long-life objects like files and short-life objects like data and programs can be accessed in a uniform manner.

These differences arise from the manner in which an object is accessed in a capability based system.

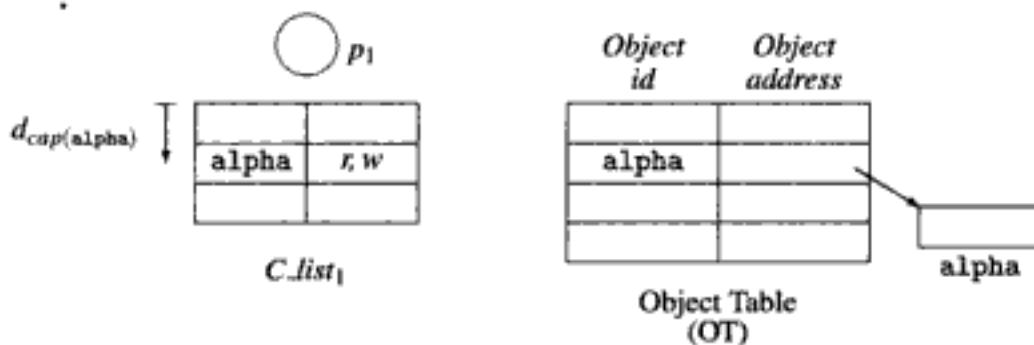


Fig. 8.14 Capability based addressing

Capability based addressing Figure 8.14 shows a schematic of capability based addressing of objects. Each object has an unique id. The *object table* (OT) is a system-wide table that contains location information for all objects in the system. The *object address* field of an OT entry indicates address of the object in the computer's primary or secondary memory. Access to an object is implemented as follows: A process P_1 performs an operation $\langle op_i \rangle$ on an object by using an instruction of the form

$$\langle op_i \rangle \quad d_{Cap(obj_i)} \quad (8.1)$$

where $d_{Cap(obj_i)}$ is the displacement of $Cap(obj_i)$ in P_1 's C-list. The system uses $d_{Cap(obj_i)}$ to locate the capability in P_1 's C-list. The object id in the capability, viz. $alpha$, is used to locate $alpha$'s entry in OT. The object address in the OT entry is used to access $alpha$ to implement $\langle op_i \rangle$. Special techniques analogous to address translation buffers (see Section 6.2.3), viz. capability registers and cache memories, are used to make object access more efficient.

Capability based addressing provides some vital flexibility to the OS. The OS can move objects around in memory for better memory management, or move them between memory and disk for cost-effective access performance, without affecting the manner in which a program accesses the object. This advantage is called the *uniform addressing* advantage of capabilities.

Operations on objects and capabilities A process P_k possessing $Cap(obj_i)$ can perform the operations listed in Table 8.9 on obj_i and $Cap(obj_i)$. As discussed before, operations on obj_i are subject to access privileges in the capability. In many capability based systems, operations on a capability are also subject to access privileges in the capability. For example, a process may be able to create a subset capability of

$\text{Cap}(\text{obj}_i)$ only if $\text{Cap}(\text{obj}_i)$ contains the access privilege 'create subset capability'. This feature controls the operations that processes can perform on capabilities.

Table 8.9 Permissible operations on objects and capabilities

<p style="margin: 0;">Operations on objects</p> <ul style="list-style-type: none"> • Read or modify the object • Destroy the object • Copy the object • Execute the object <p style="margin: 0;">Operations on capabilities</p> <ul style="list-style-type: none"> • Make a copy of the capability • Create a 'subset' capability • Use it as a parameter in a function/procedure call • Pass the capability for use by another process • Delete the capability
--

8.9.2 Sharing and Protection of Objects

Object access in a capability based system proceeds as follows:

1. When process P_i wishes to perform an operation on an object Obj_i , it executes an instruction of the form $<\text{op}_i> d_{\text{Cap}(\text{obj}_i)}$ where $<\text{op}_i>$ is the operation code for the desired operation.
2. The CPU checks whether execution of $<\text{op}_i>$ is consistent with the access privileges contained in $\text{Cap}(\text{obj}_i)$. If so, it performs the operation; otherwise, it raises a protection violation interrupt.
3. OS aborts the process if a protection violation interrupt arises.

CPU performs the following additional actions if op_i is the operation 'create a new object': It creates a new object and creates an entry for the object in the OT. It puts the object id and address of the newly created object in the entry. It now creates a capability containing the entire set of access privileges for the object and puts this capability in the C-list of P_i . It also puts $d_{\text{Cap}(\text{obj}_i)}$ in the result register of operation op_i . Process P_i saves this value for use while accessing obj_i in future. Analogous actions are performed when a process makes a copy of an object.

Sharing of objects occurs when a process passes a capability for an object to another process. The recipient process enters the capability in its own C-list. Sharing is implicit in the fact that both C-lists contain a capability for the object. Protection is implicit in the fact that these capabilities may confer different access privileges on the processes. Example 8.4 illustrates sharing and protection of objects.

Example 8.4 The following events occur in an OS:

1. Process P_1 creates an object obj_i . The OS inserts a capability $\text{Cap}_1(\text{obj}_i)$ in P_1 's C-list and returns displacement of the capability to P_1 . This capability confers the entire set of access privileges on its holder, including the access right 'pass', which enables the process to pass the capability to other processes.

2. P_1 creates a subset capability of $Cap_1(obj_i)$ containing only the read privilege. Let us call this $Cap_2(obj_i)$.
3. P_1 passes $Cap_2(obj_i)$ to process P_2 .
4. P_2 accepts the capability. (Typically steps 3 and 4 are performed using a hand-shake mechanism.) The capability is entered in the C-list of P_2 at the displacement $d_{Cap_2(obj_i)}$.
5. P_2 now uses $d_{Cap_2(obj_i)}$ to perform a read access to obj_i .

Figure 8.15 illustrates the resulting C-lists of P_1 and P_2 . The processes possess different access privileges for obj_i .

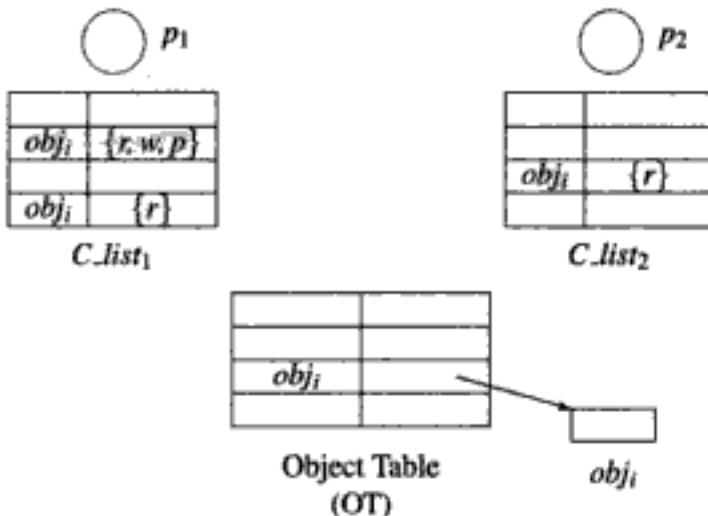


Fig. 8.15 Capability based object sharing and protection

Protection of capabilities Protection using capabilities is based on the fundamental assumption that capabilities cannot be forged or tampered with. This assumption would be invalid if a process can access its C-list and modify the capabilities existing in it. For example, process P_2 of Figure 8.15 could alter the access privileges field of the capability for obj_i and give itself a ‘write’ access privilege. It could then use the modified capability to modify object obj_i !

Tampering and forgery of capabilities is prevented using the following two approaches:

- Tagged architectures
- Capability segments.

These approaches are similar to the approaches used to protect data against illegal operations and to protect programs against modification, respectively.

Tagged architecture The run-time representation of an object consists of two fields—a *tag* field and a *value* field. The tag field describes the type of the object, while the value field contains a representation of the object’s value. The tag field can

destroyed without such information. Dangling pointers can exist—that is, an object may be destroyed while some capabilities still exist for it—or an object may exist long after capabilities for it have been destroyed. Preventing both these situations requires use of expensive garbage collection techniques.

Confinement of capabilities Confinement implies restricting the use of a capability to a given set of processes. The confinement problem arises if a process needlessly passes a capability to other processes. Such actions cause a proliferation of capabilities, which complicates garbage collection and prolongs the life of an object. Proliferation can also undermine protection by violating the need-to-know principle. Confinement can be achieved by making the passing of a capability itself an access right. If process P_i turns off the ‘pass’ access right while passing a capability to P_j , P_j will not be able to pass the capability to any other process.

Revocation of capabilities Revocation of all capabilities for an object is the most difficult problem in a capability based system since there is no way to know which processes hold capabilities for the object. Difficulties in revocation also imply that access privileges cannot be granted to a process for a limited period of time. Interestingly, revocation is possible in the case of software capabilities that are protected through encryption. In Amoeba, change of the key granted to an object would automatically invalidate all existing capabilities of the object. For selective revocation the owner can invalidate all capabilities and then issue fresh ones that are encrypted using the new key of the object to some processes. However, this is an expensive and obtrusive operation—every process holding a capability for the object is affected when *any* capability of the object is to be revoked.

8.10 UNIX SECURITY

As mentioned in Section 8.6.1, Unix employs encryption for password security. Under an option, it uses a shadow passwords file that is accessible only to the root, which forces an intruder to use an exhaustive attack to crack passwords. Each Unix user has a distinct id in the system. The system administrator creates non-overlapping groups of users and assigns a unique group id to each group. The credential of a user is comprised of his user id and group id. It is stored in the passwords table and becomes the authentication token of the user after the user is authenticated.

Unix defines three user classes—file owner, user group and other users—and provides only three access rights r , w and x (representing read, write and execute, respectively). Thus, the ACL needs to record only the presence of three access rights for each of the user classes. A bit-encoded access descriptor is used. It is stored in a field in the directory entry of a file. The identity of the file owner is stored in another field of the file’s directory entry. Figure 8.18 shows the Unix ACLs as reported in a directory listing. The file `sigma` can be read by any user in the system, but can be written only by its owner. `delta` is a read-only file for all user classes, while `phi`

This arrangement provides flexibility because the authentication scheme used in an application can be changed without having to recompile the application. Application developers can use PAM to enhance application security in several ways—to employ a password encryption scheme of own choice, to set resource limits to users so that they cannot launch denial of service attacks, and to allow specific users to login only at specific times from specific places.

The system administrator maintains a PAM configuration file for each application that is authorized to employ PAM. Each PAM configuration file specifies how authentication is to be performed and what actions, such as mounting of home directories or logging of the authentication event, are to be taken after a user is authenticated. The configuration file also names the mechanism that is to be employed when a user wishes to change his password. PAM permits several authentication modules to be ‘stacked’; these modules are invoked one after another. An application can use this facility to authenticate a user through several means such as passwords and biometric identification, to enhance security.

Linux provides file access protection based on user id and group id of a process. When a server such as the NFS accesses a file on behalf of a user, file protection should be performed using the user id and group id of the user rather than those of the server. To facilitate this, Linux provides the system calls `fsuid` and `fsgid` through which a server can temporarily assume the identity of its client.

The Linux kernel supports loadable kernel modules. This feature has been employed to provide enhanced access controls through loadable kernel modules called the *Linux Security Modules* (LSM). Use of LSMs permits many different security models to be supported. The basic schematic of LSM is simple: The kernel invokes an access validation function before accessing an object. An LSM provides this function, which may permit or deny the access to go through. The Security Enhanced Linux (SELinux) of the US National Security Agency has built additional access control mechanisms through LSM which provide mandatory access control.

The Linux kernel provides the exec-shield patch which enables protection against exploitation of buffer overflows and data structure overwriting to launch security attacks.

8.12 WINDOWS SECURITY

The Windows security model has several elements of C2- and B2-class systems according to the Trusted Computer System Evaluation Criteria (TCSEC) of the US Department of Defense. It provides discretionary access control, object reuse protection, auditing of security-related events, a security reference monitor (SRM) that enforces access control, and a trusted path for authentication that would defeat Trojan horse attacks for masquerading. Among other notable features, it provides security for client-server computing through access tokens, which are analogous to capabilities (see Section 8.9).

Windows security is based around the use of *security identifiers* (SIDs); a secu-

urity identifier is assigned to a user, a computer, or a domain, which is comprised of several computers. The important fields in an SID are a 48-bit identifier authority value, which identifies the computer or domain that issued the SID, and a few 32-bit subauthority or relative identifier (RID) values that are used primarily to generate unique SIDs for entities created by the same computer or domain.

Each process and thread has an *access token* that identifies its security context. (As elsewhere in the text, we will use the term *process* as being generic to both a process and a thread.) An access token is generated when a user logs on, and it is associated with the initial process created for the user. A process can create more access tokens through the *LogonUser* function. An access token contains a user account SID and a group account SID. These fields are used by the security reference monitor to decide whether the process holding the access token can perform certain operations on an object. An access token also contains a privilege array indicating any special privileges held by the process, such as a privilege for creating back-ups of files, impersonating a client, and shutting down a computer. It may also contain a few super-privileges for loading and unloading drivers, taking ownership of objects, and creating new access tokens.

An object such as a file has a *security descriptor* associated with it. The security descriptor contains the object owner's id, a *discretionary access control list* (DACL) and a *system access control list* (SACL). Both DACL and SACL are lists of *access control entries* (ACEs). Each ACE contains an SID of a user. An ACE in a DACL either allows or disallows an access to the object by a user with a matching SID. This arrangement permits medium granularity and yet helps to make the DACL compact; however, the entire DACL has to be processed to determine whether a user is allowed to access the object in a specific manner. An object that can contain other objects, such as a directory or a folder, is called a *container* object; we will call the objects contained in it as its child objects. An ACE in the DACL of a container object contains flags to indicate how the ACE is to apply to a child object—identically, not at all, or in some other manner. An important option is that the ACE may be inherited by a child object that is itself a container object, but it may not be further inherited by objects that may be created within the child object. This feature helps to limit the propagation of access control privileges.

The SACL is used to generate an audit log. The ACEs in the SACL indicate which operations on the object by which users or groups of users should be audited. An entry is made in the audit log when any of these operations is performed.

The *impersonation* feature in the Windows security model provides security in client–server computing. When a server performs some operations on objects on behalf of a client, these operations should be subject to the access privileges of the client rather than those of the server; otherwise, the client would be able to realize operations on these objects that exceed its own access privileges. Analogously, the security audit information that is generated when the server accesses an object on behalf of a client should contain the identity of the client rather than that of the

12. Different nodes of a distributed system may concurrently create new objects. Describe a scheme that can ensure uniqueness of object ids in an OS.
13. Study and describe the provisions in Unix for (a) finding the id of the user who owns a file, (b) deciding whether a user belongs to the same user group as the owner of a file.

BIBLIOGRAPHY

Ludwig (1998) describes different kinds of viruses, while Ludwig (2002) discusses email viruses. Spafford (1989) discusses the Morris internet worm that caused havoc in 1988, and Berghel (2001) describes the Code Red worm of 2001.

Landwehr (1981) discusses formal models for computer security. Voydock and Kent (1983) discuss security issues in distributed systems and practical techniques used to tackle them.

Shannon (1949) is the classical work in computer security. It discusses the diffusion and confusion properties of cyphers. Denning and Denning (1979) and Lempel (1979) contain good overviews of data security and cryptology, respectively. Schneier (1996), and Ferguson and Schneier (2003) are texts on cryptography, while Pfleger and Pfleger (2003) is a text on computer security. Stallings (2003) discusses cryptography and network security.

Naor and Yung (1989) discuss one-way hash functions. Rivest (1991) describes the MD4 message digest function. The goal of MD4 is to make it computationally infeasible to produce two messages with an identical message digest, or to produce a message with a given message digest. MD4 is extremely fast and resists cryptanalysis attacks successfully. Rivest (1992) describes MD5, which is more conservative and a bit slower than MD4. Preneel (1998) describes cryptographic primitives for information authentication.

Access matrix based protection is discussed in Lampson (1971) and Popek (1974). Organick (1972) discusses the MULTICS protection rings. The *setuid* feature of Unix is described in most books on Unix.

Dennis and Van Horn (1966) is a widely-referenced paper on the concept of capabilities. Levy (1984) describes a number of capability based systems. Mullender and Tanenbaum (1986) and Tanenbaum (2001) describe the software capabilities of Amoeba. Anderson *et al* (1986) discuss software capabilities with a provision for containment.

The Trusted Computer System Evaluation Criteria (TCSEC) of the US Department of Defense offer a classification of security features of computer systems. It is described in DoD (1985).

Spafford *et al* (2003) discuss security in Solaris, Mac OS, Linux, and FreeBSD operating systems. Wright *et al* (2002) discuss the linux security modules. Russinovich and Solomon (2005) discuss security features in Windows.

1. Anderson, M., R. D. Pose, and C. S. Wallace (1986): "A password-capability system," *The Computer Journal*, **29** (1), 1–8.
2. Berghel, H. (2001): "The Code Red worm," *Communications of the ACM*, **44** (12), 15–19.
3. Denning, D. E., and P. J. Denning (1979): "Data security," *Computing Surveys*, **11** (4).
4. Dennis, J. B., and E. C. Van Horn (1966): "Programming semantics for multiprogrammed computations," *Communications of the ACM*, **9** (3).

5. DoD (1985): *Trusted Computer System Evaluation Criteria*, US Department of Defense.
6. Ferguson, N. and B. Schneier (2003): *Practical Cryptography*, John Wiley & Sons.
7. Fluhrer, S., I. Mantin, and A. Shamir (2001): "Weaknesses in the key scheduling algorithm of RC4," *Proceedings of Eighth Annual Workshop on Selected Areas in Cryptography*.
8. Lampson, B. W. (1971): "Protection," *Operating Systems Review*, **8** (1), 18–24.
9. Landwehr, C. E. (1981): "Formal models for computer security," *Computing Surveys*, **13** (3), 247–278.
10. Lempel, A. (1979): "Cryptology in transition," *Computing Surveys*, **11** (4), 286–303.
11. Levy, H. M. (1984): *Capability Based Computer Systems*, Digital Press, Mass.
12. Ludwig, M. A. (1998): *The giant black book of computer viruses*, 2nd edition, American Eagle, Show Low.
13. Ludwig, M. A. (2002): *The Little Black Book of Email Viruses*, American Eagle, Show Low.
14. Menezes, A., P. van Oorschot, and S. Vanstone (1996): *Handbook of Applied Cryptography*, CRC Press.
15. Mullender, S. P., and A. Tanenbaum (1986): "The design of a capability based distributed operating system," *Computer Journal*, **29** (4).
16. Nachenberg, C. (1997): "Computer virus-anti virus coevolution," *Communications of the ACM*, **40**, 46–51.
17. Naor, M., and M. Yung (1989): "Universal one-way hash functions and their cryptographic applications," *Proceedings of the Twenty First Annual ACM Symposium on Theory of Computing*, 33–43.
18. Organick, E. I. (1972): *The MULTICS System*, MIT Press, Mass.
19. Oppliger, R. (1997): "Internet security: fire walls and beyond," *Communications of the ACM*, **40** (5), 92–102.
20. Pfleger, C. P., and S. Pfleger (2003): *Security in computing*, Prentice Hall, N.J.
21. Popek, G. J. (1974): "Protection structures," *Computer*, **7** (6), 22–33.
22. Preneel, B. (1998): *Cryptographic primitives for Information Authentication—State of the art in applied cryptography*, LNCS 1528, Springer Verlag, 1998.
23. Rivest, R. (1991): "The MD4 message digest algorithm," *Proceedings of Advances in Cryptology—Crypto'90, Lecture Notes in Computer Science, volume 537*, Springer-Verlag, 303–311.
24. Rivest, R. (1992): "The MD5 Message digest algorithm," Request for comments, RFC 1321.
25. Russinovich, M. E., and D. A. Solomon (2005): *Microsoft Windows Internals, Fourth edition*, Microsoft Press, Redmond.
26. Schneier, B. (1996): *Applied cryptography*, 2nd edition, John Wiley.
27. Shannon, C. E. (1949): "Communication Theory of Secrecy Systems," *Bell Systems Journal*, October 1949.
28. Spafford, E. H. (1989): "The internet worm: crisis and aftermath," *Communications of the ACM*, **32** (6), 678–687.
29. Spafford, G., S. Garfinkel, and A. Schwartz (2003): *Practical UNIX and Internet Security*, Third edition, O'Reilly, Sebastopol.

30. Stallings, W. (2003): *Cryptography and Network Security: Principles and Practice*, 3rd edition, Prentice Hall, N. J.
31. Stiegler, H. G. (1979): "A structure for access control lists," *Software—Practice and Experience*, 9 (10), 813–819.
32. Tanenbaum, A. S. (2001): *Modern Operating Systems*, Second edition, Prentice-Hall, Englewood Cliffs.
33. Voydock, V. L., and S. T. Kent (1983): "Security mechanisms in high level network protocols," *Computing Surveys*, 15 (2), 135–171.
34. Wofsey, M. M. (1983): *Advances in Computer Security Management*, John Wiley, New York.
35. Wright, C., C. Cowan, S. Smalley, J. Morris, and G. Kroah-hartman (2002): "Linux Security modules: General security support for the Linux kernel," *Eleventh USENIX Security Symposium*.

Part III

ADVANCED TOPICS

Part I discussed fundamental concepts in an operating system—how the OS permits a programmer to create processes within an application, how processes operate and are scheduled to use the CPU, how they are allocated memory and how they can perform file processing. In Part II, we delve deeper into *how* different activities are implemented in an operating system; we consider both activities within a process and those within the operating system itself.

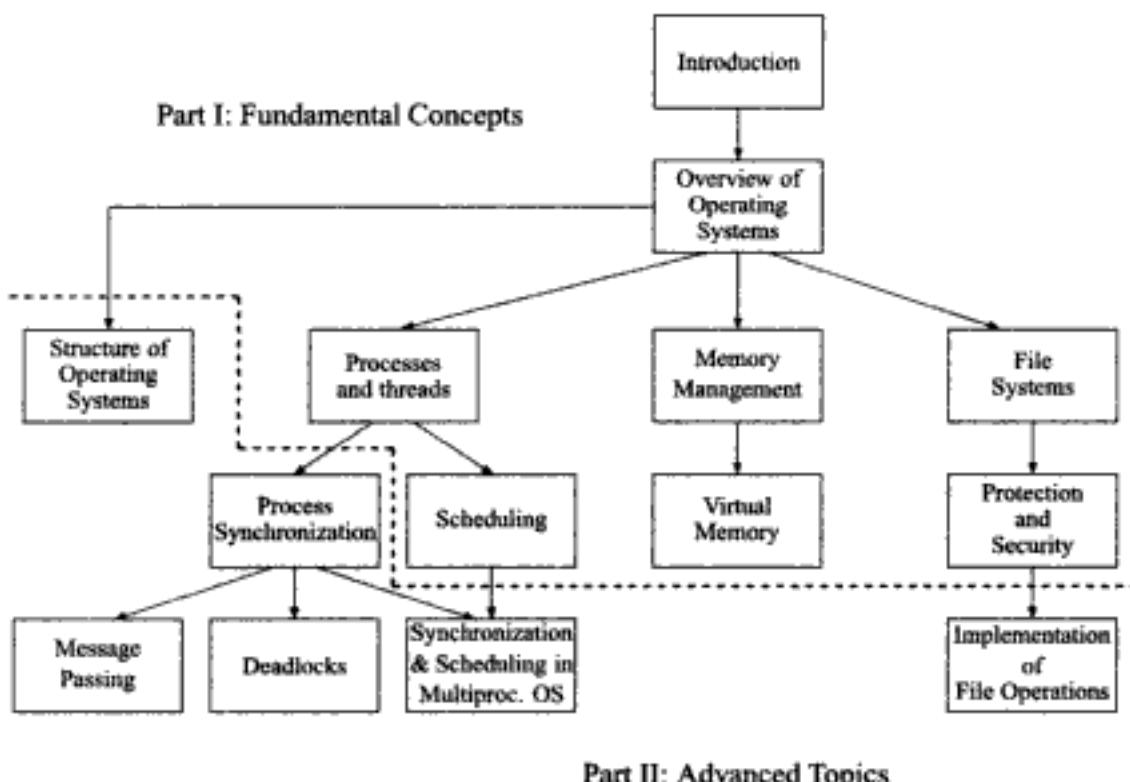
Processes within an application interact among themselves to perform their tasks harmoniously—they interact to share data and coordinate their activities, and to exchange information. *Process synchronization* is the act of implementing process interactions by making processes wait for each other's actions. *Deadlock* is a situation in which some processes wait for one another indefinitely due to faulty synchronization. A deadlock affects the performance of an application and of the operating system, so a programmer must ensure an absence of deadlock in his application and an operating system must ensure an absence of deadlocks during actions such as resource allocation.

Computer users and system administrators have diverse expectations from a file system. Apart from the convenience, protection and reliability features discussed in Part I, a computer user expects a file system to implement file operations efficiently and a system administrator expects a file system to ensure efficient use of I/O devices. The file system uses a module called the *Input-output control system* (IOCS) to achieve both these functions.

Several changes in computer technology and user requirements occur during the lifetime of an OS. The operating system has to adapt to these changes. Accordingly, the structure of an operating system should facilitate changes to implement an operating system on a computer system with a different architecture, and to provide new functionalities required by its users.

In this part, we discuss process synchronization facilities provided in operating systems, the deadlock handling methods used by operating systems, efficiency techniques used in an IOCS, and the OS design techniques that facilitate easy modification of an OS to adapt to changes in computer architecture and user requirements.

Road Map for Part II



Chapter 9: Process Synchronization

Processes of an application work towards a common goal by sharing some data and coordinating with one another. The key concepts in process synchronization are the use of *mutual exclusion* to safeguard consistency of shared data, and the use of *indivisible operations* in coordinating activities of processes. This chapter discusses some classic problems in process synchronization, and discusses how their synchronization requirements can be met using synchronization features such as *semaphores* and *monitors* provided in programming languages and operating systems.

Chapter 10: Message Passing

Processes exchange information by sending *interprocess messages*. This chapter discusses the semantics of message passing, and OS responsibilities in buffering and delivering interprocess messages.

Chapter 11: Deadlocks

A deadlock arises when processes wait for one another indefinitely due to resource sharing or synchronization. This chapter discusses how deadlocks can arise and how an OS performs *deadlock handling* to ensure an absence of deadlocks, either

through *detection* and *resolution* of deadlocks, or through resource allocation policies that perform *deadlock prevention* or *deadlock avoidance*.

Chapter 12: Implementation of File Operations

This chapter discusses the physical organization used in file systems. It starts with an overview of I/O devices and their characteristics, and discusses different *RAID organizations* which provide high reliability, fast access and high data transfer rates. The arrangements used to implement device-level I/O are then discussed, including use of *device caches* to speed up I/O operations and use of *disk scheduling* policies to improve throughput of disk devices.

This chapter also discusses the techniques of *buffering* and *blocking* of data in a file to improve the efficiency of a file processing activity in a process.

Chapter 13: Synchronization and Scheduling in Multiprocessor OSs

A multiprocessor computer system holds the promise of high throughput and computation speed-up by scheduling processes on all its CPUs. This chapter discusses how this promise is realized by using special synchronization techniques to reduce synchronization delays and special scheduling techniques to achieve computation speed-up.

Chapter 14: Structure of Operating Systems

The structure of an operating system has two kinds of features—those that contribute to simplicity of coding and efficiency of operation; and those that contribute to the ease with which an OS can be implemented on different computer systems, or can be enhanced to incorporate new functionalities. This chapter discusses three methods of structuring an operating system. The *layered structure* of operating systems simplifies coding, the *kernel-based structure* provides ease of implementation on different computer systems and the *microkernel-based structure* permits enhancement of features of an operating system.

Process Synchronization

To fulfill a common goal, interacting processes need to share data or coordinate their activities with each other. Data access synchronization ensures that shared data do not lose consistency when they are updated by several processes. It is implemented by ensuring that accesses to shared data are performed in a mutually exclusive manner. Control synchronization ensures that interacting processes perform their actions in a desired order. Computer systems provide *indivisible instructions* (also called *atomic instructions*) to support data access and control synchronization.

We begin this chapter with a discussion of *critical sections*, which are used to access shared data in a mutually exclusive manner. Following this discussion, we introduce some classic problems in process synchronization. These problems are abstractions of practical synchronization problems faced in various application domains. We analyze synchronization requirements of these problems and study important issues involved in their implementation.

In the remainder of the chapter, we discuss various synchronization facilities provided in programming languages and operating systems. These includes *semaphores*, *conditional critical regions*, and *monitors*. We discuss their use to fulfill the process synchronization requirements in the classic problems.

9.1 DATA ACCESS SYNCHRONIZATION AND CONTROL SYNCHRONIZATION

In Chapter 3 we discussed how implementing an application using a set of interacting processes may provide execution speed-up and improved response times. To work towards a common goal, interacting processes need to coordinate their activities with respect to one another. We defined two kinds of synchronization, called data access synchronization (see Def. 3.6) and control synchronization (see Def. 3.7) for this purpose. Table 9.1 summarizes their main features.

In Section 3.6.1, we defined a race condition as follows (see Def. 3.5): Let a_i and a_j be operations on shared data d_s performed by two processes P_i and P_j . $f_i(d_s)$

Table 9.1 Features of data access synchronization and control synchronization

Data access synchronization	Race conditions (see Def. 3.5) arise if processes access shared data in an uncoordinated manner. Race conditions are not reproducible, hence they make debugging difficult. Data access synchronization is used to access shared data in a mutually exclusive manner. It avoids race conditions and safeguards consistency of shared data.
Control synchronization	Control synchronization is needed if a process should perform some action a_i only after some other processes have executed a set of actions $\{a_j\}$ or only when a set of conditions $\{c_k\}$ hold.

represents the value of d_s resulting from changes, if any, caused by operation a_i . $f_j(d_s)$ is defined analogously. A race condition arises if the result of execution of a_i and a_j in processes P_i and P_j is other than $f_i(f_j(d_s))$ or $f_j(f_i(d_s))$. In Example 3.9, we saw how a race condition arises in the airline reservations system when processes update the value of *nextseatno* in an uncoordinated manner—its value increased by only 1 even though two processes added 1 to it!

We also discussed some examples of control synchronization in Section 3.6.2. The real time system of Figure 3.3 uses three processes to copy samples received from a satellite onto a disk. Here, the main process waits for its child processes to complete before terminating itself. The child processes synchronize their activities such that a new sample is copied into a buffer entry only when the buffer entry is empty and contents of a buffer entry are copied to the disk only when the buffer entry contains a new sample.

Implementing synchronization The basic technique used to implement synchronization is to block a process until an appropriate action is performed by another process or until a condition is fulfilled. Thus, data access synchronization is implemented by blocking a process until another process finishes accessing shared data. Control synchronization is implemented by blocking a process until another process performs a specific action.

While implementing process synchronization, it is important to note that the effective execution speed of a process cannot be estimated due to factors like time slicing, process priorities and I/O activities in a process. It is similarly impossible to know the relative execution speeds of processes, so a synchronization scheme must function correctly irrespective of the relative execution speeds of processes.

9.2 CRITICAL SECTIONS

Race conditions on shared data d_s (see Def. 3.5) arise because operations on d_s are performed concurrently by two or more processes. The notion of a *critical section*

(CS) is introduced to avoid race conditions.

Definition 9.1 (Critical Section (CS)) A critical section for a data item d_s is a section of code that should not be executed concurrently either with itself or with other critical section(s) for d_s .

If some process P_i is executing a critical section for d_s , another process wishing to execute a critical section for d_s will have to wait until P_i finishes executing its critical section. Thus, a CS for a data item d_s is a mutual exclusion region with respect to accesses to d_s —at most one process can execute a CS for d_s at any time.

Race conditions on a data item are avoided by performing all its update operations inside a CS for the data item. Further, to ensure that processes see consistent values of a data item, all its uses should also occur inside a CS for the data item.

We mark a CS in a piece of code by a dashed rectangular box. Note that processes may share a single copy of code, in which case a single CS for d_s may exist in an application. Alternatively, the code for each process may contain one or more CSs for d_s . Definition 9.1 covers both situations. A process that is executing a CS is said to be ‘in a CS’. We also use the terms ‘enter a CS’ and ‘exit a CS’ for situations where a process begins and ends the execution of a CS.

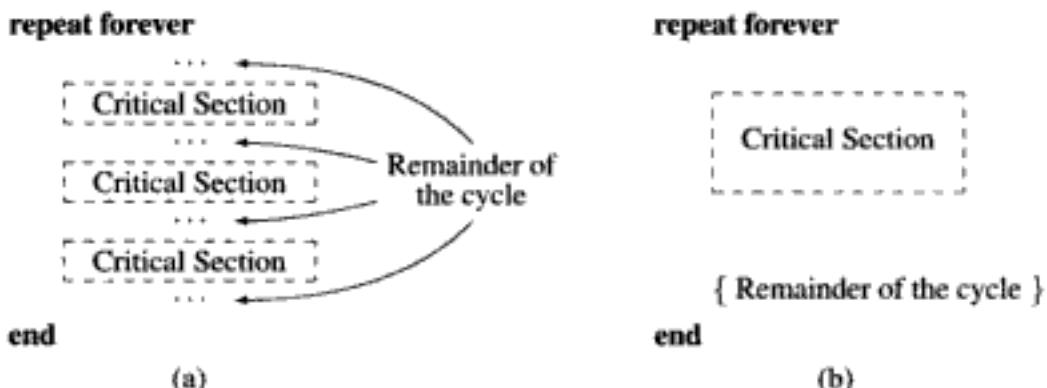


Fig. 9.1 (a) A process with many critical sections, (b) A simpler way of depicting this process

Figure 9.1(a) shows a process containing several critical sections. These critical sections may be used to access the same or different shared data items. Thus, a process may contain several critical sections for the same data item. The process contains a cycle. In each iteration, it enters a critical section when it needs to access a shared data item. At other times, it executes other code in its logic, which together constitutes “remainder of the cycle”. For simplicity, whenever possible, we use the simple process form shown in Figure 9.1(b) to depict a process.

Example 9.1 illustrates use of a CS to avoid race conditions in the airline reservation system.

Example 9.1 Figure 9.2 shows use of critical sections in the airline reservation system of Figure 3.22. Each process contains a CS in which it accesses and updates

the shared variable *nextseatno*. Absence of race conditions can be shown as follows: Let $f_i(\text{nextseatno})$ and $f_j(\text{nextseatno})$ represent the value of *nextseatno* resulting from changes, if any, caused by execution of critical sections in P_i and P_j , respectively. Let P_i and P_j attempt to execute their critical sections concurrently. From the definition of CS, it follows that only one of them can execute its CS at any time, so the resulting value of *nextseatno* will be either $f_i(f_j(\text{nextseatno}))$ or $f_j(f_i(\text{nextseatno}))$. From Def. 3.5, a race condition does not arise.

<pre> if nextseatno < capacity then allottedno:=nextseatno; nextseatno:=nextseatno+1; else display "sorry, no seats available"; </pre>	<pre> if nextseatno < capacity then allottedno:=nextseatno; nextseatno:=nextseatno+1; else display "sorry, no seats available"; </pre>
<i>process P_i</i>	<i>process P_j</i>

Fig. 9.2 Use of CS in airline reservation

If processes in an application make frequent use of a shared data item d_s , execution of critical sections for d_s may become a performance bottleneck. This would cause delays in the execution of processes, and would adversely affect performance of the application. The severity of this problem is reduced if processes do not spend too much time inside a CS. Both processes and the kernel must cooperate to ensure this. A process must not execute for too long inside a CS. While in a CS, it must avoid making system calls that might put it in a *blocked* state. The kernel must not preempt a process that is engaged in executing a CS. This condition is hard to meet as it requires the kernel to know whether a process is inside a CS at any moment. It cannot be met if processes implement critical sections on their own, i.e., without involving the kernel. Nevertheless, in this Chapter we will assume that a process spends only a short time inside a critical section.

9.2.1 Properties of a CS Implementation

A CS implementation for a data item d_s is like a scheduler for a resource. It must keep track of all processes that wish to enter a CS for d_s and select a process for entry in a CS in accordance with the notions of mutual exclusion, efficiency and fairness. Table 9.2 summarizes the essential properties an implementation of CS must possess. We discuss these properties in the following.

Mutual exclusion guarantees that more than one process would not be in a CS for d_s at any time. Apart from correctness, a CS implementation should also guarantee that any process wishing to enter a CS would not be delayed indefinitely, i.e., *starvation* would not occur. This guarantee is in two parts, and is provided by the other two properties.

The *progress* property ensures that if some processes are interested in entering

Table 9.2 Essential properties of a CS implementation

Property	Description
Correctness	At any moment, at most one process may execute a CS for a data item d_s .
Progress	When a CS is not in use, one of the processes wishing to enter it will be granted entry to the CS.
Bounded wait	After a process P_i has indicated its desire to enter a CS for d_s , the number of times other processes can gain entry to a CS for d_s ahead of P_i is bounded by a finite integer.

a CS for a data item d_s , one of them will be granted entry if no process is currently inside a CS for d_s —that is, the privilege to enter a CS for d_s will not be reserved for some process that is not interested in entering a CS at present. If this property were not satisfied, processes wishing to use a CS may be delayed until some specific process P_s decides to use a CS. Process P_s may never use a CS for d_s , so absence of this property might delay a process indefinitely.

The progress property ensures that the CS implementation will necessarily choose one of the requesting processes to enter a CS. However, even this is not enough to ensure that a requesting process P_i gains entry to a CS in finite time because a CS implementation may ignore P_i while choosing a process for entry to a CS. The *bounded wait* property ensures that this does not happen by limiting the number of times other processes can be favored over P_i . Since execution of a CS takes a short time, this property ensures that every requesting process will gain entry to its CS in finite time.

9.2.2 The Problem of Busy Wait

A process may implement a CS for a data item d_s using the following simple code:

```

while (some process is in a CS for  $d_s$ )
    { do nothing }
    Critical
    section

```

In the **while** loop, the process checks if some other process is in a CS for the same data item. If so, it keeps looping until the other process exits its CS.

A *busy wait* is a situation in which a process repeatedly checks if a condition that would enable it to get past a synchronization point is satisfied. It ends only when the condition is satisfied. Thus a busy wait keeps the CPU busy in executing a process

even as the process does nothing! Lower priority processes are denied use of the CPU, so their response times suffer. System performance also suffers.

A busy wait also causes a curious problem in a uniprocessor system. Consider the following situation: A high priority process P_i is blocked on an I/O operation and a low priority process P_j enters a CS for data item d_s . When P_i 's I/O operation completes, P_j is preempted and P_i is scheduled. If P_i now tries to enter a CS for d_s using the **while** loop described earlier, it would face a busy wait. This busy wait denies the CPU to P_j , hence it is unable to complete its execution of the CS and exit. This prevents P_i from entering its CS. Processes P_i and P_j now wait for each other indefinitely. This situation is called a *deadlock*. Because a high priority process waits for a process with a low priority, this situation is also called *priority inversion*. The priority inversion problem is addressed using the *priority inheritance protocol*, wherein a low priority process that holds a resource temporarily acquires the priority of the highest priority process that needs the resource. In our example, process P_j would temporarily acquire the priority of process P_i , which would enable it to get scheduled and exit from its critical section. However, use of the priority inheritance protocol in these situations is impractical because it would require the kernel to note minute details of the operation of processes.

To avoid busy waits, a process waiting for entry to a CS should be put into the *blocked* state. Its state should be changed to *ready* only when it can be allowed to enter its CS.

9.2.3 History of CS Implementations

Historically, implementation of critical sections has gone through three important stages—algorithmic approaches, software primitives and concurrent programming constructs. *Algorithmic approaches* depended on a complex arrangement of checks to ensure mutual exclusion between processes using a CS. Correctness of a CS implementation depended on correctness of these checks, and was hard to prove due to the logical complexity of the checks. This was a serious deficiency in the development of large systems containing concurrent processes. However, a major advantage of this approach was that no special hardware, programming language or kernel features were required to implement a CS.

A set of *software primitives* for mutual exclusion (e.g., the P and V primitives of Dijkstra) were developed to overcome the logical complexity of algorithmic implementations. These primitives were implemented using some special architectural features of a computer system, and possessed useful properties for implementing a CS. These properties could also be used to construct proofs of correctness of a concurrent system. However, experience with such primitives showed that ease of use and proof of correctness still remained major obstacles in the development of large concurrent systems.

Development of *concurrent programming constructs* was the next important step in the history of CS implementations. These constructs used data abstraction and

encapsulation features specifically suited to the construction of concurrent programs. They had well defined semantics that were enforced by the language compiler. This feature made construction of large concurrent systems more practical.

We discuss algorithmic approaches to CS implementation in Section 9.7. Sections 9.8–9.9 discuss programming language primitives and constructs. Section 9.10 discusses the abstraction and encapsulation facilities for concurrent programming provided by monitors.

9.3 RACE CONDITIONS IN CONTROL SYNCHRONIZATION

Processes use control synchronization to coordinate their activities with respect to one another. A frequent requirement in process synchronization is that a process P_i should perform an action a_i only after process P_j has performed some action a_j . A pseudo-code for such processes is shown in Figure 9.3. This synchronization requirement is met using the technique of *signaling*.

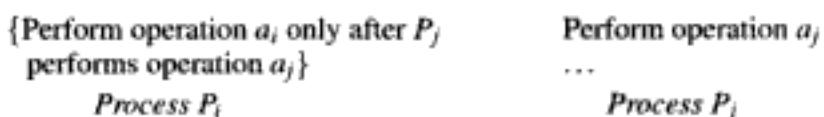


Fig. 9.3 Processes requiring control synchronization

Signaling Figure 9.4 shows how signaling is performed. The synchronization data consists of two boolean variables. Variable *action_aj_performed* is a flag to indicate whether process P_j has performed action a_j . Variable *pi_blocked* is a flag to indicate whether process P_i has blocked itself pending execution of action a_j by process P_j . It is assumed that the kernel supports system calls to block and activate a process.

Process P_i consults variable *action_aj_performed* to check if process P_j has already performed action a_j . If this is not the case, it sets *pi_blocked* to *true* and makes a system call to block itself. It goes ahead to perform action a_i if a_j has already been performed. Process P_j performs action a_j and checks if process P_i has blocked itself before performing action a_j . If so, it makes a system call to activate P_i . Otherwise, it sets *action_aj_performed* to *true* so that process P_i will know that P_j has performed action a_j .

Table 9.3 shows an execution of the system in which process P_i faces indefinite blocking due to a race condition. Process P_i checks the value of *action_aj_performed* and finds that action a_j has not been performed. It is poised to set the variable *pi_blocked* to *true* when it gets preempted. Process P_j is now scheduled. It performs action a_j and checks whether process P_i is blocked. Since *pi_blocked* is *false*, it simply sets *action_aj_performed* to *true* and continues its execution. Sometime later, P_i is scheduled. It sets *pi_blocked* to *true* and makes a system call to block itself. Process P_i now sleeps for ever!

Consider the **if** statements in processes P_i and P_j as the operations f_i and f_j on

```

var
    operation.aj_performed : boolean;
    pi_blocked : boolean;
begin
    operation.aj_performed := false;
    pi_blocked := false;

Parbegin
    ...
    if operation.aj_performed = false
    then
        pi_blocked := true;
        block(Pi);
        {perform operation ai}
    ...
    ...
    ...
Parend;
end.

process Pi
process Pj

```

Fig. 9.4 An attempt at signaling through boolean variables

Table 9.3 Race condition in process synchronization

Time	Actions of process P _i	Actions of process P _j
<i>t</i> ₁	if <i>action.aj_performed</i> = <i>false</i> then	
<i>t</i> ₂		{perform action a _j }
<i>t</i> ₃		if <i>pi_blocked</i> = <i>true</i> then
<i>t</i> ₄		<i>action.aj_performed</i> := <i>true</i>
:		
<i>t</i> ₂₀	<i>pi_blocked</i> := <i>true</i> ;	
<i>t</i> ₂₁	<i>block</i> (P _i);	

the state of the system. The result of their execution should have been one of the following: process P_i blocks itself, gets activated by P_j and performs action a_i ; or process P_i finds that P_j has already performed a_j and goes ahead to perform action a_i . However, as seen in Table 9.3, process P_i blocks itself and does not get activated. From Def. 3.5, this is a race condition.

The race condition arises because process P_i is preempted after it checks if $action.aj_performed = true$, but before it can set $pi_blocked = true$. The race condition can be avoided if we can guarantee that P_i would be able to complete both these actions

before getting preempted. We introduce the notion of an indivisible operation (also called an atomic operation) to achieve this.

Definition 9.2 (Indivisible operation) An indivisible operation on a set of data items $\{d_s\}$ is an operation that cannot be executed concurrently with any other operation involving a data item included in $\{d_s\}$.

The race condition shown in Table 9.3 would not arise if the if statements in Figure 9.4 are implemented as indivisible operations, because if process P_i finds $action_aj_performed = false$, it would be able to set $pi_blocked = true$ without getting preempted. We could achieve this by defining two indivisible operations *check_action_aj_performed* and *post_action_aj_performed* to perform the if statements of processes P_i and P_j , respectively. Figure 9.5 shows a signaling arrangement using these indivisible operations. Figure 9.6 shows details of the indivisible operations *check_action_aj_performed* and *post_action_aj_performed*. When *action_aj_performed* is *false*, indivisible operation *check_aj* is deemed to be complete after process P_i is blocked; it would enable process P_j to perform operation *post_aj*.

```

var
    action_aj_performed : boolean;
    pi_blocked : boolean;

begin
    action_aj_performed := false;
    pi_blocked := false;

Parbegin
    ...
    check_aj;
    {perform action ai}
    ...
    {perform action aj}
    post_aj;
Parend:
end.

process Pi process Pj

```

Fig. 9.5 Control synchronization by signaling using indivisible operations

An indivisible operation on $\{d_s\}$ is like a critical section on $\{d_s\}$. However, we differentiate between them because a CS has to be explicitly implemented in a program, whereas the hardware or software of a computer system may provide indivisible operations as its primitive operations. In the next Section we discuss how indivisible operations on semaphores can be used to implement data access synchronization and control synchronization without race conditions.

9.4 IMPLEMENTING CRITICAL SECTIONS AND INDIVISIBLE OPERATIONS

Process synchronization requires critical sections or signaling operations that are indivisible. These are implemented using indivisible instructions provided by com-

```

procedure check_aj
begin
  if action_aj_performed = false
  then
    pi_blocked := true;
    block (Pi)
  end;

procedure post_aj
begin
  if pi_blocked = true
  then
    pi_blocked := false;
    activate(Pi)
  else
    action_aj_performed := true;
  end;

```

Fig. 9.6 Indivisible operations for signaling

puter systems, and lock variables.

Indivisible instructions In Section 3.6.1 we saw how race conditions can arise if processes execute a load-add-store instruction sequence on shared data. If a computer system contains more than one CPU, race conditions can arise even during execution of a single instruction that makes more than one access to a memory location, for example, an instruction that increments the value of a variable.

Since mid 1960's, computer systems have provided special features in their architecture to avoid race conditions while accessing a memory location containing shared data. The basic theme is that all accesses to a memory location made by one instruction should be implemented without permitting another CPU to access the same location. Two popular techniques used for this purpose are locking the memory bus (e.g., in Intel 80x86 processors) and providing special instructions that avoid race conditions (e.g., in IBM/370 and M68000 processors). We will use the term *indivisible instruction* as a generic term for all such features.

Use of a lock variable A *lock variable* is used to bridge the gap between critical sections or indivisible operations, and indivisible instructions provided in a computer system. The basic idea is to close a lock at the start of a critical section or an indivisible operation and open it at the end of the critical section or the indivisible operation. A close-the-lock operation fails if the lock is already closed by another process. This way only one process can execute a critical section or an indivisible operation. Race conditions on the lock variable are avoided by using indivisible instructions to access the lock variable.

A process that fails to close a lock must retry the operation. This scheme involves

a busy wait; however, it does not last for long—it lasts only until the process that has closed the lock finishes executing a critical section or an indivisible operation, both of which require only a short time.

Figure 9.7 illustrates how a critical section is implemented using an indivisible instruction and a lock variable. The indivisible instruction performs the actions indicated in the dashed box—it tests the value of the lock and loops back to itself if the lock is *closed*, else it closes the lock. We illustrate use of two indivisible instructions—called test-and-set and swap instructions—to implement critical sections and indivisible operations in the following.

<code>entry_test:</code> <code> if lock = closed</code> <code> then goto entry_test;</code> <code> lock := closed;</code> <code>{ Critical section or</code> <code> indivisible operation }</code> <code>lock := open;</code>	Performed by an indivisible instruction
---	---

Fig. 9.7 Implementing a critical section or indivisible operation using a lock variable

Test-and-set (TS) instruction This indivisible instruction in the IBM/370 performs two actions. It ‘tests’ the value of a memory byte such that the condition code indicates whether the value was zero or nonzero. It also sets all bits in the byte to 1s. No other CPU can access the memory byte until both actions are complete. This instruction can be used to implement the statements enclosed in the dashed box in Figure 9.7.

```

LOCK      DC      X'00'      Lock is initialized to open
ENTRY_TEST TS      LOCK      Test-and-set lock
            BC    7, ENTRY_TEST Loop if lock was closed

            ...
            { Critical section or
            indivisible operation }

NVI      LOCK, X'00'      Open the lock (by moving 0s)

```

Fig. 9.8 Implementing a critical section or indivisible operation using test-and-set

Figure 9.8 is an IBM/370 assembly language program that shows how indivisibility of *wait* and *signal* operations is achieved. The first line in the assembly language program declares variable `LOCK` and initializes it to hexadecimal 0. `LOCK` is used as a lock variable with the convention that a non-zero value in `LOCK` implies that the lock is *closed*, and a zero value implies that it is *open*. The `TS` instruction sets the condition code according to the value of `LOCK` and also changes its value to *closed*. The condition code indicates whether the value of `lock` was *closed* before the instruction was executed. The branch instruction `BC 7, TEST` checks the condition code

and loops back to the TS instruction if the lock was *closed*. This way a program (or programs) that find the lock to be *closed* execute the loop in a busy wait until lock is set to *open*. The MVI instruction puts 0s in all bits of LOCK, i.e., it opens the lock. TS is an indivisible instruction, so opening the lock would enable only one process looping at TEST to proceed.

Swap instruction The swap instruction exchanges contents of two memory locations. It is an indivisible instruction, hence no other CPU can access any of the locations during swapping. Figure 9.9 shows how a critical section or an indivisible operation can be implemented using the swap instruction. (For convenience, we use the same coding conventions as used for the TS instruction.) The temporary location TEMP is set to a non-zero value and its contents are swapped with LOCK. This action closes the lock. Old value of the lock is now available in TEMP. It is tested to see if the lock was already closed. If so, the process loops on the swap-and-compare action until LOCK is set to *open*. The process executing the critical section or indivisible operation opens the lock at the end of the operation. This action enables one process to get past the BC instruction.

TEMP	DS	1	Reserve one byte for TEMP
LOCK	DC	X'00'	Lock is initialized to open
	MVI	TEMP, X'FF'	X'FF' is used to close the lock
ENTRY.TEST	SWAP	LOCK, TEMP	
	COMP	TEMP, X'00'	Test old value of lock
	BC	7, ENTRY.TEST	Loop if lock was closed
			{ Critical section or indivisible operation }
	MVI	LOCK, X'00'	Open the lock

Fig. 9.9 Implementing a critical section or indivisible operation using a swap instruction

9.5 CLASSIC PROCESS SYNCHRONIZATION PROBLEMS

As discussed in Sections 9.2 and 9.3, critical sections and signaling are the key elements of process synchronization. A solution to a process synchronization problem should use a suitable combination of these elements. It must also possess three important properties:

- Correctness
- Maximum concurrency
- No busy waits.

Correctness criteria depend on the nature of a problem. These criteria project requirements concerning data access synchronization and control synchronization of interacting processes. Maximum concurrency within an application is needed to provide execution speed-up and good response times. To achieve it, processes should be

able to execute freely when they are not blocked due to synchronization requirements. As discussed in Section 9.2.2, busy waits are undesirable because they lead to degradation of system performance and response times, so synchronization should be performed by blocking a process rather than through busy waits.

In this Section, we analyze some classic problems in process synchronization and discuss issues (and common mistakes) in designing their solutions. In later Sections we implement their solutions using various synchronization features provided in programming languages.

9.5.1 Producers–Consumers With Bounded Buffers

A producers–consumers system with bounded buffers consists of an unspecified number of producer and consumer processes and a finite pool of buffers (see Figure 9.10). Each buffer is capable of holding one record of information—it is said to become *full* when a producer writes into it, and *empty* when a consumer copies out a record contained in it; it is empty to start with. A producer process produces one record at a time and writes it into the buffer. A consumer process consumes information one record at a time.

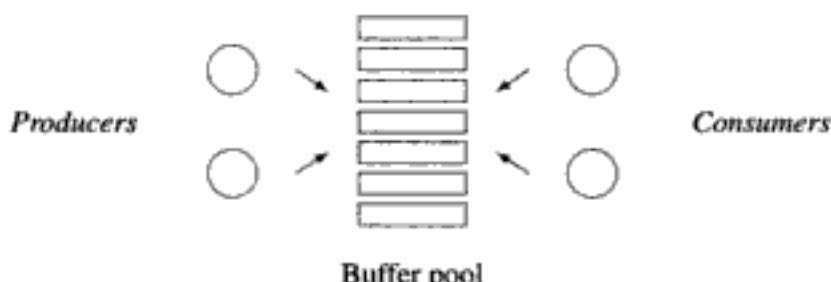


Fig. 9.10 Producers and consumers

A producers–consumers system with bounded buffers is a useful abstraction for many practical synchronization problems. A print service is a good example in the OS domain. A print daemon is a consumer process. A fixed sized queue of print requests is the bounded buffer. A process that adds a print request to the queue is a producer process.

A solution to the producers–consumers problem must satisfy following conditions:

1. A producer must not overwrite a full buffer.
2. A consumer must not consume an empty buffer.
3. Producers and consumers must access buffers in a mutually exclusive manner.

The following condition is sometimes imposed:

4. Information must be consumed in the same order in which it is put into the buffers, i.e., in FIFO order.

```

begin
Parbegin
    var produced : boolean;
repeat
    produced := false;
    while produced = false
        if an empty buffer exists
        then
            { Produce in a buffer }
            produced := true;
        { Remainder of
        the cycle }
    forever;
Parend
end.

Producer

```



```

var consumed : boolean;
repeat
    consumed := false ;
    while consumed = false
        if a full buffer exists
        then
            { Consume a buffer }
            consumed := true;
        { Remainder of
        the cycle }
    forever;

```


Consumer

Fig. 9.11 Solution outline for producers-consumers containing busy waits

Figure 9.11 shows an outline for the producers-consumers problem. Producer and consumer processes access a buffer inside a critical section. A producer enters its CS and checks to see whether an empty buffer exists. If so, it produces into that buffer, else it merely exits from its CS. This sequence is repeated until it finds an empty buffer. The boolean variable *produced* is used to break out of the **while** loop after the producer produces in the empty buffer. Analogously, a consumer makes repeated checks until it finds a full buffer to consume from.

An undesirable aspect of this outline is the presence of busy waits. To avoid a busy wait, a producer process should check for existence of an empty buffer only once. If none exists, the producer should be blocked until an empty buffer becomes available. When a consumer consumes from a buffer, it should activate a producer that is waiting for an empty buffer. Similarly, a producer must activate a consumer waiting for a full buffer. A CS cannot provide these facilities hence we must look elsewhere for a solution to the problem of busy waits.

When we re-analyze the producers-consumers problem in this light, we notice that though it involves mutual exclusion between a producer and a consumer using the same buffer, it is really a signaling problem. After producing a record in a buffer, a producer should signal a consumer that wishes to consume the record from the same buffer. Similarly, after consuming a record in a buffer, a consumer should signal a producer that wishes to produce a record in that buffer. These requirements can be met using the signaling arrangement shown in Figure 9.5.

An improved outline using this approach is shown in Figure 9.12 for a simple producers-consumers system that consists of a single producer, a single consumer and a single buffer. The operation *check_b_empty* performed by the producer blocks it if the buffer is full, while the operation *post_b_full* sets *buffer_full* to *true* and activates

```

var
    buffer : ...;
    buffer_full : boolean;
    producer_blocked, consumer_blocked : boolean;
begin
    buffer_full := false;
    producer_blocked := false;
    consumer_blocked := false;
Parbegin
    repeat
        check_b_empty;
        { Produce in the buffer }
        post_b_full;
        { Remainder of the cycle }
    forever;
Parend:
end.
Producer
repeat
    check_b_full;
    { Consume from the buffer }
    post_b_empty;
    { Remainder of the cycle }
forever;
Consumer

```

Fig. 9.12 An improved outline for a single buffer producers-consumers system using signaling

the consumer if the consumer is blocked for the buffer to become full. Analogous operations *check_b_full* and *post_b_empty* are defined for use by the consumer process. The boolean flags *producer_blocked* and *consumer_blocked* are used by these operations to note if the producer or consumer process is blocked at any moment. Figure 9.13 shows details of the indivisible operations. This outline will need to be extended to handle multiple buffers or multiple producer/consumer processes. We discuss this aspect in Section 9.8.2.

9.5.2 Readers and Writers

A readers-writers system consists of a set of processes using some shared data. A process that only reads the data is a *reader*; a process that modifies or updates it is a *writer*. We use the terms *reading* and *writing* for accesses to the shared data made by reader and writer processes, respectively. The correctness conditions for the readers-writers problem are as follows:

1. Many readers can perform reading concurrently.
 2. Reading is prohibited while a writer is writing.
 3. Only one writer can perform writing at any time.
- Conditions 1–3 do not specify which process should be preferred if a reader and a writer process wish to access the shared data at the same time. The following additional condition is imposed if it is important to give a higher priority to readers in order to meet some business goals:

```

procedure check_b_empty
begin
  if buffer_full = true
  then
    producer_blocked := true;
    block (producer);
end;

procedure post_b_full
begin
  buffer_full := true;
  if consumer_blocked = true
  then
    consumer_blocked := false;
    activate (consumer);
end;

```

Operations of producer


```

procedure check_b_full
begin
  if buffer_full = false
  then
    consumer_blocked := true;
    block (consumer);
end;

procedure post_b_empty
begin
  buffer_full := false;
  if producer_blocked = true
  then
    producer_blocked := false;
    activate (producer);
end;

```

Operations of consumer

Fig. 9.13 Indivisible operations for the producers-consumers problem

4. A reader has a non-preemptive priority over writers, i.e., it gets access to the shared data ahead of a waiting writer, but it does not preempt an active writer.

This system is called a *readers preferred readers-writers system*. A *writers preferred readers-writers system* is analogously defined.

Figure 9.14 illustrates an example of a readers-writers system. The readers and writers share a bank account. The reader processes *print statement* and *stat analysis* merely read the data from the bank account, hence they can execute concurrently. *credit* and *debit* modify the balance in the account. Clearly only one of them should be active at any moment and none of the readers should be active when they modify the data.

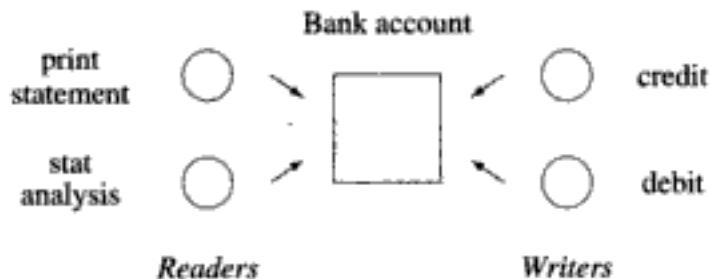


Fig. 9.14 Readers and writers in a banking system

We determine the synchronization requirements of a readers-writers system by analyzing its correctness conditions. Condition 3 requires that a writer should perform writing in a critical section. When it finishes writing, it should activate one

waiting writer or activate all waiting readers. This can be achieved using a signaling arrangement. From condition 1, concurrent reading is permitted. We should maintain a count of readers reading concurrently. When the last reader finishes reading, it should activate a waiting writer.

Figure 9.15 contains an outline for a readers-writers system. Writing is performed in a CS. A CS is not used in a reader as that would prevent concurrency between readers. A signaling arrangement is used to handle blocking and activation of readers and writers. It is complicated by the fact that a writer may have to signal many readers. So we do not specify its details in the outline; we shall discuss it in Section 9.8.3. The outline of Figure 9.15 does not satisfy the bounded wait condition for both readers and writers, however it provides maximum concurrency.

Parbegin repeat If a writer is writing then { wait }; { read } If writer(s) waiting then <i>activate a writer if no other readers reading</i> forever; Parend	repeat If reader(s) are reading, or a writer is writing then { wait }; { write } If reader(s) or writer(s) waiting then <i>activate all readers or activate one writer</i> forever;
<u>Reader(s)</u>	<u>Writer(s)</u>

Fig. 9.15 Solution outline for readers-writers without writer priority

9.5.3 Dining Philosophers

Five philosophers sit around a table pondering philosophical issues. A plate of spaghetti is kept in front of each philosopher, and a fork is placed between each pair of philosophers (see Figure 9.16). To eat, a philosopher must pick up the two forks placed between him and his immediate neighbors on either side, one at a time. The problem is to design processes to represent the philosophers such that each philosopher can eat when hungry and none dies of hunger.

The correctness condition in the dining philosophers system is that a hungry philosopher should not face indefinite waits when he decides to eat. The challenge is to design a solution that does not suffer from either *deadlocks*, where processes become blocked waiting for each other, or *livelocks*, where processes are not blocked but defer to each other indefinitely. Consider the outline of a philosopher process P_i shown in Figure 9.17, where details of process synchronization have been omitted. This solution is prone to deadlock, because if all philosophers simultaneously lift their left forks, none will be able to lift the right fork! It also contains race con-

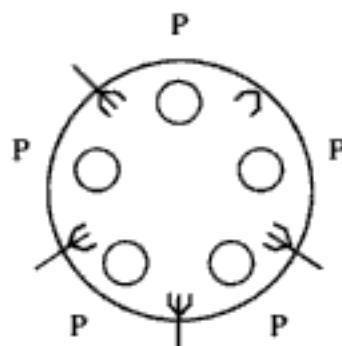


Fig. 9.16 Dining philosophers

ditions because neighbors might fight over a shared fork. We can avoid deadlocks by modifying the philosopher process so that if the right fork is not available, the philosopher would defer to his left neighbor by putting down the left fork and repeating the attempt to take the forks sometime later. However, this approach suffers from livelocks because the same situation may recur.

```

repeat
  if left fork is not available
    then
      block ( $P_i$ );
      lift left fork;
      if right fork is not available
        then
          block ( $P_i$ );
          lift right fork;
          { eat }
          if left neighbor is waiting for his right fork
          then
            activate (left neighbor);
          if right neighbor is waiting for his left fork
          then
            activate (right neighbor);
          { think }
forever

```

Fig. 9.17 Outline of a philosopher process P_i

An improved outline for the dining philosophers problem is given in Figure 9.18. A philosopher checks availability of forks in a CS and also picks up the forks in the CS. Hence race conditions cannot arise. This arrangement ensures that at least some philosopher(s) can eat at any time and deadlocks cannot arise. A philosopher who

cannot get both forks at the same time blocks himself. However, it does not avoid busy waits because the philosopher gets activated when any of his neighbors puts down a shared fork, hence he has to check for availability of forks once again. This is the purpose of the **while** loop. Some innovative solutions to the dining philosophers problem prevent deadlocks without busy waits (see Problem 20 in Exercise 9). Deadlock prevention is discussed in detail in Chapter 11.

```

repeat
    successful := false;
    while (not successful)
        if both forks are available then
            lift the forks one at a time;
            successful := true;
        if successful = false
            then
                block ( $P_i$ );
                { eat }
                put down both forks;
                if left neighbor is waiting for his right fork
                then
                    activate (left neighbor);
                if right neighbor is waiting for his left fork
                then
                    activate (right neighbor);
                { think }
            forever

```

Fig. 9.18 An improved outline of a philosopher process

9.6 STRUCTURE OF CONCURRENT SYSTEMS

A concurrent system is a system containing concurrent processes. It consists of three key components:

- Shared data
- Operations on shared data
- Processes.

Shared data can be of two kinds—data used and manipulated by processes and data defined and used for synchronization between processes. An operation is a convenient unit of code, typically a procedure in a programming language, which manipulates shared data.

A *synchronization operation* is an operation on synchronization data. Semantics of synchronization operations determine the ease, logical complexity and reliability of a concurrent system implementation. In the following Sections we introduce programming language features for synchronization and discuss semantics of the syn-

chronization operations provided by them. We also illustrate their use in concurrent systems with the help of snapshots of the system taken at different times.

Snapshot of a concurrent system A *snapshot* of a concurrent system is a view of the system at a specific time. It shows relationships between shared data, operations and processes at that moment. We use the pictorial conventions shown in Figure 9.19 to depict a snapshot. A process is shown as a circle. A circle with a cross in it indicates a blocked process. A data item is represented by a rectangular box. The value of the data item, if known, is shown inside the box. An oval shape enclosing a data item indicates that the data item is shared.

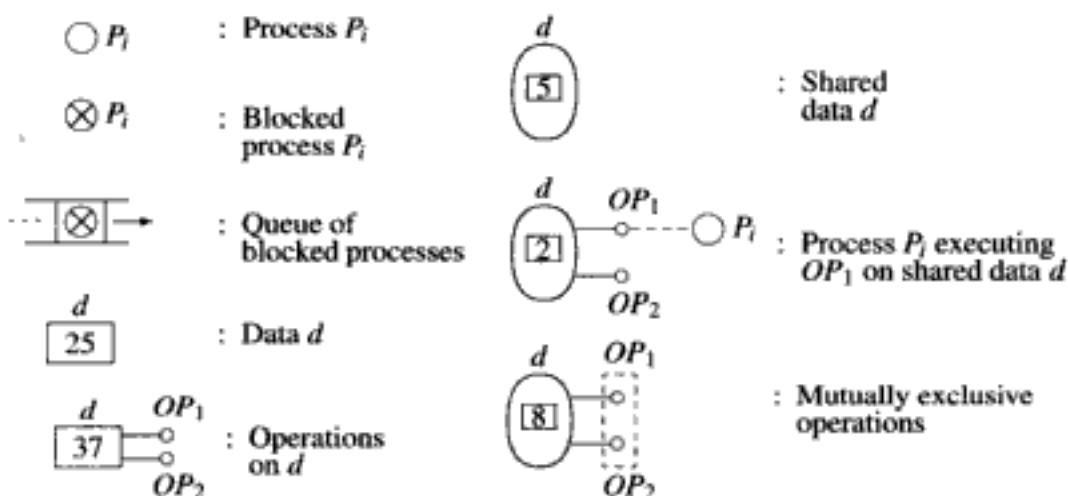


Fig. 9.19 Pictorial conventions for snapshots of concurrent systems

Operations on data are shown as connectors or sockets joined to the data. An oval shape enclosing a data item indicates that the data item is shared. A dashed line connects a process and an operation on data if the process is currently engaged in executing the operation. We have been using a dashed rectangular box to enclose code executed as a critical section. We extend this convention to operations on data. Hence mutually exclusive operations on data are enclosed in a dashed rectangular box. A queue of blocked processes is associated with the dashed box to show the processes waiting to perform one of the operations.

The execution of a concurrent system is represented by a series of snapshots.

Example 9.2 Consider the system of Figure 9.3, where process P_i performs action a_i only after process P_j performs action a_j . We assume that operations a_i and a_j operate on some shared data items X and Y , respectively. Let the system be implemented using the operations *check.aj* and *post.aj* of Figure 9.6. This system comprises of the following components:

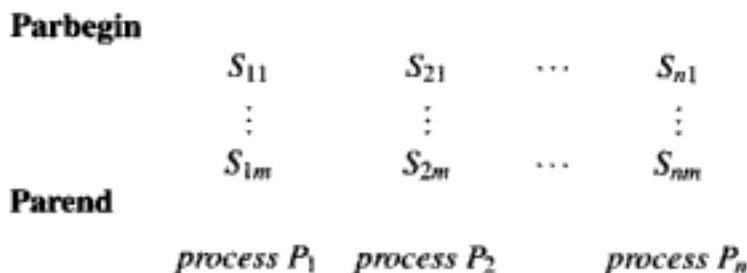
- | | |
|--------------------------------|---|
| Shared data | : Boolean variables <i>operation.aj.performed</i> and <i>pi.blocked</i> , both initialized to <i>false</i> , and data items X and Y . |
| Operations on application data | : Operations a_i and a_j . |

when synchronization involves more than two processes.

We show an algorithm for CS implementation as a pseudo-code with the following features:

1. The **Parbegin–Parend** control structure is used to enclose code that is to be executed in parallel. This control structure has the syntax **Parbegin** *<list of statements>* **Parend**. If *<list of statements>* contains n statements, execution of the **Parbegin–Parend** control structure spawns n processes, each process consisting of the execution of one statement in *<list of statements>*. For example, **Parbegin** S_1, S_2, S_3, S_4 **Parend** initiates four processes that execute S_1, S_2, S_3 and S_4 , respectively.

The statement grouping facilities of a language such as **begin–end**, can be used if a process is to consist of a block of code instead of a single statement. For visual convenience, we depict concurrent processes created in a **Parbegin–Parend** control structure as follows:



where statements $S_{11} \dots S_{1m}$ form the code of process P_1 , etc.

2. Declarations of shared variables are placed before a **Parbegin**.
3. Declarations of local variables are placed at the start of a process.
4. Comments are enclosed within braces '{ }'.
5. Indentation is used to show nesting of control structures.

We begin by discussing CS implementations for use by two processes. Later we extend some of these schemes for use by n processes.

9.7.1 Two Process Algorithms

Algorithm 9.1 First attempt

```

var      turn : 1 .. 2;
begin
    turn := 1;
Parbegin
    repeat
        while turn = 2
            do { nothing };
            { Critical Section }
    repeat
        while turn = 1
            do { nothing };
            { Critical Section }

```

```

turn := 2;
{ Remainder of
the cycle }
forever;
Parend
end.

```

Process P₁

```

turn := 1;
{ Remainder of
the cycle }
forever;

```

Process P₂

Variable *turn* is a shared variable. The notation 1 .. 2 in its declaration indicates that it takes values in the range 1–2, i.e., its value is either 1 or 2. It is initialized to 1 before processes *P*₁ and *P*₂ are created. Each process contains a CS for some shared data *d_s*. The shared variable *turn* is used to indicate which process can enter its CS next. Let process *P*₁ wish to enter its CS. If *turn* = 1, *P*₁ can enter straightaway. After exiting its CS, it sets *turn* to 2 so that *P*₂ can enter its CS. If *P*₁ finds *turn* = 2 when it wishes to enter its CS, it waits in the **while** loop until *P*₂ exits from its CS and executes the assignment *turn* := 1. Thus the correctness condition is satisfied. However, processes may encounter a busy wait before gaining entry to the CS.

Use of shared variable *turn* leads to a problem. Let process *P*₁ be in its CS and process *P*₂ be in the remainder of the cycle. If *P*₁ exits from its CS, finishes the remainder of its cycle and wishes to enter its CS once again, it would encounter a busy wait until after *P*₂ uses its CS. This situation does not violate the bounded wait condition, since *P*₁ has to wait for *P*₂ to go through its CS exactly once. However, the progress condition is violated since *P*₁ is currently the only process interested in using its CS, but it is unable to do so. Algorithm 9.2 tries to eliminate this problem.

Algorithm 9.2 Second attempt

```

var   c1, c2 : 0 .. 1;
begin
  c1 := 1;
  c2 := 1;
Parbegin
  repeat
    while c2 = 0
      do { nothing };
    c1 := 0;
    { Critical Section }
    c1 := 1;
    { Remainder of
      the cycle }
  forever;
Parend
end.

```

Process P₁

```

repeat
  while c1 = 0
    do { nothing };
  c2 := 0;
  { Critical Section }
  c2 := 1;
  { Remainder of
    the cycle }
forever;

```

Process P₂

Variable *turn* of Algorithm 9.1 has been replaced by two shared variables c_1 and c_2 . These variables can be looked upon as status flags for processes P_1 and P_2 , respectively. P_1 sets c_1 to 0 while entering its CS, and sets it back to 1 after exiting from its CS. Thus $c_1 = 0$ indicates that P_1 is in its CS and $c_1 = 1$ indicates that it is not in CS. Process P_2 checks the value of c_1 to decide whether it can enter its CS. This check eliminates the progress violation of Algorithm 9.1 because processes are not forced to take turns using their CSs.

Algorithm 9.2 violates the mutual exclusion condition when both processes try to enter their CSs at the same time. Both c_1 and c_2 will be 1 (since none of the processes is in its CS), hence both processes will enter their CSs. To avoid this problem, the statements **while** $c_2 = 0$ **do** { *nothing* }; and $c_1 := 0$; in process P_1 could be interchanged. (Similarly, in process P_2 **while** $c_1 = 0$ **do** { *nothing* }; and $c_2 := 0$; could be interchanged.) This way c_1 will be set to zero before P_1 checks the value of c_2 , hence both processes cannot be in their CSs at the same time. However, if both processes try to enter their CSs at the same time, both c_1 and c_2 will be 0 hence both processes will wait for each other indefinitely. This is a deadlock situation.

Both, the correctness violation and the deadlock possibility, can be eliminated if a process defers to the other process when it finds that the other process also wishes to enter its CS. This can be achieved as follows: if P_1 finds that P_2 is also trying to enter its CS, it can set c_1 to 0. This will permit P_2 to enter its CS. P_1 can wait for some time and try to enter its CS after turning c_1 to 1. Similarly, P_2 can set c_2 to 0 if it finds that P_1 is also trying to enter its CS. However, this approach may lead to a situation in which both processes defer to each other indefinitely. This situation is called a *livelock*.

Dekker's algorithm Dekker's algorithm combines the useful features of Algorithms 9.1–9.2 to avoid a livelock situation. A variable named *turn* is used to avoid livelocks. If both processes try to enter their CSs, *turn* indicates which process should be allowed to enter. Many other features of Dekker's algorithm are analogous to those used in the previous algorithms.

Algorithm 9.3 Dekker's Algorithm

```

var      turn : 1 .. 2;
           $c_1, c_2$  : 0 .. 1;
begin
     $c_1 := 1$ ;
     $c_2 := 1$ ;
    turn := 1;
Parbegin
    repeat                                repeat
         $c_1 := 0$ ;                       $c_2 := 0$ ;
        while  $c_2 = 0$  do                while  $c_1 = 0$  do
            if turn = 2 then            if turn = 1 then
                begin

```

```

 $c_1 := 1;$   $c_2 := 1;$ 
while  $turn = 2$  do { nothing };
 $c_1 := 0;$   $c_2 := 0;$ 
end; end;
{ Critical Section } { Critical Section }
 $turn := 2;$   $turn := 1;$ 
 $c_1 := 1;$   $c_2 := 1;$ 
{ Remainder of { Remainder of
the cycle } the cycle }
forever; forever;
Parend
end.

```

Process P₁ *Process P₂*

Variables c_1 and c_2 are used as status flags of the processes. The statement **while** $c_2 = 0$ **do** in P_1 checks whether it is safe for P_1 to enter its CS. To avoid the correctness problem of Algorithm 9.2, the statement $c_1 := 0$ in P_1 precedes it. If $c_2 = 1$ when P_1 wishes to enter a CS, P_1 skips the **while** loop and enters its CS straightaway. If both processes try to enter their CSs at the same time, value of $turn$ is used to force one of them to defer to the other. For example, if P_1 finds $c_2 = 0$, it defers to P_2 only if $turn = 2$, else it simply waits for c_2 to become 1 before entering its CS. Process P_2 , which is also trying to enter its CS at the same time, is forced to defer to P_1 only if $turn = 1$. In this manner the algorithm satisfies mutual exclusion and also avoids deadlock and livelock conditions.

Peterson's algorithm Peterson's algorithm is simpler than Dekker's algorithm. It uses a boolean array *flag*, which contains one flag for each process. These flags are equivalent to status variables c_1, c_2 of earlier algorithms. A process sets its flag to *true* when it wishes to enter a CS and sets it back to *false* when it exits from the CS. Processes are assumed to have the ids P_0 and P_1 . A process id is used as a subscript to access the status flag of a process in the array *flag*. Variable *turn* is for avoiding livelocks, however it is used differently than in Dekker's algorithm.

Algorithm 9.4 Peterson's Algorithm

```

var    flag : array [0 .. 1] of boolean;
        turn : 0 .. 1;
begin
    flag[0] := false;
    flag[1] := false;
Parbegin
    repeat
        flag[0] := true;
        turn := 1;
    repeat
        flag[1] := true;
        turn := 0;

```

by Lamport.

Algorithm 9.5 An n Process Algorithm

```

const     $n = \dots;$ 
var       $flag : \text{array } [0..n - 1] \text{ of } (\text{idle}, \text{want-in}, \text{in-CS});$ 
            $turn : 0 .. n - 1;$ 
begin
  for  $j := 0$  to  $n - 1$  do
     $flag[j] := \text{idle};$ 
Parbegin
  process  $P_i$ :
    repeat
      repeat
         $flag[i] := \text{want-in};$ 
         $j := turn;$ 
        while  $j \neq i$ 
          do if  $flag[j] \neq \text{idle}$ 
            then  $j := turn \{ \text{Loop here!} \}$ 
            else  $j := j + 1 \bmod n;$ 
         $flag[i] := \text{in-CS};$ 
         $j := 0;$ 
        while ( $j < n$ ) and ( $j = i$  or  $flag[j] \neq \text{in-CS}$ )
          do  $j := j + 1;$ 
      until ( $j \geq n$ ) and ( $turn = i$  or  $flag[turn] = \text{idle}$ );
       $turn := i;$ 
      { Critical Section }
       $j := turn + 1 \bmod n;$ 
      while ( $flag[j] = \text{idle}$ ) do  $j := j + 1 \bmod n;$ 
       $turn := j;$ 
       $flag[i] := \text{idle};$ 
      { Remainder of the cycle }
    forever
  process  $P_k$ : ...
Parend
end.

```

The variable $turn$ is still used to indicate which process may enter its CS next. Each process has a 3-way status flag which takes the values idle , want-in and in-CS . The flag has the value idle when a process is in the remainder of its cycle. A process turns its flag to want-in whenever it wishes to enter a CS. It now makes a few checks to decide whether it may change the flag to in-CS . It checks the flags of other processes in an order that we call the modulo order. The modulo order is $P_{turn}, P_{turn+1}, \dots, P_{n-1}, P_0, P_1, \dots, P_{turn-1}$. In the first **while** loop, the process checks whether any

process ahead of it in the modulo order wishes to use its CS. If not, it turns its flag to *in-CS*.

Since many processes may make this check concurrently, more than one process may simultaneously reach the same conclusion. Hence another check is made to ensure correctness. The second **while** loop checks whether any other process has turned its flag to *in-CS*. If so, the process changes its flag back to *want-in* and repeats all the checks. All other processes, which had changed their flags to *in-CS* also change their flags back to *want-in* and repeat the checks. These processes will not tie for entry to a CS again because they have all turned their flags to *want-in*, hence only one of them can get past the first **while** loop. This feature avoids the livelock condition. The process earlier in the modulo order from P_{turn} will get in and enter its CS ahead of other processes.

This solution contains a certain form of unfairness since processes do not enter their CSs in the order in which they request entry to a CS. This unfairness is eliminated in the Bakery algorithm by Lamport [1974].

Bakery algorithm When a process wishes to enter a CS, it chooses a token such that the number contained in the token is larger than any number issued earlier. *choosing* is an array of boolean flags. *choosing[i]* is used to indicate whether process P_i is currently engaged in choosing a token. *number[i]* contains the number in the token chosen by process P_i . $number[i] = 0$ if P_i has not chosen a token since the last time it entered the CS. Processes enter CS in the order of numbers in their tokens.

Algorithm 9.6 Bakery Algorithm

```

const    n = . . . ;
var      choosing : array [0 .. n - 1] of boolean;
           number : array [0 .. n - 1] of integer;
begin
  for j := 0 to n - 1 do
    choosing[j] := false;
    number[j] := 0;
Parbegin
  process  $P_i$  :
    repeat
      choosing[i] := true;
      number[i] := max (number[0], . . . , number[n - 1]) + 1;
      choosing[i] := false;
      for j := 0 to n - 1 do
        begin
          while choosing[j] do { nothing };
          while number[j] ≠ 0 and (number[j], j) < (number[i], i)
            do { nothing };
        end;

```

9.8.1.2 Bounded Concurrency

We use the term *bounded concurrency* for the situation in which up to c processes can concurrently perform an operation op_i , where c is a constant ≥ 1 . Bounded concurrency is implemented by initializing a semaphore sem_c to c . Every process wishing to perform op_i performs a $wait(sem_c)$ before performing op_i , and a $signal(sem_c)$ after performing it. From the semantics of *wait* and *signal* operations, it is clear that up to c processes can concurrently perform op_i . Semaphores used to implement bounded concurrency are called *counting semaphores*.

Figure 9.24 illustrates how a set of concurrent processes share five printers.

```

var printers : semaphore := 5;
Parbegin
    repeat
        wait(printers);
        { Use a printer }
        signal(printers);
        { Remainder of the cycle }
    forever;
Parend
end.
Process p1 ... Process pn

```

Fig. 9.24 Bounded concurrency using semaphores

9.8.1.3 Signaling Between Processes

Consider the synchronization requirements of processes P_i and P_j shown in Figure 9.4. A semaphores can be used to achieve this synchronization as shown in Figure 9.25. Here process P_i performs a $wait(sync)$ before executing action a_i and P_j performs a $signal(sync)$ after executing action a_j . Semaphore $sync$ is initialized to 0 hence P_i will get blocked on $wait(sync)$ if P_j has not already performed a $signal(sync)$. It would proceed to perform action a_i only after process P_j performs a $signal$. Unlike the solution of Figure 9.4, race conditions cannot arise because the *wait* and *signal* operations are indivisible. The signaling arrangement can be used repetitively; it does not require resetting as in the signaling arrangement of Figure 9.12.

9.8.2 Producers–Consumers Using Semaphores

As discussed in Section 9.5.1, the producers–consumers problem is a signaling problem. After consuming a record from a buffer, a consumer signals to a producer that is waiting to produce in the same buffer. Analogously, a producer signals to a waiting consumer. Hence we should implement producers–consumers using the signaling arrangement shown in Figure 9.25.

For simplicity, we first discuss the solution for the single buffer case shown in Figure 9.26. The buffer pool is represented by an array of buffers with a single

```

var sync : semaphore := 0;
Parbegin
    ...
    wait(sync); {Perform action  $a_i$ }
    ...
Parend
end.

```

Process P_i

```

signal(sync); {Perform action  $a_j$ }

```

Process P_j

Fig. 9.25 Signaling using semaphores

element in it. Two semaphores *full* and *empty* are declared. They are used to indicate the number of full and empty buffers, respectively. *full* is initialized to 0 and *empty* is initialized to 1. A producer performs a *wait(empty)* before starting the produce action and a consumer performs a *wait(full)* before a consume action.

```

type item = ...;
var
    full : Semaphore := 0; { Initializations }
    empty : Semaphore := 1;
    buffer : array [0] of item;
begin
Parbegin
    repeat
        wait(empty);
        buffer[0] := ...; { i.e. produce }
        signal(full);
        { Remainder of the cycle }
    forever;
Parend
end.

```

Producer

```

repeat
    wait(full);
    x := buffer[0]; { i.e. consume }
    signal(empty);
    { Remainder of the cycle }
forever;

```

Consumer

Fig. 9.26 Producers-consumers with a single buffer

Initially consumer(s) would get blocked on a consume and only one producer would get past the *wait(empty)* operation. After completing the produce action it performs *signal(full)*. This enables one consumer to enter, either immediately or in future. When the consumer finishes a consume, it performs a *signal(empty)* operation that enables a producer to enter and perform a produce action. This solution avoids busy waits since semaphores are used to check for empty or full buffers, hence a process gets blocked if it cannot find a full or empty buffer. The total concurrency is 1, sometimes a producer executes and sometimes a consumer executes. Example 9.3

```

const           n = ...;
type            item = ...;
var
    buffer : array [0..n - 1] of item;
    full : Semaphore := 0; { Initializations }
    empty : Semaphore := n;
    prod_ptr, cons_ptr : integer;
begin
    prod_ptr := 0;
    cons_ptr := 0;
Parbegin
    repeat
        wait(empty);
        buffer[prod_ptr] := ...;
        { i.e. produce }
        prod_ptr := prod_ptr+1 mod n;
        signal(full);
    forever;
Parend
end.

```

Producer

```

repeat
    wait(full);
    x := buffer[cons_ptr];
    { i.e. consume }
    cons_ptr := cons_ptr+1 mod n;
    signal(empty);
forever;

```

Consumer

Fig. 9.28 Bounded buffers using semaphores

9.8.3 Readers-Writers Using Semaphores

Key features of the readers-writers problem are evident from the outline of Figure 9.15. These are as follows:

- Any number of readers can read concurrently.
- Readers and writers must wait if a writer is writing. When the writer exits, either all waiting readers should be activated or one waiting writer should be activated.
- A writer must wait if readers are reading. It must be activated when the last reader exits.

We implement these features by keeping track of the number of readers and writers that wish to read or write at any moment. The following counters are introduced for this purpose:

<i>runread</i>	count of readers currently reading
<i>totread</i>	count of readers waiting to read or currently reading
<i>runwrite</i>	count of writers currently writing
<i>totwrite</i>	count of writers waiting to write or currently writing

Values of these counters are incremented and decremented at appropriate times by the processes. For example, a reader process increments *totread* when it decides

to read and it increments *runread* when it actually starts reading. It decrements both *totread* and *runread* when it finishes reading. Thus *totread* – *runread* indicates how many readers are waiting to begin reading. Similarly *totwrite* – *runwrite* indicates the number of writers waiting to begin writing. Values of these counters are used as follows: A reader is allowed to begin reading when *runwrite* = 0 and a writer is allowed to begin writing when *runread* = 0 and *runwrite* = 0. The value of *totread* is used to activate all waiting readers when a writer finishes writing (note that *runread* = 0 when a writer is writing).

```

Parbegin
repeat
  if runwrite ≠ 0
  then
    { wait };
    { read }
    if runread = 0 and
      totwrite ≠ 0
    then
      activate one writer
  forever;
Parend
Reader(s)

repeat
  if runread ≠ 0 or
  runwrite ≠ 0
  then { wait };
    { write }
  if totread ≠ 0 or totwrite ≠ 0
  then
    activate totread readers
    or activate one writer
  forever;
Writer(s)

```

Fig. 9.29 Refined solution outline for readers-writers without writer priority

The outline of Figure 9.15 is refined to obtain the solution shown in Figure 9.29. Values of the counters are examined inside critical sections. It is assumed that their values are incremented or decremented in other critical sections at appropriate places. This solution does not use explicit critical sections for readers or writers. Instead they are blocked until they can be allowed to start reading or writing.

Blocking of readers and writers resembles the blocking of producers and consumers in the producers-consumers problem, so it is best handled using semaphores for signaling. We introduce two semaphores named *reading* and *writing*. A reader process performs *wait(reading)* before starting to read. This action blocks the reader process if conditions permitting it to read are not currently satisfied, else the reader can get past the *wait* operation and start reading. Similarly, a writer process performs a *wait(writing)* before writing.

Who should perform *signal* operations to activate the reader and writer processes blocked on their *wait* operations? Obviously conditions governing the start of reading or writing were not satisfied when the reader or writer processes were blocked. These conditions change when any of the counter values change, i.e., when a reader finishes reading or a writer finishes writing. Hence processes must perform appropriate *signal* operations after completing a read or a write operation.

```

var
    totread, runread, totwrite, runwrite : integer;
        reading, writing : semaphore := 0;
            mutex : semaphore := 1;
begin
    totread := 0;
    runread := 0;
    totwrite := 0;
    runwrite := 0;
Parbegin
    repeat
        wait(mutex);
        totread := totread+1;
        if runwrite = 0 then
            runread := runread+1;
            signal(reading);
            signal(mutex);
            wait(reading);
            { Read }
            wait(mutex);
            runread := runread-1;
            totread := totread-1;
            if runread = 0 and
                totwrite > runwrite
            then
                runwrite := 1;
                signal(writing);
                signal(mutex);
        forever;
Parend
end.

```

Reader(s)

Writer(s)

```

repeat
    wait(mutex);
    totwrite := totwrite+1;
    if runread = 0 then
        runwrite := 1;
        signal(writing);
        signal(mutex);
    while (runread<totread) do
        begin
            runread := runread+1;
            signal(reading);
        end;
        if runread = 0 and
            totwrite > runwrite then
                runwrite := 1;
                signal(writing);
                signal(mutex);
    forever;

```

Fig. 9.30 Readers and writers using semaphores

The *wait* operation has a very low failure rate in most systems using semaphores, i.e., processes performing *wait* operations are seldom blocked. This characteristic is exploited in some methods of implementing semaphores to reduce the overhead. In the following, we describe three methods of implementing semaphores and examine their overhead implications. Recall that we use the term process as a generic term for both processes and threads.

Kernel-level implementation The kernel implements the *wait* and *signal* procedures of Figure 9.31. All processes in a system can share a kernel-level semaphore. However, every *wait* and *signal* operation results in a system call; it leads to high overhead of using semaphores. In a uniprocessor OS with a non-interruptible kernel, it would not be necessary to use a lock variable to eliminate race conditions, so the overhead of busy waits in the *Close_Lock* operation can be eliminated.

User-level implementation The *wait* and *signal* operations are coded as library procedures, which are linked with an application program so that processes of the application can share user-level semaphores. The *block_me* and *activate* calls are actually calls on library procedures, which handle blocking and activation of processes themselves as far as possible and make system calls only when they need assistance from the kernel. This implementation method would suit user-level threads because the thread library would already provide for blocking, activation and scheduling of threads. The thread library would make a *block_me* system call only when all threads of a process are blocked.

Hybrid implementation The *wait* and *signal* operations are again coded as library procedures, and processes of an application can share the hybrid semaphores. *block_me* and *activate* are system calls provided by the kernel and the *wait* and *signal* operations make these calls only when processes have to be blocked and activated. In principle, it is possible to implement hybrid semaphores even if such kernel support is not available. For example, the system calls *block me* and *activate* can be replaced by system calls for receiving and sending messages or signals. However, message passing is subject to availability of buffer space, and signals can be ignored by processes, so it is best to use a separate *block_me* system call for semaphore implementation.

9.9 CONDITIONAL CRITICAL REGIONS

The conditional critical region (CCR) is a control structure in a higher level programming language. It provides two features for process synchronization—it provides mutual exclusion over accesses to shared data, and it permits a process executing CCR to block itself until a specified boolean condition becomes *true*.

Figure 9.32 shows a concurrent program using the CCR construct **region** *x* **do** .. Variable *x* is declared with the attribute **shared**. It can be used in any process of the program. However, it must be used only within a CCR, i.e., within a **region** *x* **do**

```

const n = ...;
type item = ...;
var
    buffer_pool : Shared record
        buffer : array [0..n - 1] of item;
        full : integer := 0;
        prod_ptr : integer := 0;
        cons_ptr : integer := 0;
    end
begin
Parbegin
    var produced_info : item;
    repeat
        { Produce in produced_info }
        region buffer_pool do
            begin
                await (full < n);
                buffer[prod_ptr]
                    := produced_info;
                prod_ptr :=
                    prod_ptr + 1 mod n;
                full := full + 1;
            end;
        { Remainder of the cycle }
        forever;
Parend
end.
Producer
var for_consumption : item;
repeat
    region buffer_pool do
        begin
            await (full > 0);
            for_consumption :=
                buffer[cons_ptr];
            cons_ptr :=
                cons_ptr + 1 mod n;
            full := full - 1;
        end;
    { Consume from for_consumption }
    { Remainder of the cycle }
forever;
Consumer

```

Fig. 9.33 Bounded buffer using conditional critical region

any changes to the code of the producer and consumer processes.

Figure 9.34 contains a solution to the readers-writers problem using conditional critical regions. *read_write* is a shared variable containing the counters *runread* and *runwrite*. A reader should read only when no writer is writing. Hence a reader process performs an **await** (*runwrite* = 0) before starting to read. A writer performs a write inside **region read_write do**, hence it is adequate to use an **await** (*runread* = 0) before starting to write. When a writer completes, a reader blocked on *runwrite* = 0 in an **await** statement gets activated and exits the region. The condition *runwrite* = 0 still holds, so other readers blocked on this condition also get activated. Thus there is no need for the counter *totread* of Figure 9.30. The counter *totwrite* is not needed due to a similar reason.

Note that the count *runwrite* of this solution is redundant. Its purpose is to block a reader from entering its CS if a writer is writing. However, since writing is performed inside a CS, no reader can enter its CS while a writer is writing! Hence *runwrite*

```

type item = ...;
var
    read_write : shared record
        runread : integer := 0;
        runwrite : integer := 0;
    end
begin
Parbegin
    repeat
        region read_write do
            begin
                await (runwrite = 0);
                runread := runread + 1;
            end;
            { Read }
            region read_write do;
                runread := runread - 1;
            end;
            { Remainder of the cycle }
        forever;
Parend
end.

```

Reader(s)

```

repeat
    region read_write do
        begin
            await (runread = 0);
            runwrite := runwrite + 1;
            { Write }
            runwrite := runwrite - 1;
        end;
        { Remainder of the cycle }
forever;

```

Writer(s)

Fig. 9.34 Readers and Writers without writer priority

is bound to be 0 every time a reader executes the statement **await** (*runwrite* = 0). A more compact and elegant solution is obtained by eliminating *runwrite* and all statements incrementing or testing its value.

9.9.1 Implementation of CCR

Implementation of the **await** statement is straightforward. The code generated by the compiler can evaluate the condition and, if *false*, make a *block me* system call to block the process. How and when should the blocked process be activated? This is best achieved by linking the condition in the **await** statement with an event known to the scheduling component of the kernel. In Figure 9.33, conditions in the **await** statements (viz. *full* < *n* and *full* > 0) involve fields of *buffer_pool*. Each CCR is implemented as a CS on *buffer_pool*, so these conditions can change only when some process executes the CCR. Hence every time a process exits a CCR, conditions in all **await** statements in the CCR on which processes are blocked can be checked. A blocked process whose **await** condition is satisfied can now be activated.

However, activation of a process is not always so simple. Semantics of the CCR construct permit any boolean condition to be used in an **await** statement. If the condition does not involve a field of the CCR control variable, execution of some

statement outside a CCR may also change the truth value of an `await` condition. For example, conditions involving values returned by system calls like time-of-day can become true even if no process executes a CCR. Such conditions will have to be checked periodically to activate blocked processes. These checks increase the overhead of implementing CCRs.

9.10 MONITORS

A *monitor* is a programming language construct that supports both data access synchronization and control synchronization. A monitor type resembles a class in a language like C++ or Java. It has the four aspects summarized in Table 9.5—declaration and initialization of shared data, operations on shared data and synchronization operations. A concurrent program creates monitor objects, i.e., objects of a monitor type, and uses them to perform operations on shared data and implement process synchronization using synchronization operations. We refer to an object of a monitor type as a monitor variable, or simply as a monitor.

Table 9.5 Aspects of a monitor type

Aspect	Description
Data declaration	Shared data and condition variables are declared here. Copies of this data exist in every object of a monitor type.
Data initialization	Data are initialized when a monitor, i.e. an object of a monitor type, is created.
Operations on shared data	Operations on shared data are coded as procedures of the monitor type. The monitor ensures that these operations are executed in a mutually exclusive manner.
Synchronization operation	Procedures of the monitor type use synchronization operations <code>wait</code> and <code>signal</code> over <i>condition variables</i> to synchronize execution of processes.

Operations on shared data are coded as procedures of the monitor. To ensure consistency of shared data and absence of race conditions in their manipulation, the monitor ensures that its procedures are executed in a mutually exclusive manner. Calls on procedures of a monitor are serviced in a FIFO manner to satisfy the bounded wait property. This feature is implemented by maintaining a queue of processes that wish to execute monitor procedures. Process synchronization is implemented using the `signal` and `wait` operations on synchronization data called *condition variables*.

Condition variables The process synchronization support in monitors resembles the event mechanism described in Section 3.3.6 in many respects. Events are called *conditions* in monitors. A condition variable, which is simply a variable with the attribute **condition**, is associated with each condition in a monitor. Thus,

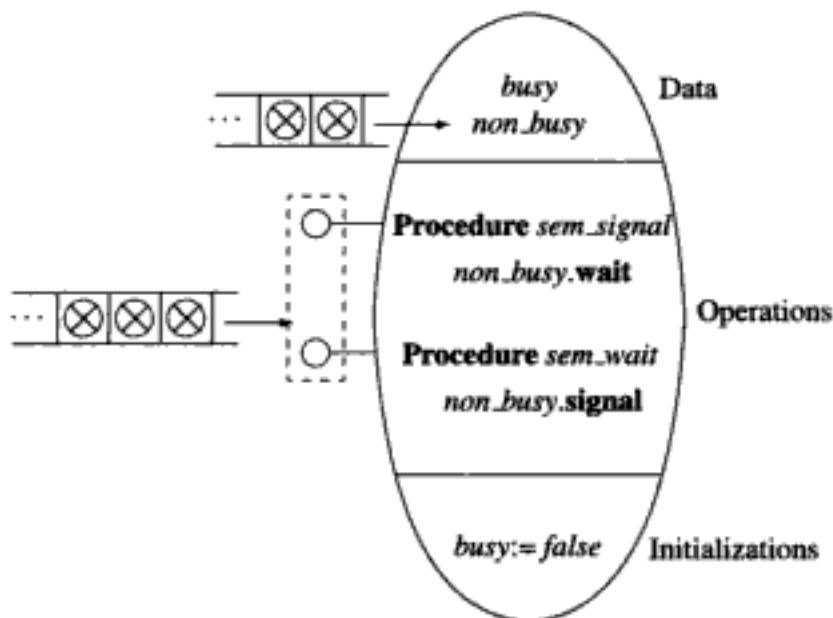


Fig. 9.36 A monitor implementing binary semaphore

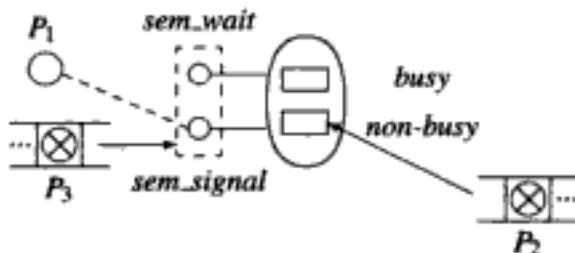


Fig. 9.37 Snapshot of the system of Example 9.4

Scheduling process P_2 would delay the signaling process P_1 , which seems unfair. Scheduling P_1 would imply that P_2 is not really activated until P_1 leaves the monitor. Hoare (1974) proposed the first alternative. Brinch Hansen (1973) proposed that a **signal** statement should be the last statement of a monitor procedure; this way, the process executing **signal** exits the monitor procedure immediately and the process activated by the **signal** statement can be scheduled.

As mentioned earlier, use of condition variables has two advantages: First, a **signal** has no effect if no process is blocked on the condition variable. This feature simplifies the logic of operations on shared data. Example 9.5 illustrates this fact in the context of solution to the producers-consumers problem. Second, unlike in conditional critical regions, processes are activated by execution of **signal** statements. Hence the implementation of monitors need not repeatedly check whether a blocked process can be activated; this reduces the process synchronization overhead.

| Figure 9.38 shows a solution to the producers-consumers problem using monitors.

```

type Disk_Mon_type : monitor
  const n = ... ; { Size of IO list }
    m = ... ; { Number of user processes }
  type Q_element = record
    proc_id : integer;
    track_no : integer;
  end;
  var
    IO_list : array [1..n] of Q_element;
    proceed : array [1..m] of condition;
    whose_IO_in_progress, count : integer;
  procedure IO_request (proc_id, track_no : integer);
  begin
    { Enter proc_id and track_no in IO_list }
    count := count + 1;
    if count > 1 then proceed[proc_id].wait;
    whose_IO_in_progress := proc_id;
  end;
  procedure IO_complete (proc_id : integer);
  var id : integer;
  begin
    if whose_IO_in_progress ≠ proc_id then { indicate error }
      { Remove request of this process from IO_list }
    count := count - 1;
    if count > 0 then
      begin
        { Select the IO operation to be initiated from IO_list }
        id := id of requesting process;
        proceed[id].signal;
      end;
    end;
    begin { Initializations }
      count := 0;
      whose_IO_in_progress := 0;
    end;
  end;

```

Fig. 9.39 A monitor for implementing the disk scheduler

The disk scheduling arrangement is as follows: In the code of a user process, each I/O operation is preceded by a call to the disk monitor procedure *IO_request*. Process id and details of the I/O operation are parameters of this call. *IO_request* enters these parameters in *IO_list*. If an I/O operation is in progress, it blocks the user process until its I/O operation can be initiated. This blocking is performed by issuing a *wait* on a condition variable.

When the disk scheduler decides to schedule the I/O operation, it executes a *signal* statement to activate the user process. The user process now exits the monitor procedure and starts its I/O operation. After the I/O operation, it invokes the monitor

(2,85), (3,40) and (4,100). Processes P_1 – P_4 are blocked on different elements of *proceed*, which is an array of condition variables. *IO_complete* is called by P_6 . It selects the I/O operation on track number 85, and obtains the id of the process that made this request from *IO_list*. This id is 2, hence it executes the statement *proceed[2].signal*. Process P_2 is now activated and proceeds to perform its I/O operation.

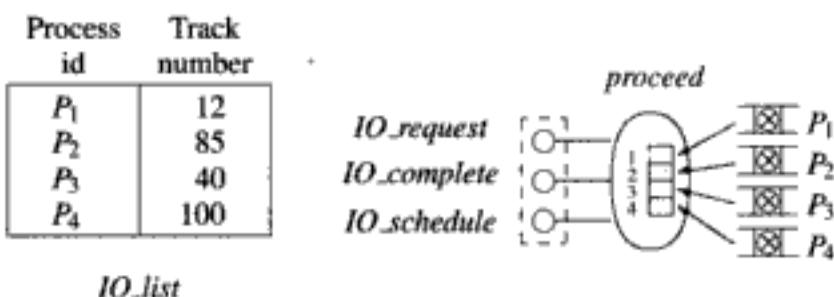


Fig. 9.42 Snapshot of the disk monitor

This disk monitor makes the simplifying assumption that a process can be allowed to perform its own I/O operations. If this is not the case, the I/O operation should be performed by the disk monitor itself (see Problem 23 of Exercise 9).

9.10.2 Monitors in Java

A Java class becomes a monitor type when the attribute `synchronized` is associated with one or more methods in the class. An object of such a class is a monitor. The Java virtual machine ensures mutual exclusion over the `synchronized` methods in a monitor. Each monitor contains a single unnamed condition variable. A thread waits on the condition variable by executing the call `wait()`. The `notify()` call is like the `signal` operation described in Section 9.10. It wakes one of the threads waiting on the condition variable. The Java virtual machine does not implement FIFO behavior for the `wait` and `notify` calls. Thus it does not provide the bounded wait property. The `notifyAll()` call activates all threads waiting on the condition variable.

Provision of a single condition variable in a monitor has the effect of clubbing together all threads that require synchronization in the monitor. It leads to busy waits in an application if threads need to wait on different conditions. This problem is encountered in a readers-writers system. When a writer is active, all readers wishing to read and all writers wishing to write will have to wait on the same condition variable. If the writer executes a `notify()` call when it finishes writing, either one reader or one writer will be activated. However, concurrent reading is desirable, so it is not enough to activate only one reader. The writer would have to use a `notifyAll()` call to activate all waiting threads. If readers are preferred, all writer threads will have to perform `wait()` calls once again. If writers are preferred, all reader threads and some writer threads will have to perform `wait()` calls once again. Thus, a reader or writer thread may be activated many times before it gets an opportunity to perform reading or writing. A producers-consumers system would also suffer from a similar

balanced number of *wait* and *signal* operations on sem_i . Thus the undo operation does not interfere with normal operation of processes using semaphores.

Unix 4.4BSD permits semaphores to be placed in memory areas shared by a set of processes. Processes using such semaphores can access values of the semaphores in a mutually exclusive manner by using indivisible instructions. To facilitate blocking and activation of processes, the use of semaphores proceeds as follows: A process wishing to perform a *wait* operation examines the value of the semaphore. If the value is > 0 , it decrements the value and continues its execution. If the value is 0, it adds its own id to a wait-list for the semaphore that is maintained in the shared memory and makes a *block me* system call. If the wait-list is non-empty, a process performing a *signal* operation performs an *activate this process* system call on some process in the list. This approach provides fast synchronization by avoiding system calls in cases where a *wait* operation does not lead to blocking (see Section 9.8.4 for a related discussion), or a *signal* operation does not lead to activation of a blocked process.

9.12 PROCESS SYNCHRONIZATION IN LINUX

Linux provides a Unix-like semaphore (see Section 9.11) for use by user processes. It also provides two kinds of semaphores for use by a kernel—a conventional semaphore and a reader-writer semaphore. Implementation of the conventional semaphore for the kernel is analogous to the kernel-level implementation scheme discussed in Section 9.8.4. The kernel uses a structure that contains the value of the semaphore, a list of waiting processes and a flag to indicate whether any processes are waiting for the semaphore. Thus, unlike the scheme of Section 9.8.4, a lock is not used to avoid race conditions on the value of the semaphore; instead, the *wait* and *signal* operations use indivisible instructions to decrement and increment the value of the semaphore. This reduces the overhead of *wait* and *signal* operations. The list of waiting processes is separately locked when processes are added to it or removed from it by these operations. The flag of waiting processes is used to reduce this locking overhead as follows: The *signal* operation checks the flag to find whether any processes are waiting on the semaphore, and locks the list of waiting processes only if the flag is set.

The reader-writer semaphore provides capabilities that can be used to implement the readers-writers problem of Section 9.8.3 within a kernel so that many processes can read a kernel data structure concurrently but only one process can update it. The kernel implements the reader-writer semaphore by maintaining a count of the number of readers that are concurrently performing a read operation, and a list of processes waiting to perform a read or write operation, which is organized in the chronological order. The implementation of reader-writer semaphores does not favor either readers or writers—it permits processes to enter their critical sections in FIFO order, except that consecutive readers can read concurrently. Thus, a reader process would have to wait if a writer process is already waiting, and would be enabled to

read sometime after the writer completes.

Kernels older than the Linux 2.6 kernel implemented mutual exclusion in the kernel space through system calls. However, as mentioned in Section 9.8.4, a *wait* operation has a low failure rate, i.e., a process is rarely blocked on a *wait* call, so many of the system calls are actually unnecessary. The Linux 2.6 kernel provides a fast user space mutex called *futex*. A *futex* is an integer in shared memory on which only certain operations can be performed. The *wait* operation on a *futex* makes a system call only when a process needs to be blocked on the *futex*, and the *signal* operation on a *futex* makes a system call only when a process is to be activated. The *wait* operation also provides a parameter through which a process can indicate how long it should be blocked on the *wait*. When this time elapses, the *wait* operation fails and returns an error code to the process that made the call.

9.13 PROCESS SYNCHRONIZATION IN WINDOWS

Windows is an object-oriented system. All entities in the system are represented by objects. The kernel provides objects called *dispatcher objects* for thread synchronization. Each of the objects representing processes, files and events embed a dispatcher object. This way, these objects can be used for thread synchronization. Mutexes and semaphores are also represented by objects.

A dispatcher object is either in the *signaled* state or in the *nonsignaled* state. If it is in the *nonsignaled* state, any thread that wishes to synchronize with it is put in the *waiting* state. The semantics of an object determine which of the threads waiting on it would be activated when it is signaled and what would be its new state when it is signaled or when a thread performs a wait operation on it (see Table 9.6). A thread object enters the *signaled* state when it terminates, whereas a process object enters the *signaled* state when all threads in it terminate. In both cases, all threads waiting on the object are activated. A job object is signaled when the job time specified for it expires. Windows also maintains the signal status of a thread or process object as a boolean variable, and permits a thread to access it. This way, a thread can check the status of a thread or process without becoming blocked.

The file object is signaled when an I/O operation on the file completes. All threads waiting on it are activated and its synchronization state changes back to *nonsignaled*. If no threads are waiting on it, a thread that waits on it sometime in future will get past the wait operation and the synchronization state of the file object would be changed to *nonsignaled*. The console input object has an analogous behavior except that only one waiting thread is activated when it is signaled. The file change object is signaled when the system detects changes. It behaves like the file object in other respects.

Threads use the event, mutex and semaphore objects for mutual synchronization. They signal these objects by executing library functions that lead to appropriate sys-

3. Prove that the progress condition is satisfied in the case of Dekker's algorithm. (*Hint:* Consider $turn = 2$ and process P_1 trying to enter its CS.)
4. For Dekker's algorithm, prove that
 - (a) Deadlock condition cannot arise
 - (b) Livelock condition cannot arise.
5. Is the bounded wait condition satisfied by Peterson's algorithm?
6. The following changes are made in Peterson's algorithm (see Algorithm 9.4): The statements $flag[0] := true$ and $flag[0] := false$ in process P_0 are interchanged (and analogous changes are made in process P_1). Discuss which properties of the implementation of critical sections are violated by the resulting system.
7. If the statement **while** $flag[1]$ **and** $turn = 1$ in Peterson's algorithm is changed to **while** $flag[1]$ **or** $turn = 1$ (and analogous changes are made in process P_1), which properties of critical section implementation are violated by the resulting system?
8. Explain the behavior of a process P_i in Algorithm 9.5 when process P_i finishes its CS and no other process is interested in entering its CS.
9. In Lamport's Bakery algorithm a process that chooses a token earlier in time receives a smaller token than a process that chooses a token later. Explain the need for the first **while** loop in this context.
10. Extend the producers-consumers solution of Figure 9.11 to permit many producer and consumer processes.
11. The solution of the producers-consumers problem shown in Figure 9.43 uses kernel calls *block* and *activate* for process synchronization. It contains a race condition. Describe how the race condition arises.
12. Identify features of the producers-consumers solution of Figure 9.28 that need to be modified if more than one producer or more than one consumer exists in the system. (*Hint:* Do not make any assumptions concerning relative speeds of processes.)
13. The readers-writers solution of Figure 9.30 uses two semaphores even though a single entity—the shared data—is to be controlled. Modify this solution to use a single semaphore *rw_permission* instead of semaphores *reading* and *writing*. (*Hint:* perform a *wait(rw_permission)* in the reader only if reading is not already in progress.)
14. Modify the readers-writers solution of Figure 9.30 to implement a writer preferred readers-writers system.
15. Show that a deadlock can arise if the reader process of Figure 9.30 is modified to perform a *wait(reading)* inside its CS on *mutex* (rather than performing it after its CS) when $runwrite \neq 0$.
16. Show that a critical section implemented using semaphores would satisfy the bounded wait property if the *signal* operation activates processes in FIFO order.
17. A resource is to be allocated to requesting processes in a FIFO manner. Each process is coded as

```

repeat
  ...
  request-resource(process_id, resource_id);
  { Use resource }
  release-resource(process_id, resource_id);
  { Remainder of the cycle }

```

be performed by the disk scheduler itself rather than by user processes. Give the modified code for *Disk.Mon.type* and compare its behavior with that of the original monitor.

24. An airline reservation system is to be implemented using monitors. The system must process booking and cancellation requests and perform appropriate actions.
 - (a) Identify the shared data, operations and processes required.
 - (b) Implement the reservation system using monitors.
25. A customer gives the following instructions to a bank manager: Do not credit any funds to my account if the balance in my account exceeds n , and hold any debits until the balance in the account is large enough to permit the debit.

Implement a bank account using a monitor to implement these instructions.
26. The synchronization problem called *sleeping barber* is defined as follows: A barber shop has a single barber, a single barber's chair in a small room and a large waiting room with n seats. The barber and his chair are visible from the waiting room. After servicing one customer, the barber checks if any customers are waiting in the waiting room. If so, he admits one of them and starts serving him, else he goes to sleep in the barber's chair. A customer enters the waiting room only if there is at least one vacant seat and either waits for the barber to call him if the barber is busy, or wakes the barber if he is asleep.

Identify the synchronization requirements between the barber and customer processes. Code the barber and customer processes such that deadlocks do not arise.
27. An interprocess message communication system is to be implemented using monitors. The system handles messages exchanged by 4 user processes, using a global pool of 20 message buffers. The system is to operate as follows:
 - (a) The system uses the asymmetric naming convention (see Section 10.1.1).
 - (b) The system allocates a message buffer and copies the sender's message in it. If no free buffer exists, the sender is blocked until a message buffer becomes available.
 - (c) When a receiver performs a *receive*, it is given a message sent to it in FIFO order. A message is copied into the area provided by the receiver, and the message buffer is freed. A receiver is blocked if no message exists for it.
 - (d) The system detects a deadlock if one arises.

Write a monitor called *Communication_Manager* to perform the above functions. What are the condition variables used in it? Clearly explain how blocked senders and receivers are activated.
28. A monitor is to be written to simulate a clock manager used for real-time control of concurrent processes. The clock manager maintains a variable named *clock* to maintain the current time. The OS supports a signal called *elapsed_time* that is generated every 2 msec. The clock manager indicates a signal handling action for *elapsed_time* (see Section 3.5.1) and updates *clock* at every occurrence of the signal. A typical request made to the manager is 'wake me up at 9.00 a.m.' The manager blocks the processes making such requests and arranges to activate them at the designated times. Implement this monitor.
29. Nesting of monitor calls implies that a procedure in monitor A calls a procedure of

- another monitor, say monitor B. During execution of the nested call, the procedure of monitor A continues to hold its mutual exclusion. Prove that nested monitor calls can lead to deadlocks.
30. Write a short note on the implementation of monitors. Your note must discuss
 - (a) How to achieve mutual exclusion between the monitor procedures.
 - (b) Whether monitor procedures need to be coded in a re-entrant manner.
 31. A large data collection D is used merely to answer statistical queries, so updates are never performed on D . To improve response to queries a part of D , say D_1 , is loaded in memory to answer user queries. Queries concerning D_1 are processed concurrently. If no queries are active on D_1 , and queries exist on some other part of data, say D_2 , D_2 is loaded in memory and queries on it are processed concurrently. This system may be called a readers-readers system if D is split into two parts D_1 and D_2 . Implement this system using any primitive or control structure for concurrency.
 32. An old bridge on a busy highway is too narrow to permit two-way traffic, so one-way traffic is to be implemented on the bridge by alternately permitting vehicles traveling in opposite directions to use the bridge. The following rules are formulated for use of the bridge:
 - (a) At any time the bridge is used by vehicle(s) traveling in one direction only.
 - (b) If vehicles are waiting to cross the bridge at both ends, only one vehicle from one end is allowed to cross the bridge before a vehicle from the other end starts crossing the bridge.
 - (c) If no vehicles are waiting at one end, then any number of vehicles from the other end are permitted to cross the bridge.

Develop a concurrent system to implement these rules.
 33. When vehicles are waiting at both ends, the rules of Problem 32(a) lead to poor use of the bridge. Hence, upto 10 vehicles should be allowed to cross the bridge in one direction even if vehicles are waiting at the other end. Implement the modified rules.

BIBLIOGRAPHY

Dijkstra (1965) discussed the mutual exclusion problem, described Dekker's algorithm and presented a mutual exclusion algorithm for n processes. Lamport (1974, 1979) describes and proves the Bakery algorithm. Ben Ari (1982) describes the evolution of mutual exclusion algorithms and provides a proof of Dekker's algorithm. Ben Ari (2006) discusses concurrent and distributed programming. Peterson (1981), Lamport (1986, 1991) and Raynal (1986) are other sources on mutual exclusion algorithms.

Dijkstra (1965) proposed semaphores. Hoare (1972) and Brinch Hansen (1972) discuss the critical and conditional critical regions, which are synchronization constructs that preceded monitors. Brinch Hansen (1973) and Hoare (1974) describe the monitor concept. Buhr *et al* (1995) describe different monitor implementations. Richter (1999) describes thread synchronization in C/C++ programs under Windows. Christopher and Thiruvathukal (2001) describe the concept of monitors in Java, compare it with the monitors of Brinch Hansen and Hoare, and conclude that Java synchronization is not as well developed as the Brinch Hansen and Hoare monitors.

A synchronization primitive or construct is complete if it can be used to implement *all* process synchronization problems. The completeness of semaphores has been discussed in

Patil (1971), Kosaraju (1975) and Lipton (1974).

Brinch Hansen (1973, 1977) and Ben Ari (1982, 2006) discuss the methodology for building concurrent programs. Owicki and Gries (1976) and Francez and Pneuli (1978) deal with the methodology of proving the correctness of concurrent programs.

Vahalia (1996) and Stevens and Rago (2005) discuss process synchronization in Unix, Beck *et al* (2002), Bovet and Cesati (2003) and Love (2005) discuss synchronization in Linux, Mauro and McDougall (2001) discuss synchronization in Solaris, while Richter (1999) and Russinovich and Solomon (2005) discuss synchronization features in Windows.

1. Beck, M., H. Bohme, M. Dziadzka, U. Kunitz, R. Magnus, C. Schroter, and D. Verworner (2002): *Linux Kernel Programming*, Pearson Education,
2. Ben Ari, M. (1982): *Principles of Concurrent Programming*, Prentice-Hall International, Englewood Cliffs, New Jersey.
3. Ben Ari, M. (2006): *Principles of Concurrent and Distributed Programming, Second edition*, Prentice-Hall, Englewood Cliffs, New Jersey (to appear).
4. Bovet, D. P., and M. Cesati (2003): *Understanding the Linux Kernel*, O'Reilly, Sebastopol.
5. Brinch Hansen, P. (1972): Structured multiprogramming. *Communications of the ACM*, **15** (7), 574–578.
6. Brinch Hansen, P. (1973): *Operating System Principles*, Prentice-Hall, Englewood Cliffs, New Jersey.
7. Brinch Hansen, P. (1975): “The Programming language concurrent Pascal,” *IEEE Transactions on Software Engineering*, **1** (2), 199–207.
8. Brinch Hansen, P. (1977): *The Architecture of Concurrent Programs*, Prentice-Hall, Englewood Cliffs, New Jersey.
9. Buhr, M., M. Fortier, and M. H. Coffin (1995): “Monitor classification,” *Computing Surveys*, **27** (1), 63–108.
10. Chandy, K. M., and J. Misra (1988): *Parallel Program Design : A Foundation*, Addison-Wesley, Reading.
11. Christopher, T. W., and G. K. Thiruvathukal (2001): *Multithreaded and networked programming*, Sun Microsystems.
12. Courtois, P. J., F. Heymans, and D. L. Parnas (1971): “Concurrent control with readers and writers, *Communications of the ACM*, **14** (10), 667–668.
13. Dijkstra, E. W. (1965): “Cooperating sequential processes,” Technical Report EWD-123, Technological University, Eindhoven.
14. Francez, N., and A. Pneuli (1978): “A proof method for cyclic programs,” *Acta Informatica*, **9**, 133–157.
15. Hoare, C. A. R. (1972): “Towards a theory of parallel programming,” in *Operating Systems Techniques*, Hoare, C.A.R., and R.H. Perrot (Ed.), Academic Press, London, 1972.
16. Hoare, C. A. R (1974): “Monitors : an operating system structuring concept,” *Communications of the ACM*, **17** (10), 549–557.
17. Kosaraju, S. (1973): “Limitations of Dijkstra's semaphore primitives and petri nets,” *Operating Systems Review*, **7**, 4, 122–126.
18. Lamport, L. (1974): “A new solution of Dijkstra's concurrent programming problem,” *Communications of the ACM*, **17**, 453–455.

19. Lamport, L. (1979): "A new approach to proving the correctness of multiprocess programs," *ACM Transactions on Programming Languages and Systems*, **1**, 84–97.
20. Lamport, L. (1986): "The mutual exclusion problem," *Communications of the ACM*, **33** (2), 313–348.
21. Lamport, L. (1991): "The mutual exclusion problem has been solved," *ACM Transactions on Programming Languages and Systems*, **1**, 84–97.
22. Lipton, R. (1974): "On synchronization primitive systems," Ph.D. Thesis, Carnegie-Mellon University.
23. Love, R. (2005): *Linux Kernel Development, Second edition*, Novell Press.
24. Mauro, J., and R. McDougall (2001): *Solaris Internals—Core Kernel Architecture*, Prentice-Hall.
25. Owicki, S., and D. Gries (1976): "Verifying properties of parallel programs : An axiomatic approach," *Communications of the ACM*, **19**, 279–285.
26. Patil, S. (1971): "Limitations and capabilities of Dijkstra's semaphore primitives for co-ordination among processes," Technical Report, MIT.
27. Peterson, G. L. (1981): "Myths about the mutual exclusion problem," *Information Processing Letters*, **12**, 3.
28. Raynal, M. (1986): *Algorithms for Mutual Exclusion*, MIT Press, Cambridge.
29. Richter, J. (1999): *Programming Applications for Microsoft Windows, Fourth edition*, Microsoft Press, Redmond.
30. Russinovich, M. E., and D. A. Solomon (2005): *Microsoft Windows Internals, Fourth edition*, Microsoft Press, Redmond.
31. Stevens, W. R., and S. A. Rago (2005): *Advanced Programming in the Unix Environment, Second edition*, Addison Wesley Professional.
32. Vahalia, U. (1996): *Unix Internals—The New Frontiers*, Prentice-Hall, Englewood Cliffs.

Message Passing

We discussed four kinds of process interaction in Section 3.6—data access synchronization, control synchronization, message passing, and exchange of signals. Each kind of interaction suits a different purpose. Data access synchronization and control synchronization have been dealt with extensively in Chapter 9. A kernel provides signals to convey occurrence of exceptional situations to a process, hence use of signals has a standard syntax and semantics. Interprocess messages (or simply, messages) are provided to support exchange of information. Message passing uses the syntax supported by the kernel, but information conveyed in a message does not have a standard syntax and semantics; the receiver process decides how to deduce its meaning.

Message passing suits a variety of situations where exchange of information plays a key role. Its best use is in a service paradigm, where a process sends a message to another process to use some service offered by it. Due to its generality, message passing can also be used for other forms of process interaction like data access synchronization and control synchronization; however, it is much more convenient to use special synchronization features discussed in Chapter 9 for these purposes.

The generality and utility of message passing depends on features of the system calls used to send and receive messages, and on the arrangements used by the kernel to store and deliver messages. We describe different message passing arrangements and discuss their significance for user processes and for the kernel. We also describe message passing in Unix and Windows operating systems.

10.1 OVERVIEW OF MESSAGE PASSING

As discussed in Section 3.6, processes interact with one another in four ways— data access synchronization, control synchronization, message passing, and exchange of signals. It is important to differentiate between exchange of messages and exchange of signals. Signals have a standard syntax and semantics and are used to implement

pre-defined interactions between processes. Interprocess messages (or simply, messages) aim at generality of interaction—a message does not have a standard syntax and semantics; the receiver process decides how to deduce its meaning.

Message passing suits situations where processes wish to exchange information, or situations where one process wishes to influence, and possibly alter, the course of execution of another process. Message passing can also be used for other forms of interaction like data access synchronization and control synchronization; however, it is much simpler to use special synchronization features discussed in Chapter 9 for these purposes.

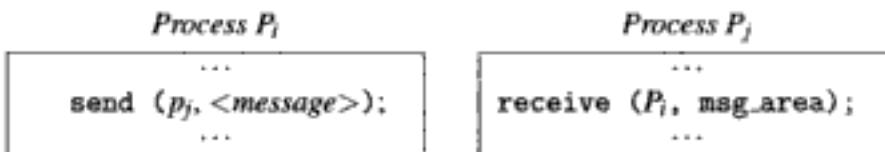


Fig. 10.1 Message passing

Figure 10.1 shows an example of message passing. Process P_i sends a message to process P_j by executing the statement `send (P_j , <message>)`, where `send` is a library module. `send` makes a system call `send` with P_j and the message as parameters. Execution of the statement `receive (P_i , msg_area)`, where `msg_area` is an area in P_j 's address space, results in a system call `receive`.

At a `send` call by P_i , the kernel checks whether process P_j is blocked on a `receive` call, in which case it copies the message into `msg_area` and activates P_j . If process P_j has not already executed a `receive` call, the kernel arranges to deliver the message to it when it executes a `receive` call sometime in future. When process P_j receives the message, it interprets the message and takes appropriate action. This may result in a reply being sent to P_i by executing a `send` statement.

In principle, message passing can be performed using shared variables. For example, `msg_area` in Figure 10.1 could be a shared variable. P_i could deposit a value or a message in the shared variable and P_j could collect it from there. However, this approach requires every pair of communicating processes to create a shared variable. They would have to estimate the correct size of the shared variable and share its name. They would also have to use synchronization analogous to the producers-consumers problem (see Section 9.5.1) to ensure that a receiver process would access a message in a shared variable only after a sender process has deposited it there. Interprocess messages are simpler to use because the kernel is responsible for buffering and delivery of messages. Further, messages become indispensable in a distributed system environment where the shared variable approach is not feasible. Other advantages of interprocess messages have been described earlier in Section 3.6.3.

Issues in message passing Table 10.1 describes the issues in message passing. These issues dictate implementation arrangements and also influence the power and

Table 10.1 Issues in message passing

Issue	Important aspects
Naming of processes	Processes participating in a message transfer are either explicitly indicated in <code>send</code> and <code>receive</code> statements, or they are deduced by the kernel in some other manner.
Method for transferring messages	Whether a sender process is blocked until the message sent by it is delivered, the order in which messages are delivered to the receiver process, and handling of exceptional conditions in message passing.
Kernel responsibilities	Buffering of messages pending delivery to recipient processes. Blocking and activation of processes.

generality of message passing. For example, requiring a sender process to know the identity of a receiver process may limit the scope of message passing to processes in the same application program. Relaxing this requirement would extend message passing to processes of different application programs and even to processes executing in different computer systems. Similarly, providing FIFO message delivery may be rather restrictive; processes may wish to receive messages in some order of their own choice.

System calls `send` and `receive` act as communication primitives. The number and meaning of their parameters are determined by the issues described in Table 10.1. We discuss various aspects of these primitives in the following Sections.

10.1.1 Direct and Indirect Naming

`send` and `receive` statements shown in Figure 10.1 use *direct naming*. Here, sender and receiver processes mention each other's names using the following syntax:

```
send (<destination_process>, <message>);
receive (<source_process>, <message_area>);
```

where `<destination_process>` and `<source_process>` are process names (typically, they are process id's assigned by the kernel), `<message>` is the textual form of the message to be sent, and `<message_area>` is a memory area in the receiver's address space where the message is to be delivered.

In *symmetric naming*, both sender and receiver processes have to specify each other's names. This requirement seems quite natural, however it raises two difficulties. First, processes that do not belong to the same application do not know each other's names. Second, this requirement is inconvenient for server processes. For example, a print server in an OS receives print requests from all processes in the system. It does not know which process will send it the next print request, so it cannot specify the `<source_process>` in a `receive` call. *Asymmetric naming* is used in

such situations. The `send` and `receive` statements using asymmetric naming have the following syntax:

```
send (<destination_process>, <message>);
receive (<message_area>);
```

Here the receiver does not name the process from which it wishes to receive a message; the kernel gives it a message sent to it by *some* process.

In *indirect naming*, processes do not mention each other's names in `send` and `receive` statements. We discuss indirect naming in Section 10.3.

10.1.2 Blocking and Non-blocking Sends

The `send` primitive has two variants. A blocking `send` blocks a sender process till the message being sent is delivered to the destination. A non-blocking `send` permits a sender to continue execution after executing a `send` irrespective of whether the message is delivered immediately. The `receive` primitive is typically blocking. Thus a process performing a `receive` must wait until a message can be delivered to it. Message passing using blocking and non-blocking `sends` are also known as *synchronous* and *asynchronous* message passing, respectively.

Message passing using blocking `sends` provides some nice properties for both user processes and the kernel. A sender process has a guarantee that the message sent by it is delivered before it continues its execution. This property simplifies the design of concurrent processes. The kernel actions are also simpler because the kernel can allow a message to remain in the sender's memory area until it is delivered. However blocking may unnecessarily delay a sender process in certain situations—for example, while communicating with a heavily loaded print server.

When a non-blocking `send` is used, a sender is free to continue its execution immediately after sending a message. However, it has no means of knowing when (and whether) its message is delivered to the receiver, hence it does not know when it is safe to reuse the memory area containing the text of a message. To overcome this problem, the kernel allocates a buffer in the system area and copies a message into the buffer. This feature involves substantial memory commitment when many undelivered messages exist. It also consumes CPU time as a message has to be copied twice—once into a system buffer when a `send` call is executed, and later into the message area of the receiver at the time of message delivery.

Example 10.1 Consider a process P_i with the following text:

```
Send ( $P_j$ , " ... ");
...
Send ( $P_k$ , " ... ");
...
```

If a blocking `send` is used, process P_i would be blocked on the first `send` until process P_j executes a matching `receive` statement. Thus, its progress would depend on when

processes P_j and P_k perform their `receive`'s. If a non-blocking `send` is used, P_i can make progress independent of P_j and P_k . However, the kernel must buffer its messages to P_j and P_k until they are delivered. Process P_i has no means of knowing whether the messages have been delivered to P_j and P_k .

10.1.3 Exceptional Conditions

Several exceptional conditions can arise during message delivery. Some of these are:

1. The destination process mentioned in a `send` does not exist.
2. In symmetric naming the source process mentioned in a `receive` does not exist.
3. A `send` cannot be executed because the kernel has run out of buffer memory.
4. No message exists for a process when it executes a `receive` statement.

In cases 1 and 2, the process executing the `send` or `receive` statement that leads to the exception may be canceled and its termination code may be set to describe the exceptional condition. In case 3, the sender process may be blocked until some buffer space becomes available due to a `receive` executed by some other process. Case 4 is really not an exception if `receives` are blocking (they generally are!), but it may be treated as an exception to provide some flexibility to the receiving process. For example, the kernel may return an appropriate status code to the process when this condition arises. If the status code is 'no messages available', the receiver process has the choice to either wait for arrival of a message, or re-execute `areceive` after some time. In the former case, it may repeat the `receive` statement with a flag to indicate that occurrence of this condition should be suppressed. To support these options the following syntax could be used for `areceive` statement:

```
receive (<source_process>, flags, status_area, <message_area>);
```

where `<source_process>` is specified only when symmetric naming is used and `status_area` is the area where the status information should be returned by the kernel. The `receive` library module passes these parameters to the kernel in a `receive` system call. Kernel actions concerning exceptional conditions are governed by `flags`. The `receive` call returns after putting status information in `status_area`.

More severe exceptions belong to the OS domain. These include communication deadlocks, wherein processes enter into a mutual wait condition due to `receive`'s, or difficult-to-handle situations like a process waiting a long time on a `receive` call.

10.2 IMPLEMENTING MESSAGE PASSING

10.2.1 Buffering of Interprocess Messages

When a process P_i sends a message to some process P_j using a non-blocking `send`, the kernel builds an *interprocess message control block* (IMCB) to store all information

ing process. This scheme becomes further simplified if fixed length messages are used in the system. If message lengths are variable, or non-blocking *sends* are used, the kernel may have to reserve a considerable amount of memory for interprocess messages. In such cases, it may save message texts on the disk. An IMCB would then contain address of the disk block where the message is stored, rather than the message text itself.

10.2.2 Delivery of Interprocess Messages

Two possibilities arise when a process P_i sends a message to process P_j :

- Process P_j has already performed a *receive* and is in the *blocked* state.
 - Process P_i has not performed a *receive* but may perform it sometime in future.

In the former case, the kernel must arrange to deliver the message to P_j straightaway, and change its state to *ready*. In the latter case, the kernel must deliver the message when P_i performs a matching *receive*.

Message delivery actions of the kernel are triggered by *send* and *receive* events, so they can be implemented using event control blocks (ECBs). As discussed in Section 3.3.6, an ECB is created for an event when some process decides to wait for its occurrence. An ECB contains three fields:

- Description of an anticipated event
 - Id of the process that awaits this event
 - An ECB pointer for forming ECB lists.

An ECB is used to activate the awaiting process when the anticipated event occurs.

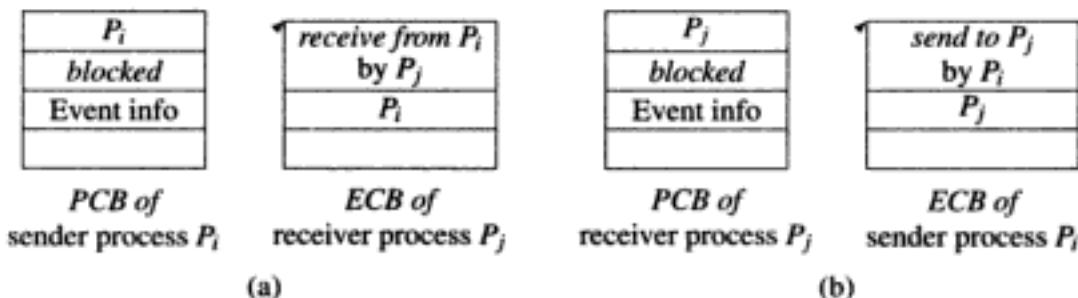


Fig. 10.4 ECBs in symmetric naming and blocking sends (a) at *send*, (b) at *receive*

Figure 10.4 shows use of ECBs to implement message passing with symmetric naming and blocking *sends*. Let process P_i send a message to process P_j . Figure 10.4(a) illustrates the case when process P_i executes a *send* call before P_j executes a *receive* call. When control reaches the event handler for *send* events, the handler checks and finds that process P_j is currently not blocked for a message, i.e., the *send* event was not anticipated. The event handler knows that process P_i is to be blocked

awaiting delivery of the message when P_j performs a *receive*, that is, a *receive* event is now anticipated. So it creates an ECB for the event ‘*receive from P_i by P_j* ’ and puts P_i as the identity of the process that would be affected by the event. Process P_i is put into the *blocked* state and the address of the ECB is put in the event info field of P_i ’s PCB.

Figure 10.4(b) illustrates the case when process P_j executes a *receive* call before P_i executes a *send* call. An ECB for a ‘*send to P_j by P_i* ’ event is created, and the id of P_j is put in the ECB to indicate that the state of P_j would be affected when the *send* event occurs.

At *send to P_j by P_i* :

- S_1 Create an IMCB and initialize its fields;
- S_2 If an ECB for a ‘*send to P_j by P_i* ’ event exists
then
 - (a) Deliver the message to P_j ;
 - (b) Activate P_j ;
 - (c) Destroy the ECB and the IMCB;
 - (d) Return to P_i ;
- S_4 else
 - (a) Create an ECB for a ‘*receive from P_i by P_j* ’ event and
put id of P_i as the process awaiting the event;
 - (b) Change the state of P_i to blocked and put the ECB
address in P_i ’s PCB;
 - (c) Add the IMCB to P_j ’s IMCB list;

At *receive from P_i by P_j* :

- R_1 If a matching ECB for a ‘*receive from P_i by P_j* ’ event exists
then
 - (a) Deliver the message from appropriate IMCB in P_j ’s list;
 - (b) Activate P_i ;
 - (c) Destroy the ECB and the IMCB;
 - (d) Return to P_j ;
- R_3 else
 - (a) Create an ECB for a ‘*send to P_j by P_i* ’ event and
put id of P_j as the process awaiting the event;
 - (b) Change the state of P_j to blocked and put the ECB
address in P_i ’s PCB;

Fig. 10.5 Kernel actions in message passing using symmetric naming and blocking *sends*

Figure 10.5 shows complete details of kernel actions to implement message passing using symmetric naming and blocking *sends*. For reasons mentioned earlier, the kernel creates an IMCB even though a sender or receiver process is blocked until message delivery. When process P_i sends a message to process P_j , the kernel first checks whether the *send* is anticipated, i.e., whether an ECB was created for the *send*

event. This would have happened if process P_j had already executed a *receive* call for a message from P_i . If this is the case, action S_3 immediately delivers the message to P_j and changes its state from *blocked* to *ready*. The ECB and the IMCB are now destroyed. If an ECB for *send* does not exist, action S_4 creates an ECB for a *receive* call by process P_j , which is now anticipated, blocks the sender process, and enters the IMCB in the IMCB list of process P_j . Converse actions are taken at a *receive* call. A message is delivered to process P_j if a matching *send* had already occurred, and P_i is activated; otherwise, an ECB is created for a *send* call and P_j is blocked.

Actions when non-blocking *sends* are used are analogous except for blocking and activation of the sender (see actions $S_4(b)$ and $R_2(b)$ in Figure 10.5). Creation of an ECB when a message being sent cannot be delivered immediately (see action $S_4(a)$) is also unnecessary since a sender does not wait for a message to be received.

10.3 MAILBOXES

A mailbox is a repository for interprocess messages. It has a unique identity. The owner of a mailbox is typically the process that created it. Only the owner process can receive messages from a mailbox. Any process that knows the identity of a mailbox can send messages to it. We will call these processes the *users* of a mailbox.

Figure 10.6 illustrates message passing using a mailbox named `sample`. Process P_i creates the mailbox using the call `create_mailbox`. Process P_j sends a message to the mailbox using the mailbox name, instead of a process id, in its *send* call. This is an instance of indirect naming. A message sent to `sample` is stored in a buffer. It is delivered to process P_i when P_i performs a *receive* on `sample`. Some other processes, e.g., P_k and P_l , might also send messages to `sample`. Both `create_mailbox` and *send* calls would return with status codes to indicate success or failure of the call. The kernel may associate a fixed set of buffers with each mailbox, or it may allocate buffers from a common pool of buffers when a message is sent.

A kernel may provide a fixed set of mailbox names. Alternatively, it may permit user processes to create names of their choice. In the former case, confidentiality of communication between a pair of processes cannot be guaranteed because any process can create or connect to any mailbox. This problem becomes less severe when processes can assign mailbox names of their own choice. In Section 10.4, we discuss the Unix facility for naming of mailboxes.

To exercise control over creation and destruction of mailboxes, the kernel may require a process to explicitly ‘connect’ to a mailbox before starting to use it, and to ‘disconnect’ after finishing its use. This way it can destroy a mailbox to which no process is connected. Alternatively, the kernel may permit the owner of a mailbox to destroy it. In that case, the kernel has to notify destruction of the mailbox to all users who have ‘connected’ to it. The kernel may permit the ownership of a mailbox to be transferred to another process.

Advantages of mailboxes Use of a mailbox has following advantages:

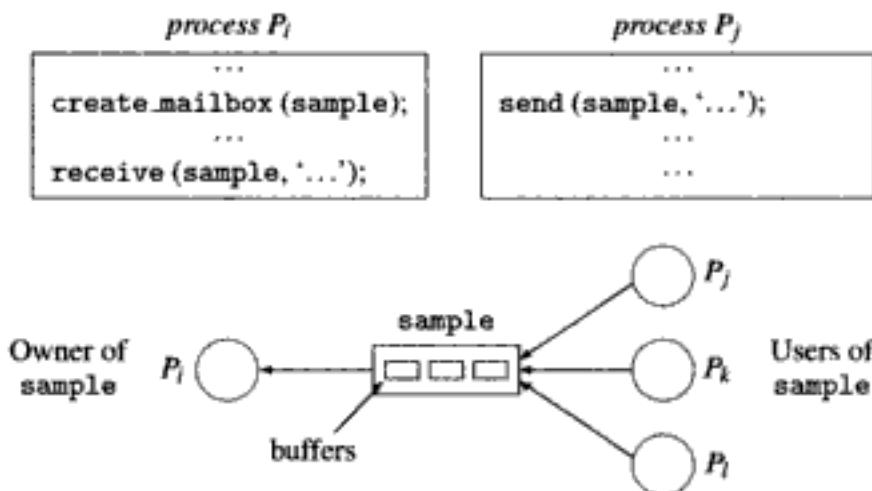


Fig. 10.6 A mailbox

- *Anonymity of receiver:* A process sending a message to a mailbox need not know identity of the receiver process. If an OS permits the receiver of a mailbox to be changed dynamically, a process can take over the functionality of another process.
- *Classification of messages:* A process may create several mailboxes, and use each mailbox to receive messages of a specific kind. This arrangement permits easy classification of messages (see Example 10.2).

Anonymity of a receiver process can be used to transfer a function from one process to another. Consider an OS whose kernel is structured in the form of multiple processes communicating through messages. Interrupts relevant to the process scheduling function can be modeled as messages sent to a mailbox named `scheduling`. If the OS wishes to use different process scheduling criteria during different periods of the day, it may implement different schedulers as processes and pass ownership of the `scheduling` mailbox between these processes. This way, the process scheduler, which currently owns `scheduling`, can receive all scheduling related messages. Functionalities of OS servers can be similarly transferred. For example, all print requests can be directed to a laser printer instead of a dot matrix printer by simply changing the ownership of a `print` mailbox.

Although a process can send a message to a mailbox anonymously, it is often necessary to know the identity of the sender. For example, a server would like to return a status code for each request made to it. This can be achieved by passing the sender's id along with the text of the message. The sender of a message would not know the identity of the server, hence it must receive the server's reply using an asymmetric `receive`. If this is not desired, `send_to_mailbox` can be provided as a blocking call with an implicit return message. Now, return of a status code would become a part of the convention for a server call.

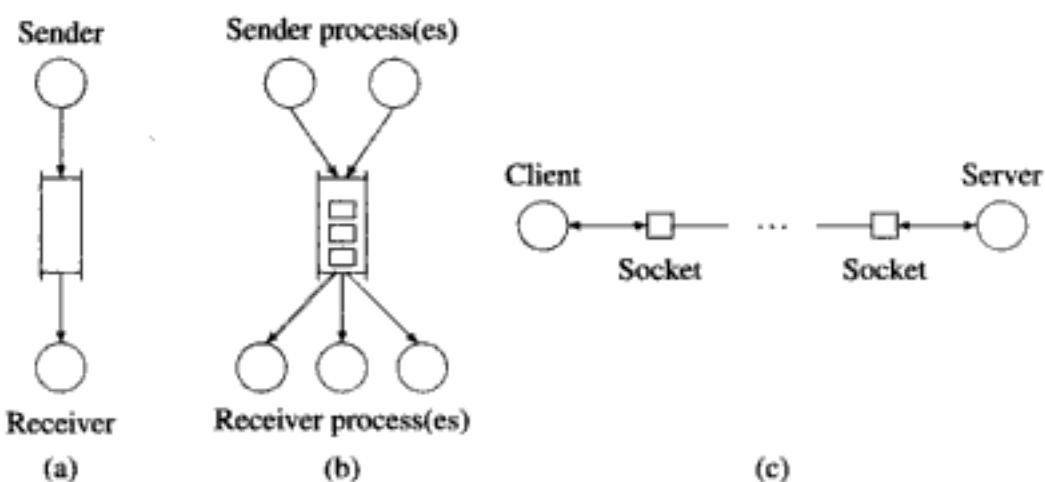


Fig. 10.8 Interprocess communication in Unix: (a) Pipe, (b) message queue, (c) socket

through the call *pipe*.

The kernel operates a pipe as a queue by maintaining two offsets—one offset is used for writing data into a pipe and another for reading data from the pipe. A pipe is implemented like a file, except for two differences. The size of a pipe is limited so that data in a pipe is located in the direct blocks of the inode (see Section 7.12.1). The read and write offsets are maintained in the inode instead of in the file structure, which forbids a process from changing the offset of a pipe through any means other than reading or writing data. If a pipe is full, the process wishing to write data into it is put to sleep. Similarly, a process reading data from a pipe is put to sleep if the pipe is empty.

Message queues The concept of a message queue is analogous to the concept of a mailbox. A message queue is created and owned by one process. Other processes can send or receive messages to or from a queue in accordance with access permissions specified by the creator of the message queue. These permissions are specified using the same conventions as file permissions in Unix. A message queue has a finite capacity in terms of the number of bytes that it can buffer.

A message queue is created by a *msgget* call *msgget (key, flag)* where *key* specifies the name of the message queue and *flag* indicates some options. The kernel maintains an array of message queues and their keys. The first *msgget* call with a given key results in creation of a new message queue. The process executing the call becomes the owner of the message queue. The position of the message queue in the system array becomes the message queue id. It is returned by the *msgget* call. The process uses this id for sending or receiving messages. The naming issue is tackled as follows: If a process executes a *msgget* call with a key that matches the name of an existing message queue, the kernel simply returns its message queue id. This way a message queue can be used by any process in the system.

Each message consists of a message type that is an integer and a message text.

BIBLIOGRAPHY

Interprocess communication in the RC4000 system is described in Brinch Hansen (1970). Accetta *et al* (1986) discuss the scheme used in Mach. Bach (1986), McKusick *et al* (1996), Vahalia (1996), and Stevens and Rago (2005) discuss message passing in Unix. Bovet and Cesati (2003) discuss message passing in Linux, while Russinovich and Solomon (2005) discuss message passing in Windows.

Geist *et al* (1996) describe and compare the PVM and MPI message passing standards for parallel programming.

1. Accetta, M., R. Baron, W. Bolosky, D. B. Golub, R. Rashid, A. Tevanian, and M. Young (1986) : "Mach: A new kernel foundation for Unix development," *Proceedings of the Summer 1986 USENIX Conference*, June 1986, 93–112.
2. Bach, M. J. (1986): *The Design of the Unix Operating System*, Prentice-Hall, Englewood Cliffs.
3. Bovet, D. P., and M. Cesati (2003): *Understanding the Linux Kernel*, O'Reilly, Sebastopol.
4. Brinch Hansen, P. (1970) : "The nucleus of a multiprogramming system," *Communications of the ACM*, 13 (4), 238–241, 250.
5. Geist, G., J. A. Kohl, and P. M. Papadopoulos (1996): "PVM and MPI: a comparison of features," *Calculateurs Paralleles*, 8 (2).
6. McKusick, M. K., K. Bostic, M. J. Karels, and J. S. Quarterman (1996): *The Design and Implementation of the 4.4 BSD Operating System*, Addison Wesley, Reading.
7. Russinovich, M. E., and D. A. Solomon (2005): *Microsoft Windows Internals, Fourth edition*, Microsoft Press, Redmond.
8. Stevens, W. R., and S. A. Rago (2005): *Advanced Programming in the Unix Environment, Second edition*, Addison Wesley Professional.
9. Tanenbaum, A. S. (2001): *Modern Operating Systems, Second edition*, Prentice-Hall, Englewood Cliffs.
10. Vahalia, U. (1996): *Unix Internals—The New frontiers*, Prentice-Hall, Englewood Cliffs.

Deadlocks

A *deadlock* is a situation in which some processes wait for each other's actions indefinitely. It can arise in a synchronization situation, e.g., when processes wait for other processes to send them messages, or to release resources that they need. In real life, deadlocks can arise when two persons wait for phone calls from one another, or when persons crossing a narrow staircase in opposite directions meet in the middle of the staircase.

Processes involved in a deadlock remain blocked permanently. This affects OS performance indices like throughput and resource efficiency. A deadlock involving OS processes can have more severe consequences; it may bring operation of the OS to a standstill.

Operating systems handle only deadlocks caused by sharing of resources in the system. Such deadlocks arise when some conditions concerning resource requests and resource allocations hold simultaneously. Deadlock handling approaches use these conditions as the basis for their actions. *Deadlock detection* detects a deadlock by checking whether all conditions necessary for a deadlock hold simultaneously. *Deadlock resolution* removes a deadlock by aborting some processes so that other processes involved in the deadlock can resume their operation. The *deadlock prevention* and *deadlock avoidance* approaches ensure that deadlocks cannot occur, by not allowing the conditions for deadlocks to hold simultaneously.

Operating systems use resource allocation policies that ensure absence of deadlocks. We discuss these policies and relate them to the fundamental deadlock handling approaches.

11.1 DEFINITION OF DEADLOCK

A set of processes is deadlocked if each of them waits for an event that can be caused only by process(es) in the set. Thus, each process waits for an event that cannot occur. This situation arises if the conditions in Def. 11.1 are satisfied.

Definition 11.1 (Deadlock) A deadlock involving a set of processes D is a situation in which each process P_i in D satisfies two conditions

1. Process P_i is blocked on some event e_j .
2. Event e_j can be caused only by actions of other process(es) in D .

Each process capable of causing event e_j for which process P_i waits itself belongs to D . This property makes it impossible for event e_j to occur, making process P_i wait indefinitely. Every other process in D also waits indefinitely.

Various kinds of deadlocks can arise in an OS. We have a resource deadlock if the event awaited by each process in D is the granting of some resource. A communication deadlock occurs when awaited events pertain to receipt of interprocess messages, and a synchronization deadlock occurs when awaited events concern exchange of signals between processes. An OS is primarily concerned with resource deadlocks because allocation of resources is an OS responsibility. The other two forms of deadlock are seldom tackled by an OS.

11.2 DEADLOCKS IN RESOURCE ALLOCATION

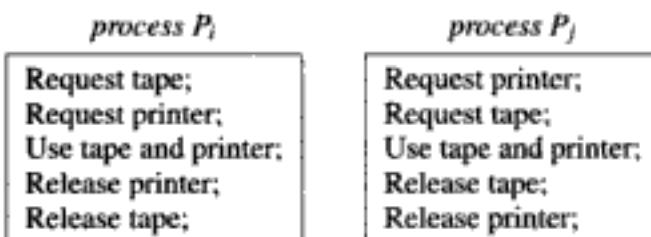
We differentiate between a resource unit, which is a resource of a specific kind, and a resource class, which is the collection of resource units of a specific kind. For example, the printer class contains all printer units in the system. We use the notation R_i for a resource class, and r_j for a resource unit in a resource class.

Three events concerning resource allocation can occur in a system—request for a resource, allocation of a resource, and release of a resource. Table 11.1 describes these events. A request event occurs when some process P_i makes a request for a resource r_l . If r_l is currently allocated to some process P_k , process P_i gets blocked on an allocation event for r_l . In effect, P_i is waiting for P_k to release r_l . A release event by P_k frees resource r_l . It can cause the allocation event for which P_i is waiting. Process P_i will face an indefinite wait if P_k 's release of r_l is indefinitely delayed. Example 11.1 illustrates how this can happen.

Table 11.1 Events related to resource allocation

Event	Description
Request	A process requests a resource through a system call. If the resource is free, the kernel allocates it to the process immediately; otherwise, it changes the state of the process to <i>blocked</i> .
Allocation	The process becomes the <i>holder</i> of the resource allocated to it. The resource state information is updated and the state of the process is changed to <i>ready</i> .
Release	A process releases a resource through a system call. If some processes are blocked on the allocation event for the resource, the kernel uses some tie-breaking rule, e.g., FCFS allocation, to decide which process should be allocated the resource.

Example 11.1 A system contains one tape and one printer and two processes P_i and P_j that use the tape and the printer as follows:



Resource requests by P_i and P_j take place in the following order:

1. Process P_i requests the tape
2. Process P_j requests the printer
3. Process P_i requests the printer
4. Process P_j requests the tape.

The first two resource requests are granted straightaway because a tape and a printer exist in the system. Now, P_i holds the tape and P_j holds the printer. When P_i asks for the printer, it is blocked until P_j releases the printer. Similarly, P_j is blocked until P_i releases the tape. From Def. 11.1, the set of processes $\{P_i, P_j\}$ is deadlocked.

11.2.1 Conditions for a Resource Deadlock

We can interpret Parts 1 and 2 of Def. 11.1 to determine the conditions under which resource deadlocks can occur. A process $P_j \in D$ is blocked for an allocation event to occur, which can be caused only by actions of some other process $P_k \in D$. Since $P_k \in D$, Parts 1 and 2 of Def. 11.1 apply to P_k as well. In other words, the resource required by process P_j is allocated to P_k , which itself waits for some other resource to be allocated to it. This condition is called the *hold-and-wait* condition.

Parts 1 and 2 of Def. 11.1 also imply that processes in D must wait for each other. This condition is called the *circular wait* condition. A circular wait may be direct, that is, P_i waits for P_k and P_k waits for P_i , or it may be through one or more other processes included in D , for example P_i waits for P_k , P_k waits for P_l and P_l waits for P_i . Other obvious conditions for resource deadlocks are that if process P_i needs a resource that is currently allocated to P_k , P_i must not be able to either share the resource with P_k or snatch it away from P_k and use it.

Table 11.2 summarizes conditions that must be satisfied for a resource deadlock to exist. All these conditions must hold simultaneously because a circular wait is essential for deadlocks, a hold-and-wait condition is essential for a circular wait, and non-shreability and non-preemptibility of resources are essential for a hold-and-wait condition.

The following condition is also essential for deadlocks to exist

Condition	Explanation
No withdrawal of resource requests	A process blocked on a resource request cannot withdraw its request.

Table 11.2 Conditions for resource deadlock

Condition	Explanation
Non-shareable resources	Resources cannot be shared. A process needs exclusive access to a resource.
Hold-and-wait	A process continues to hold the resources allocated to it while waiting for other resources.
No preemption	OS cannot preempt a resource from one process in order to allocate it to another process.
Circular waits	A circular chain of hold-and-wait conditions exists in the system.

Waits may not be indefinite if a process is permitted to withdraw a resource request and resume its execution. This condition is not stated explicitly in literature because resource requests in a general purpose OS typically possess this feature.

11.2.2 Resource State Modeling

From Example 11.1, it is clear that we must analyze resource requests and the resource allocation state of the system (hereafter simply called the ‘resource state’) to determine whether a set of processes is deadlocked.

Two kinds of models are used to represent the allocation state of a system. A *graph model* can depict the allocation state of a restricted class of systems in which a process can request and use exactly one resource unit of each resource class. It permits use of a simple graph algorithm to determine whether the circular wait condition is satisfied by processes. A *matrix model* has the advantage of generality. It can model allocation state in systems that permit a process to request any number of units of a resource class.

11.2.2.1 Graph Models

A *resource request and allocation graph* (RRAG) contains two kinds of nodes—process nodes and resource nodes. A process node is depicted by a circle. A resource node is depicted by a rectangle and represents one class of resources. The number of bullet symbols in a resource node indicates how many units of that resource class exist in the system. Two kinds of edges can exist between a process node and a resource node. An *allocation edge* is directed from a resource node to a process node. It indicates that one unit of the resource class is allocated to the process. A *request edge* is directed from a process node to a resource node and indicates that the process is blocked on a request for one unit of the resource class. A request edge is replaced by an allocation edge when a pending request is granted. An allocation edge is deleted when a process releases a resource unit allocated to it.

A *wait-for graph* (WFG) can represent the resource allocation state more con-

$WF_{n-1} = \{P_n\}$ violates condition (11.2). Any other subset of $\{P_1, \dots, P_n\}$ similarly violates condition (11.2) for some process. Hence there is no deadlock in the system.

Now let $P_n \equiv P_1$, i.e., we have a cycle in RRAG and WFG. Processes in the set $\{P_1, \dots, P_n\}$ now satisfy conditions (11.1) and (11.2) since

- $\{P_1, \dots, P_n\} \subseteq Blocked.P$
- For all $P_i \in \{P_1, \dots, P_n\}$, WF_i contains a single process P_l such that $P_l \in \{P_1, \dots, P_n\}$.

Hence a deadlock exists. From this analysis we can conclude that condition (11.2), which implies existence of mutual wait-for relationships between processes of D , can be satisfied only by cyclic paths, so a deadlock cannot exist unless an RRAG, or a WFG, contains a cycle.

Example 11.3 Figure 11.2 contains the RRAG for Example 11.1. The RRAG contains a cyclic path $P_i \rightarrow \text{printer} \rightarrow P_j \rightarrow \text{tape} \rightarrow P_i$. Here $WF_i = \{P_j\}$ and $WF_j = \{P_i\}$. $D = \{P_1, P_2\}$ satisfies conditions (11.1) and (11.2). Hence processes P_i and P_j are deadlocked.



Fig. 11.2 RRAG for the system of Ex. 11.1

Paths in an RRAG differ from those in a WFG when resource classes contain multiple resource units. Consider a path $P_1 \rightarrow R_1 \dots P_i \rightarrow R_i \rightarrow P_{i+1} \dots P_n$ such that a resource class R_i contains many resource units and some process P_k not included in the path holds one unit of resource class R_i . If P_k releases the unit allocated to it, that unit may be allocated to P_i . The edge (P_i, P_{i+1}) would thus vanish even without P_{i+1} releasing the unit of R_i held by it. Thus, a cyclic path in an RRAG may be broken when some process not included in the cycle releases a unit of the resource. Thus, presence of a cycle does not necessarily imply existence of a deadlock if a resource class contains more than one resource unit. We use this fact in Section 11.7 when we develop a formal characterization for deadlocks. Example 11.4 illustrates this situation.

Example 11.4 An OS contains one printer and two tapes, and three processes P_i , P_j and P_k . The nature of processes P_i and P_j is same as depicted in Example 11.1—each of them requires a tape and a printer. Process P_k requires only a tape unit for its execution. Let process P_k request for a tape before requests 1–4 are made as in Example 11.1.

Figure 11.3 shows the RRAG after all requests have been made. A cycle involving P_i and P_j exists in the graph. This cycle would be broken when process P_k completes

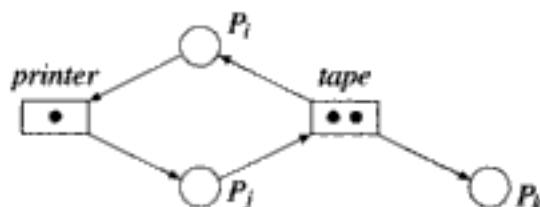


Fig. 11.3 RRAg after all requests of Ex. 11.4 are made

because the tape released by it can be allocated to P_j . Hence a deadlock does not exist. We come to the same conclusion when we analyze the set of processes $\{P_i, P_j\}$ according to Def. 11.1 because $WF_j = \{P_i, P_k\}$ and $P_k \notin \{P_i, P_j\}$ violates condition (11.2).

11.2.2.2 Matrix Model

In the matrix model, the allocation state of a system is primarily represented by two matrices. The matrix *allocated_resources* indicates how many resource units of each resource class are allocated to each process in the system. The matrix *Requested_resources* represents pending requests. It indicates how many resource units of each resource class are requested by each process in the system. If a system contains n processes and r resource classes, each of these matrices is an $n \times r$ matrix. The allocation state with respect to a resource class R_k indicates the number of units of R_k allocated to each process, and the number of units of R_k requested by each process. These are represented as n -tuples ($Allocated_resources_{1,k}, \dots, Allocated_resources_{n,k}$) and ($Requested_resources_{1,k}, \dots, Requested_resources_{n,k}$), respectively.

Some auxiliary matrices may be used to represent additional information required for a specific purpose. Two such auxiliary matrices are *Total_resources* and *free_resources*, which indicate the total number of units of each resource class and the number of units of each resource class that are free, respectively. Each of these matrices is a column matrix that has r elements in it. Example 11.5 is an example of a matrix model.

Example 11.5 Using the matrix model, the allocation state of the system of Figure 11.3 is represented as follows:

	Printer	Tape		Printer	Tape		Total resources	Printer	Tape
P_i	0	1		P_i	1	0		1	2
P_j	1	0		P_j	0	1		0	0
P_k	0	1		P_k	0	0			
Allocated resources					Requested resources				

The wait-for relationships in the system are not represented by the matrix model; they have to be deduced by an algorithm. Algorithms 11.1 and 11.2 discussed in later sections use the matrix model.

11.3 HANDLING DEADLOCKS

Table 11.3 describes three fundamental approaches used in deadlock handling. Each approach has different implications for user processes and for the OS.

Table 11.3 Deadlock handling approaches

Approach	Description
Deadlock detection and resolution	The kernel analyzes the resource state to check whether a deadlock exists. If so, it aborts some process(es) and allocates the resources held by them to other processes so that the deadlock ceases to exist.
Deadlock prevention	The kernel uses a resource allocation policy that ensures that the four conditions for resource deadlocks mentioned in Table 11.2 do not arise simultaneously. This approach makes deadlocks impossible.
Deadlock avoidance	The kernel analyzes the allocation state to determine whether granting a resource request might lead to a deadlock later on. Only requests that cannot lead to a deadlock are granted, others are kept pending until they can be granted.

When the *deadlock detection and resolution* approach is used, the kernel aborts some processes when it detects a deadlock and reallocates their resources to other requesters. Aborted processes have to be re-executed. In the system of Example 11.1, the kernel would detect a deadlock sometime after processing the fourth request. This deadlock can be resolved by aborting either P_i or P_j and allocating the resource held by the aborted process to the other process. The cost of this approach includes the cost of deadlock detection and the cost of re-executing the aborted process.

In *deadlock prevention*, a process has to conform to the resource usage rules stipulated by the kernel. For example, a simple deadlock prevention policy might require a process to make all its resource requests together, and the kernel would allocate all requested resources at the same time. This rule would prevent deadlocks, e.g., the deadlock in Example 11.1; however, the rule may force a process to obtain a resource long before it is needed.

When *deadlock avoidance* is used, the kernel grants a resource request only if it finds that granting the request would not lead to deadlocks in future. Other requests are kept pending until they can be granted. Thus, a process may face long delays in obtaining a resource. In Example 11.1, the kernel would realize the possibility of a deadlock while processing the second request, so it would not grant the printer to process P_j until process P_i completes.

The choice of the deadlock handling approach depends on relative costs of the approaches and their consequences in terms of overhead, delays in resource allocation and constraints imposed on user processes.

11.4 DEADLOCK DETECTION AND RESOLUTION

Deadlocks can be detected by checking for the presence of a cycle in an RRAG or WFG and ensuring that the graph satisfies the other conditions discussed in Section 11.7. However, a graph-based approach has the limitation that it cannot handle a process that requires more than one unit of a resource class, so we develop another approach that uses Def. 11.1 as the basis for deadlock detection.

Consider a system that does not contain a deadlock. It must contain at least one process that is not blocked on a resource request. All resources allocated to such a process would become free if the process completes its execution without requesting any more resources. These resources can be allocated to some processes that are blocked on their requests for these resources. These processes can complete and release all resources allocated to them, and so on, until all processes in the system can complete. Accordingly, we check for the presence of a deadlock in a system by trying to construct a sequence of resource allocations and releases through which all processes in the system can complete their execution. Success in constructing such a sequence implies the absence of a deadlock in the current state of the system and failure to construct it implies presence of a deadlock. Example 11.6 illustrates use of this approach.

Example 11.6 In Example 11.3 (see Figure 11.2) there is no sequence of allocations through which processes P_i and P_j can complete their execution. Hence processes P_i and P_j are in a deadlock.

In Example 11.4, the following sequence of events brings processes P_j and P_i out of the blocked state (see the RRAG of Figure 11.3):

1. Process P_k completes and releases the tape.
2. The tape is allocated to P_j . P_j now completes and releases the printer and the tape.
3. Process P_i is now allocated the printer and completes.

Hence a deadlock does not exist.

Deadlock analysis Analysis for the existence of deadlocks is performed by simulating operation of the system starting with its current state. In the simulation we consider only two events that are important for deadlock analysis—allocation of resource(s) to a process that is blocked on a resource request, and completion of a process that is not blocked on a resource request. We refer to any process that is not blocked on a resource request as a *running* process. The simulator tries to construct a sequence of resource allocations and releases through which all processes in the system, whether *running* or *blocked*, can complete their execution.

In the simulation we assume that a *running* process completes without making additional resource requests. When it completes, all resources allocated to it become free. These resources are allocated to a *blocked* process only if the allocation puts that process in the *running* state. The simulation ends when all *running* processes complete. If some processes are still in the *blocked* state, they must be waiting for

gorithm finds that process P_1 's pending request can now be satisfied, so it allocates the resources requested by P_4 and transfers P_4 to the set *Running* (see Figure 11.4(b)). Since P_4 is the only process in *Running*, it is transferred to the set *Finished*. After freeing P_4 's resources, the algorithm finds that P_1 's resource request can be satisfied (see Figure 11.4(c)) and, after P_1 completes, P_2 's resource request can be satisfied (see Figure 11.4(d)). The set *Running* is now empty so the algorithm completes. A deadlock does not exist in the system because the set *Blocked* is empty.

11.4.2 Deadlock Resolution

Given a set of deadlocked processes D , *deadlock resolution* implies breaking the deadlock to ensure progress for some processes $\{P_i\} \in D$. This can be achieved by satisfying the resource request of a process P_i in one of two ways:

1. Terminate some processes $\{P_j\} \subset D$ to free the resources required by $\{P_i\}$. (We call each process in $\{P_j\}$ a *victim* of deadlock resolution.)
2. Add new units of the resources requested by P_i .

Note that deadlock resolution only ensures some progress for a process P_i . It does not guarantee that P_i would run to completion. That would depend on the resource requests and allocations in the system after the deadlock is resolved. Example 11.9 illustrates how the RRAG and wait-for sets of processes change as a result of deadlock resolution.

Example 11.9 Figure 11.5 depicts deadlock resolution through termination of some deadlocked processes. The RRAG of Part (a) shows a deadlock involving processes P_1, P_2, P_3 and P_4 . This deadlock is resolved by choosing process P_2 as the victim. Part (b) of the figure shows the RRAG after terminating process P_2 and allocating resource R_3 held by it to process P_1 . The resource request made by P_1 is now satisfied, and process P_4 , which waited for the victim before deadlock resolution, now waits for P_1 , the new holder of the resource. This fact is important for deadlock detection in future.

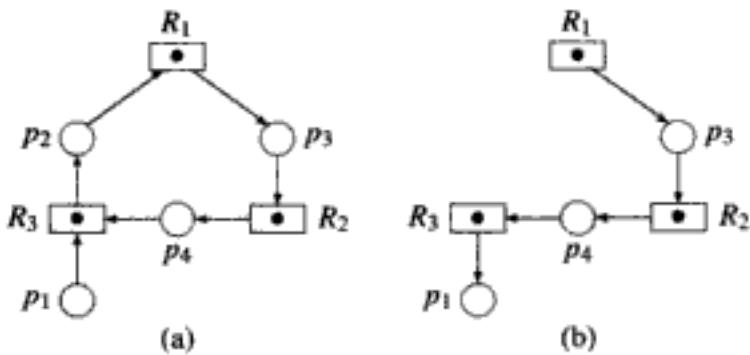


Fig. 11.5 Deadlock resolution

Addition of a new resource unit is impractical in most systems, with the exception of a system facing an interprocess communication deadlock. In such a system the kernel could create dummy messages and deliver them to some processes blocked

non-shareable mutable files.

Hold-and-wait Hold-and-wait can be prevented using two methods—either a process blocking on a request should not be permitted to hold any resources, or a process holding a resource should not be permitted to make resource requests. In the former method, a process can make a resource request only after releasing the resources held by it. It can request the released resource again when it needs. This method implies that a resource should be preemptible. In the latter method, a process may be forced to ask for all resources required by it in a single request so that it does not make another resource request in its lifetime. This method is more generally applicable. We discuss it in Section 11.5.1.

Preemption of resources Non-preemptibility is caused by the nature of a resource. In selected cases, it may be possible to circumvent this condition, e.g., the page formatting approach of the THE system can be used to make printers preemptible; however, in general, sequential I/O devices cannot be preempted.

Circular wait This condition implies existence of a cycle in a WFG or RRAG. It is a consequence of the hold-and-wait condition, which is a consequence of the non-shareability and non-preemptibility conditions. Hence a circular wait does not arise if any of the other conditions does not arise.

The circular wait condition can be independently prevented by applying *validity constraint* to each resource request. The validity constraint is a boolean function of the resource state of a requesting process. It evaluates to *false* if the request may lead to a circular wait in the system. Such a request is rejected straightaway. If the validity constraint evaluates to *true*, the resource is allocated if it is available; otherwise, the process is blocked for the resource. In Section 11.5.2 we discuss a deadlock prevention policy using this approach.

11.5.1 All Requests Together

This is the simplest of all deadlock prevention policies. A process must ask for all resources required by it in a single request. Typically, this request is made at the start of its execution. The kernel allocates all resources together, so a blocked process does not hold any resources, and the hold-and-wait condition is never satisfied. Consequently, circular waits and deadlocks cannot arise.

Example 11.10 In Example 11.1, both processes request a tape and a printer together. Now a process will either hold both resources or hold none of them, so the hold-and-wait condition is not satisfied.

Simplicity of implementation makes this policy attractive for small operating systems. However, it has one practical drawback—it adversely influences resource utilization efficiency. For example, even if a process requires a tape at the start of its execution and a printer only towards the end, it will be forced to request both tape and

printer at the start. The printer will remain idle until the latter part of its execution and any process requiring a printer will be delayed until P_i completes its execution. This situation also reduces the effective degree of multiprogramming and, therefore, reduces efficiency of CPU utilization.

11.5.2 Resource Ranking

A unique number called a *resource rank* is associated with each resource class. The validity constraint on a resource request made by a process P_i returns *true* only if the rank of the requested resource is larger than the rank of the highest ranked resource currently allocated to P_i . If the validity constraint evaluates to *true*, the resource is allocated to P_i if it is available; otherwise, P_i is blocked for the resource. If the validity constraint evaluates to *false*, the process making the request is aborted.

Consider a system consisting of five resources classes $R_1 \dots R_5$ that are assigned the resource ranks 1 ... 5, respectively. Process P_i currently holds a unit each of resource classes R_1 and R_3 . P_i would be aborted if it makes a request for R_1 , R_2 or R_3 since the rank of the requested resource does not exceed the rank of R_3 , the highest ranked resource held by P_i . If it makes a request for R_4 or R_5 , it would either be allocated the resource, or it would get blocked on the resource request.

The RRAG of Figure 11.7 illustrates the essence of resource ranking. The resources are shown such that their ranks are in ascending order from left to right. A process node is shown below the highest ranked resource allocated to it. Now, every request edge in the RRAG would point to the right in the RRAG. The RRAG shown in Figure 11.7(a) shows the situation when process P_i is allocated resources R_1 and R_3 and process P_j is allocated resource R_5 . When process P_i makes a request for R_5 , it is allowed to wait and the request edge (P_i, R_5) is added to the RRAG (see Figure 11.7(b)). When process P_j makes a request for R_3 , the request is rejected and P_j is aborted. Thus, no cycles can develop in the RRAG and no deadlocks can exist.

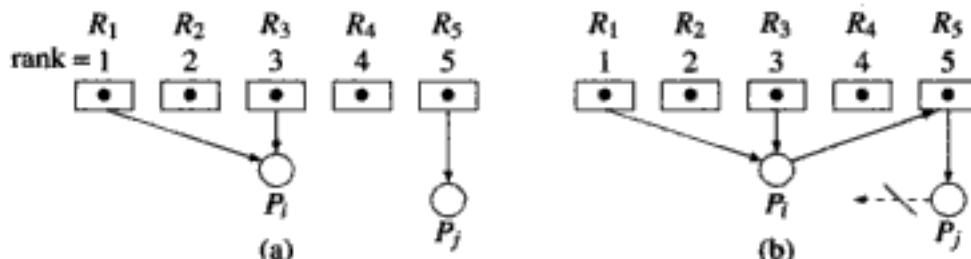


Fig. 11.7 Deadlock prevention using resource ranking

In general, absence of circular wait-for relationships in a system using resource ranking can be argued as follows: Let $rank_k$ denote the rank assigned to resource class R_k , and let process P_i hold some units of resource class R_k . If P_i gets blocked on a request for a unit of some resource class R_l , then $rank_k < rank_l$. Now consider a process P_j that holds some units of resource class R_l . Process P_j cannot request a

unit of resource class R_k since $rank_k > rank_l$. Thus, if P_i can wait for P_j , P_j cannot wait for P_i ! Hence, circular waits cannot arise.

Example 11.11 In Example 11.1, let $rank_{printer} > rank_{tape}$. Requests 1 and 2 lead to allocation of the tape and printer to P_i and P_j , respectively. The validity constraint evaluates to *true* for request 3; however, the request remains pending since the printer is not available. Request 4 would be rejected since its validity constraint evaluates to *false*, so process P_j would be aborted. Now the printer would become free. It would be allocated to process P_i .

The resource ranking policy works best when all processes require their resources in the order of increasing ranks. However, difficulties arise when a process requires resources in some other order. In Figure 11.7(a), if process P_i requires a unit of resource class R_2 , its resource request would violate the validity constraint. The only way P_i can obtain a unit of R_2 is by releasing the resource of R_3 before requesting the unit of R_2 . It would now have to re-acquire the unit of resource class R_3 . This is difficult in practice since most resources are non-preemptible.

Processes may tend to circumvent such difficulties by acquiring lower ranked resources much before they are actually needed. For instance, process P_i of Example 11.1 can acquire resource R_2 before it acquires resource R_3 . Similarly process P_j of Example 11.1 could acquire the tape before acquiring the printer. In the worst case, this policy may degenerate into the all-requests-together policy of resource allocation.

Despite these drawbacks the resource ranking policy is attractive due to its simplicity. A kernel can use this policy for its own resource requirements when it needs the resources in a fixed order. We shall discuss this aspect in Section 11.8.

11.6 DEADLOCK AVOIDANCE

Deadlock avoidance differs from deadlock prevention in one vital respect. It does not try to prevent any of the conditions for deadlock described in Section 11.2.1. Hence deadlocks are not impossible; however, the avoidance approach uses a resource allocation policy that grants a resource only if the kernel can establish that granting the request cannot lead to a deadlock either immediately or in future. It is easy to check if a deadlock is created by granting a request—deadlock analysis described in Section 11.4 can be used for this purpose. However, how would the system know whether a deadlock can arise in future?

Deadlock avoidance uses the worst-case analysis technique to check for future deadlocks. Each process is required to provide information concerning its maximum need for resources. In a simplistic approach to deadlock avoidance, an OS can use this information while admitting a process—it would admit a process only if the sum of the maximum resource needs of all processes in the system does not exceed the number of resources in the system. However, this approach is not practical because it leads to considerable idling of resources. It is more practical to permit the sum of maximum need of the processes to exceed the number of resources in the system and

When process P_i completes, it would release all resources allocated to it. We simulate this by updating $total_alloc_k$.

Similarly, a process P_g that is blocked on a request for $requested_resources_{g,k}$ units of resource class R_k , can complete if

$$total_exist_k - total_alloc_k \geq max_need_{g,k} - allocated_resources_{g,k}$$

Thus, no distinction needs to be made between *running* and *blocked* processes while determining the safety of the projected allocation state. Example 11.12 illustrates how safety of a projected allocation state is determined.

Example 11.12 A system contains 10 units of resource class R_k . The resource requirements of three processes P_1 , P_2 and P_3 are as follows:

	P_1	P_2	P_3
Maximum requirements	8	7	5
Current allocation	3	1	3

Figure 11.8 depicts current allocation state of the system. Process P_1 now makes a request for one resource unit. In the projected allocation state $total_alloc = 8$, hence it is a feasible allocation state and two unallocated units of resource class R_k are available in the system.

P_1	P_1	P_1	Total
8	3	1	7
7	1	0	7
5	3	0	10
Max need	Allocated resources	Requested resources	

Fig. 11.8 Allocation state in Banker algorithm for a single resource class

The safety of the projected state is determined as follows: Condition (11.3) is not satisfied by processes P_1 and P_2 since their balance requirements exceed the number of unallocated units in the system. However, P_3 satisfies condition (11.3) since it is exactly two units short of its maximum requirements, so the two available resource units can be allocated to P_3 if it requests them in future, and it can complete. This will release all resources allocated to it hence five resource units would become available for allocation. Now P_1 can complete since the number of unallocated units exceeds the units needed to satisfy its balance requirement. Now all resource units in the system are available to P_2 hence it, too, can complete. Thus, the processes can complete in the sequence P_3, P_1, P_2 . This makes the projected allocation state safe. Hence the algorithm will grant the request by P_1 .

The new allocation state is now (4,1,3) and $total_alloc_k = 8$. Now consider the following requests:

1. P_1 makes a request for 2 resource units.
2. P_2 makes a request for 2 resource units.
3. P_3 makes a request for 2 resource units.

```

1. Active := Running ∪ Blocked;
   for k = 1..r
      New_request[k] := Requested_resources[requesting_process, k];
2. Simulated_allocation := Allocated_resources;
   for k = 1..r      /* Compute projected allocation state */
      Simulated_allocation[requesting_process, k] :=
         Simulated_allocation[requesting_process, k] + New_request[k];
      Simulated_total_alloc[k] := Total_alloc[k] + New_request[k];
3. feasible := true;
   for k = 1..r      /* Check whether projected allocation state is feasible */
      if Total_exist[k] < Simulated_total_alloc[k] then feasible := false;
4. if feasible = true
   then      /* Check whether projected allocation state is safe */
      while set Active contains a process  $P_l$  such that
         For all k, Total_exist[k] - Simulated_total_alloc[k]
            ≥ Max_need[l, k] - Simulated_allocation[l, k]
         Delete  $P_l$  from Active;
         for k = 1..r
            Simulated_total_alloc[k] :=
               Simulated_total_alloc[k] - Simulated_allocation[l, k];
5. if set Active is empty
   then      /* Projected allocation state is safe */
      for k = 1..r      /* Delete the request from pending requests */
         Requested_resources[requesting_process, k] := 0;
      for k = 1..r      /* Grant the request */
         Allocated_resources[requesting_process, k] :=
            Allocated_resources[requesting_process, k] + New_request[k];
         Total_alloc[k] := Total_alloc[k] + New_request[k];

```

Example 11.13 Figure 11.9 illustrates operation of Banker's algorithm in a system containing four processes $P_1 \dots P_4$ and 6, 4, 9 and 5 resource units of four resource classes. The allocation state of the system is $(5, 3, 6, 3)$. Process P_2 has made a request $(0, 1, 1, 0)$, which is about to be processed. The algorithm simulates grant of this request in Step 2, and checks safety of the new state in Step 4. Figure 11.9(b) shows data structures of Banker's algorithm at the start of this check. In this state, only process P_1 can complete, so the algorithm simulates its completion. Figure 11.9(c) shows data structures after P_1 has completed. The remaining processes can complete in the order P_4, P_2, P_3 . Hence the request made by process P_2 is safe.

11.7 FORMAL CHARACTERIZATION OF RESOURCE DEADLOCKS

From Section 11.2.1, a circular wait-for relationship between processes is a necessary condition for a deadlock. This is manifest in a cycle in an RRAG or WFG. However, does the presence of a cycle imply a deadlock? In other words, is the presence of a cycle sufficient to conclude that a deadlock exists? The answer to this question depends on the nature of resource classes and resource requests in the system. Hence we begin by discussing models for resource classes and resource requests.

Resource class models A resource class R_i may contain a single instance of the resource, or it may contain many instances of the resource. We refer to these as *single instance* (SI) resource classes and *multiple instance* (MI) resource classes, respectively.

Resource request models We define two kinds of resource requests. In a *single request* (SR), a process is permitted to request one unit of only one resource class. In a *multiple request* (MR), a process is permitted to request one unit each of several resource classes. The kernel never partially allocates a multiple request, i.e., it either allocates all resources requested in an MR or does not allocate any of them. In the latter case, the process making the request is blocked until all resources can be allocated to it. Example 11.14 illustrates how multiple requests are handled in practice.

Example 11.14 In Figure 11.1(a), resource classes R_1 and R_2 are SI and MI type resource classes, respectively. In Example 11.1, all four requests made by the processes are single requests. Processes P_i and P_j could have made multiple requests involving the tape and the printer. In that case, a deadlock would not arise as a process would be allocated both printer and tape at the same time—in fact, as seen in Section 11.5.1, asking processes to make only one multiple request in their lifetime forms the basis of one approach to deadlock prevention.

Using the resource class and resource request models, we can define four kinds of systems as shown in Figure 11.10. We first discuss necessary and sufficient conditions for deadlocks in these systems. Later we present a general characterization of deadlocks that is applicable to all systems. It is possible to use WFGs to depict the resource state of SISR and SIMR systems. However, for the sake of generality, in all cases we use RRAG's to depict the resource state of a system.

		Resource request models	
		Single request (SR) model	Multiple request (MR) model
Resource instance models	Multiple instance (MI) model	Multiple instance Single request (MISR)	Multiple instance Multiple request (MIMR)
	Single instance (SI) model	Single instance Single request (SISR)	Single instance Multiple request (SIMR)

Fig. 11.10 Classification of systems according to resource class and resource request models

11.7.1 Single-instance-single-request (SISR) Systems

In an SISR system, each resource class contains a single instance of the resource and each request is a single request. As discussed in Section 11.2.2, existence of a cycle

condition (11.2). Thus one can conclude that the presence of a knot in an RRAG is a necessary and sufficient condition for the existence of a deadlock in an MISR system.

Example 11.15 The RRAG of Figure 11.3 (see Example 11.4) does not contain a knot since the path $P_i \dots P_k$ exists in it but a path $P_k \dots P_i$ does not exist in it. Now consider the situation if the following request is made:

6. P_k requests a printer.

Process P_k now blocks on the sixth request. The resulting RRAG is shown in Figure 11.11. The complete RRAG is a knot because Part 1 of Def. 11.5 is trivially

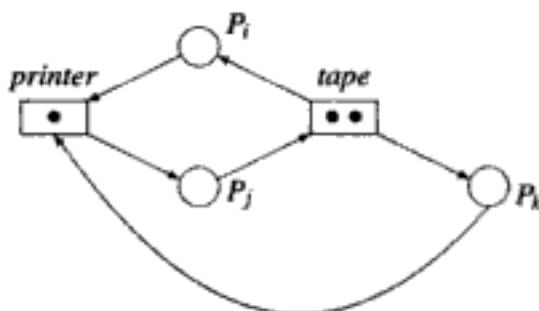


Fig. 11.11 A knot in RRAG of an MISR system implies a deadlock

satisfied, and every out-edge of every node is involved in a cycle, which satisfies Part 2 of Def. 11.5. It is easy to verify that processes $\{P_i, P_j, P_k\}$ are in a deadlock since $Blocked.P = \{P_i, P_j, P_k\}$, $WF_i = \{P_j\}$, $WF_j = \{P_i, P_k\}$ and $WF_k = \{P_j\}$ satisfies conditions (11.1) and (11.2).

11.7.3 Single-instance-multiple-request (SIMR) Systems

Each resource class in the SIMR system contains only one resource unit, hence it has exactly one out-edge in an RRAG. A process may make a multiple request, hence it may have more than one out-edge. Such a process remains blocked until each resource requested by it is available. A cycle implies that one of the resources requested by it is unavailable. Hence a cycle is a necessary and sufficient condition for deadlocks in an SIMR system.

Consider a process node P_i that has an out-edge (P_i, R_1) that is a part of a cycle, and an out-edge (P_i, R_3) that is not a part of any cycle (see Figure 11.12). Process P_i remains blocked until resource units of both R_1 and R_3 can be allocated to it. Since its out-edge (P_i, R_1) is involved in a cycle, and R_1 has only one out-edge, P_i faces an indefinite wait. P_j also faces an indefinite wait. Hence $\{P_i, P_j\}$ are involved in a deadlock.

11.7.4 Multiple-instance-multiple-request (MIMR) Systems

In the MIMR model, both process and resource nodes of an RRAG can have multiple out-edges. If none of the resource nodes involved in a cycle in the RRAG has multiple out-edges, the cycle is similar to a cycle in the RRAG of an SIMR or SISR system.

2. If a path $n_i \rightarrow \dots \rightarrow n_j$ exists in G then a path $n_j \rightarrow \dots \rightarrow n_i$ also exists in G .

A resource knot differs from a knot in that Part 1 of Definition 11.6 applies only to resource nodes. Thus, some out-edges of process nodes may not belong to the resource knot, whereas resource nodes cannot possess such out-edges.

Example 11.17 Nodes P_i, P_j, P_k, R_1, R_2 and R_3 of Figure 11.13 are involved in a resource knot if the allocation edge of resource class R_3 is (R_3, P_i) . Note that process P_i has another out-edge that is not included in the resource knot.

Clearly, a resource knot is a necessary and sufficient condition for the existence of a deadlock in an MIMR system. In fact, we state here without proof that a resource knot is a necessary and sufficient condition for deadlock in all resource systems discussed in this section (see Problem 22 in Exercise 11).

11.7.5 Processes in Deadlock

D , the set of processes in deadlock, contains processes represented by process nodes in resource knots. It also contains some other processes that face indefinite waits. We use the following notation to identify all processes in D .

- RR_i : The set of resource classes requested by process P_i .
- HS_k : The *holder set* of resource class R_k , i.e., set of processes to which units of resource class R_k are allocated.
- KS : The set of process nodes in resource knot(s) (we call it the *knot-set* of RRAG).
- AS : An *auxiliary set* of process nodes in RRAG that face indefinite waits. These nodes are not included in a resource knot.

KS is the set of process nodes included in resource knots. Now a process $P_i \notin KS$ faces an indefinite wait if all holders of some resource class R_k requested by it are included in KS . Resource classes whose holders are included in $\{P_i\} \cup KS$ similarly cause indefinite waits for their requesters. Therefore we can identify D , the set of deadlocked processes, as follows:

$$AS = \{P_i \mid RR_i \text{ contains } R_k \text{ such that } HS_k \subseteq (KS \cup AS)\} \quad (11.4)$$

$$D = KS \cup AS \quad (11.5)$$

Example 11.18 Figure 11.14 shows an RRAG of an MIMR system. The cycle $P_1 \rightarrow P_2 \rightarrow P_3 \rightarrow P_4 \rightarrow P_5 \rightarrow P_1$ in the RRAG forms a resource knot because none of R_1, R_2 or R_3 have an out-edge leaving the cycle. Hence a deadlock exists in the system. We identify the processes in D as follows:

$$KS = \{P_1, P_2, P_3\}$$

$$AS = \{P_4\} \text{ since } RR_4 = \{R_5\}, HS_5 = \{P_1\} \text{ and } \{P_1\} \subseteq \{P_1, P_2, P_3\}$$

$$D = KS \cup AS = \{P_1, P_2, P_3, P_4\}.$$

Process P_6 is not included in AS since $RR_6 = \{R_7\}$, $HS_7 = \{P_4, P_8\}$ and $HS_7 \not\subseteq (KS \cup AS)$.

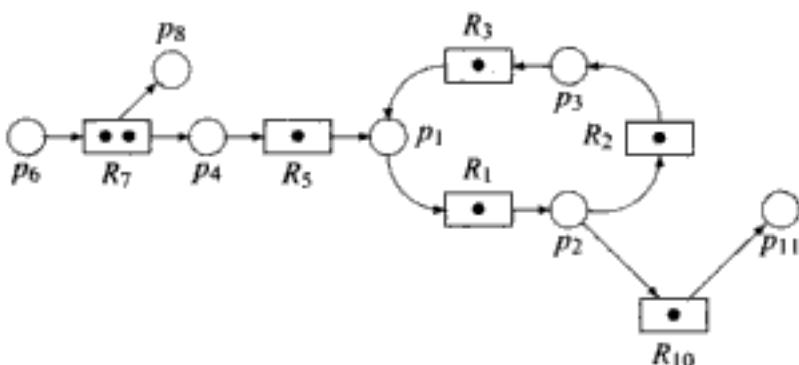


Fig. 11.14 Processes in deadlock

11.7.6 Deadlock Characterization Using Wait-For Graphs

Edges in a WFG indicate wait-for relationships between processes. Out-edges of a process P_i indicate the wait-for relationships of P_i with other processes in the system. These edges can be of two kinds depending on the nature of the request made by P_i , and the nature of resource class(es) whose units it has requested.

- **OR edges:** Edges (P_i, P_j) and (P_i, P_k) exist in the WFG if P_i requests a unit of resource class R_1 that contains two resource units allocated to processes P_j and P_k . P_i 's request can be satisfied when either P_j or P_k releases a unit of R_1 , hence these out-edges of P_i are called OR edges.
- **AND edge:** Edges (P_i, P_l) and (P_i, P_g) exist in the WFG if P_i makes a multiple request for units of single-instance resource classes R_1 and R_2 whose units are allocated to processes P_l and P_g , respectively. P_i requires resources held by both P_l and P_g , hence its out-edges are called AND edges.

We differentiate between OR and AND edges in a WFG by drawing an arc that connects the AND out-edges of a process. A process can have out-edges of both kinds at the same time; however, for simplicity, we assume that all out-edges of a process are either OR edges or AND edges. Example 11.19 illustrates OR and AND edges.

Example 11.19 Figure 11.15 shows two WFGs. The WFG in part (a) of Figure 11.15 corresponds to the RRAG in Figure 11.11, where two units of tapes are held by processes P_l and P_k , and process P_j has requested a tape unit. This situation gives rise to two out-edges for process P_j . These out-edges are OR edges. The WFG in part (b) of Figure 11.15 corresponds to the RRAG in Figure 11.12, where process P_i has made a multiple request for resources currently held by processes P_j and P_k . The out-edges of P_i are AND edges, hence we draw an arc connecting them.

Deadlock characterization for different resource systems is as follows:

- **SISR systems:** All processes in the WFG contain single out-edges. A cycle is a necessary and sufficient condition for deadlock.

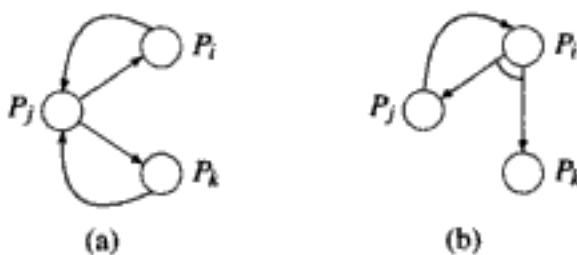


Fig. 11.15 WFGs with multiple out-edges

- *MISR systems:* All processes with multiple out-edges have OR edges. A cycle is a necessary but not a sufficient condition for a deadlock. A knot is a necessary and sufficient condition for deadlock.
- *SIMR systems:* All processes with multiple out-edges have AND edges. A cycle is a necessary and sufficient condition for a deadlock.

Deadlock characterization in the WFG of an MIMR system is not discussed here. It is left as an exercise (see Problem 23 in Exercise 11).

11.8 DEADLOCK HANDLING IN PRACTICE

Resources in an OS can be divided into hardware resources like memory and I/O devices and software resources like files containing programs or data, interprocess messages, and control blocks used by system processes or by the kernel itself. Operating systems tend to use simple deadlock handling policies that do not incur large CPU overhead. It is therefore no surprise that deadlock avoidance policies are rarely, if ever, used. An OS either uses deadlock prevention, or creates a situation in which explicit deadlock handling actions are unnecessary.

Memory Memory is a preemptible resource, hence its use by processes cannot cause a deadlock. Explicit deadlock handling is therefore unnecessary. The memory allocated to a process is freed by swapping out the process whenever the memory is needed for another process.

I/O devices The all-requests-together approach requires processes to make one multiple request for *all* their resource requirements. This policy incurs the least CPU overhead, however it has the drawback mentioned in Section 11.5.1—it leads to under-utilization of I/O devices that have to be allocated much before a process actually needs them. Resource ranking is not a feasible policy to control use of I/O devices because any assignment of resource ranks causes inconvenience to some section of users. This difficulty is compounded by the fact that I/O devices are generally non-preemptible. Avoidance incurs CPU overhead and also memory commitment for the data structures of avoidance algorithms like Banker's algorithm. Thus all approaches seem to have practical weaknesses.

Operating systems overcome this difficulty by creating virtual devices using the technique of *spooling*. For example, the system creates a virtual printer by using some disk area to store a file that is to be printed. Actual printing takes place when a printer becomes available. Since devices are created whenever needed, it is not necessary to pre-allocate them as in the all-requests-together approach unless the system faces a shortage of disk space. Management of I/O devices thus causes little CPU overhead.

• **Files and interprocess messages** A file is a user-created resource, hence a large number of files exist in an OS. Deadlock handling policies would cause unacceptably high CPU and memory overheads while controlling access to files. Hence general purpose OSs do not extend deadlock handling actions to files. Processes accessing a common set of files are expected to make their own arrangements to avoid deadlocks. OSs do not handle deadlocks caused by interprocess messages for similar reasons. Processes that exchange messages must establish their own conventions for deadlock handling.

Control blocks The kernel allocates control blocks like PCB and ECB to processes in a specific order—a PCB is allocated when a process is created and an ECB is allocated when the process gets blocked on an event. Hence resource ranking can be used here. If a simpler strategy is desired, all control blocks for a job or process can be allocated together at its initiation.

11.8.1 Deadlock handling in Unix

Most operating systems simply ignore the possibility of deadlocks involving user processes. This may be called the fourth approach to deadlock handling—the ostrich approach. Unix is no exception. However, it contains some features to address deadlocks involving processes that are executing the kernel code as a result of interrupts or system calls. The overall approach is that of deadlock prevention through resource ranking (see Section 11.5.2). Data structures in the kernel are locked and released in a standard order. However, there are exceptions to this rule because all kernel functionalities cannot lock the data structures in the standard order, so deadlocks are possible. We present simplified views of two arrangements used to avoid deadlocks.

The Unix kernel uses a buffer cache to speed-up accesses to frequently used disk blocks (see Section 12.10.2). The buffer cache consists of a pool of buffers in memory and a hashed data structure to check whether a specific disk block exists in a buffer. A list of buffers is maintained in LRU order to facilitate reuse of buffers. The normal order of accessing a disk block is to use the hashed data structure to locate a disk block, put a lock on the buffer containing the disk block, and then put a lock on the list of buffers to update the LRU status of the buffer. However, if a process merely wants to obtain a buffer for loading a new disk block, it directly accesses the list of buffers and takes off the first buffer that is not in use at the moment. To achieve

this, the process puts a lock on the list. Then it checks whether the lock on the first buffer in the list has been already set by some process. If not, it sets the lock and uses the buffer; otherwise, it repeats the process on the next buffer in the list. Deadlocks are possible because this order of locking the list and a buffer is different from the standard order of setting these locks.

Unix uses an innovative approach to avoid deadlocks. The process looking for a free buffer avoids getting blocked on its lock. It uses a special operation that tries to set a lock, but returns with a failure condition code if the lock is already set. If this happens, the process simply tries to set the lock on the next buffer, and so on until it finds a buffer that it can use. This approach avoids deadlocks by avoiding circular waits.

Another situation in which locks cannot be set in a standard order is in the file system function that establishes a link (see Section 7.4.1). A link command provides path names for a file and a directory that is to contain the link to the file. This command can be implemented by locking the directories containing the file and the link. However, a standard order cannot be defined for locking these directories. Consequently, two processes concurrently trying to lock the same directories may get deadlocked. To avoid such deadlocks, the file system function does not try to acquire both locks at the same time. It first locks one directory, updates it in the desired manner, and releases the lock. It then locks the other directory and updates it. Thus it requires only one lock at any time. This approach prevents deadlocks because the hold-and-wait condition is not satisfied by these processes.

EXERCISE 11

1. Compare and contrast the following policies of resource allocation:

- (a) All resource requests together
- (b) Allocation using resource ranking
- (c) Allocation using Banker's algorithm.

on the basis of (a) resource idling, and (b) overhead of the resource allocation algorithm.

2. When resource ranking is used as a deadlock prevention policy a process is permitted to request for a unit of resource class R_k only if $rank_k > rank_i$ for every resource class R_i whose resources are allocated to it. Explain whether deadlocks can arise if the condition is changed to $rank_k \geq rank_i$.
3. A system contains 6 units of a resource, and 3 processes that use this resource. Can deadlocks arise in the system if the maximum resource requirement of each process is 3 units?
If the system had 7 units of the resource, would the system be free of deadlocks for all time? Explain clearly.
4. A system containing preemptible resources uses the following resource allocation policy: When a resource requested by some process P_j is unavailable
 - (a) The resource is preempted from one of its holder processes P_i , if P_j is younger

than P_i . The resource is now allocated to P_i . It is allocated back to P_j when P_i completes. (A process is considered to be younger if it was initiated later.)

- (b) If the condition in 4(a) is not satisfied, P_i is blocked for the resource.

A released resource is always allocated to its oldest requester. Prove that deadlocks cannot arise in this system. Also prove that each process completes its execution in finite time.

5. Prove that the result of deadlock analysis, performed using Algorithm 11.1, does not depend on the order in which processes are transferred from *Running* to *Finished* (*Hint*: Consider Steps 1(a) and 1(b)).
6. Is the allocation state (6, 1, 2) for the system of Example 11.12 safe? Would the allocation state (5, 1, 1) be safe?
7. Would the following requests be granted in the current state?

	R_1	R_2		R_1	R_2		R_1	R_2
P_1	2	5		1	3		3	4
P_2	3	2		2	1		4	5
	Max need			Allocated resources			Total alloc	
							Total exist	

- (a) Process P_2 requests (1, 0)
 - (b) Process P_2 requests (0, 1)
 - (c) Process P_2 requests (1, 1)
 - (d) Process P_1 requests (1, 0)
 - (e) Process P_1 requests (0, 1).
 8. In the following system:
- | | R_1 | R_2 | R_3 | | R_1 | R_2 | R_3 | | R_1 | R_2 | R_3 |
|-------|----------|-------|-------|--|---------------------|-------|-------|-------------|-------------|-------|-------|
| P_1 | 3 | 6 | 8 | | 2 | 2 | 3 | | 5 | 4 | 10 |
| P_2 | 4 | 3 | 3 | | 2 | 0 | 3 | | 7 | 7 | 10 |
| P_3 | 3 | 4 | 4 | | 1 | 2 | 4 | | | | |
| | Max need | | | | Allocated resources | | | | Total alloc | | |
| | | | | | | | | Total exist | | | |
- (a) Is the current allocation state safe?
 - (b) Would the following requests be granted in the current state?
 - (i) Process P_1 requests (1, 1, 0)
 - (ii) Process P_3 requests (0, 1, 0)
 - (iii) Process P_2 requests (0, 1, 0).
 9. In a single resource system using Banker's algorithm, 2 unallocated resource units exist in the current allocation state. Three processes P_1 , P_2 and P_3 exist in the system. It is found that if P_1 or P_2 ask for allocation of a resource unit, the requests are rejected, whereas if P_3 asks for two resource units the request is granted. Explain why this may happen.
 10. A system using Banker's algorithm for resource allocation contains n_1 and n_2 resource units of resource classes R_1 and R_2 and three processes P_1 , P_2 and P_3 . The unallocated resources with the system are (1, 1). The following observations are made regarding the operation of the system:

bility of deadlocks in this system? If so, under what conditions? Suggest a solution to the deadlock problem.

18. A *phantom deadlock* is a situation wherein a deadlock handling algorithm declares a deadlock but a deadlock does not actually exist. Prove that phantom deadlocks are not detected by Algorithm 11.1 if processes are not permitted to withdraw their resource requests.
19. A road crosses a railway track at two points. Gates are constructed on the road at each crossing to stop road traffic when a train is about to pass. Train traffic is stopped if a car blocks the track. Two way traffic of cars is permitted on the road and two-way train traffic is permitted on the railway tracks.
 - (a) Discuss whether deadlocks can arise in the road-and-train traffic. Would there be no deadlocks if both road and train traffic is only one-way?
 - (b) Design a set of simple rules to avoid deadlocks in the road-and-train traffic.
20. It is proposed to use a deadlock prevention approach for the dining philosophers problem (see Section 9.5.3) as follows: Seats at the dinner table are numbered from 1 to n , and forks are also numbered from 1 to n , such that the left fork for seat i has the fork number i . Philosophers are required to obey the following rule: A philosopher must first pick up the lower numbered fork, then pick up the higher numbered fork. Prove that deadlocks cannot arise in this system.
21. A set of processes D is in deadlock. It is observed that
 - (a) If a process $P_j \in D$ is aborted, a set of processes $D' \subset D$ is still in deadlock.
 - (b) If a process $P_i \in D$ is aborted, no deadlock exists in the system.State possible reasons for this difference and explain with the help of an example. (*Hint:* Refer to Eqs. (11.4) and (11.5).)
22. Show that a resource knot in an RRAG is a necessary and sufficient condition for deadlocks in SISR, MISR, SIMR and MIMR systems.
23. Develop a deadlock characterization using a WFG for a resource system in which resource classes may contain multiple units and processes can make multiple resource requests.
24. A system uses a deadlock detection-and-resolution policy. The cost of aborting one process is considered to be one unit. Discuss how to minimize the cost of deadlock resolution in the SISR, SIMR, MISR and MIMR systems.

BIBLIOGRAPHY

Dijkstra (1965), Havender (1968) and Habermann (1969) are early works on deadlock handling. Dijkstra (1965) and Habermann (1969) discuss the Banker's algorithm. Coffman *et al* (1971) discuss the deadlock detection algorithm for a system containing multiple-instance resources. Holt (1972) provided a graph theoretic characterization for deadlocks. Islor and Marsland (1980) is a good survey paper on this topic. Zobel (1983) is an extensive bibliography. Howard (1973) discusses the practical deadlock handling approach described in Section 11.8. Tay and Loke (1995) and Levine (2003) discuss characterization of deadlocks.

Bach (1986) describes deadlock handling in Unix.

1. Bach, M. J. (1986): *The Design of the Unix Operating System*, Prentice-Hall, Engle-

Implementation of File Operations

File processing is implemented using modules of the file system and the Input Output Control System. The file system modules address issues concerning naming, sharing, protection and reliable storage of files. The IOCS modules address issues concerning efficient access to records in a file, and efficient use of I/O devices.

The IOCS layer consists of two sub-layers called *access methods* and the *physical IOCS* (PIOCS). The access methods layer addresses concerns related to efficient access to records in a file, and the physical IOCS layer addresses concerns related to device throughput.

The physical IOCS and an access method together implement operations on files. The physical IOCS performs input-output at the I/O device level and uses scheduling policies to enhance throughput of disks. The access method uses the techniques of buffering and blocking of records to reduce, and possibly eliminate, waiting times involved in accessing a record in a file.

We first discuss characteristics of I/O devices, and arrangements used to provide high reliability, fast access and high data transfer rates. We then discuss how I/O operations are performed at the level of I/O devices, the facilities offered by physical IOCS to simplify I/O operations, and how scheduling of I/O operations increases the throughput of a disk. Finally, we discuss how the techniques of buffering and blocking of records in a file reduce I/O wait times.

12.1 LAYERS OF THE INPUT OUTPUT CONTROL SYSTEM

The schematic of Figure 12.1 shows how the Input Output Control System (IOCS) implements file operations. Processes P_i and P_j are engaged in file processing activities and have already opened some files. When one of these processes makes a request to read or write data from a file, the file system passes on the request to the

IOCS. For a read operation, the IOCS checks whether the data required by the process is present in a memory area used as a *file buffer* or the *disk cache*. If so, the process is permitted to access the data immediately; otherwise, the IOCS issues one or more I/O operations to load the data into a file buffer or the disk cache, and the process has to wait until this I/O operation completes. The I/O operations are scheduled by a disk scheduling algorithm, which aims to provide high throughput of the disk. Thus the IOCS implements I/O operations in a manner that provides efficiency of file processing activities in processes and high throughput of I/O devices.

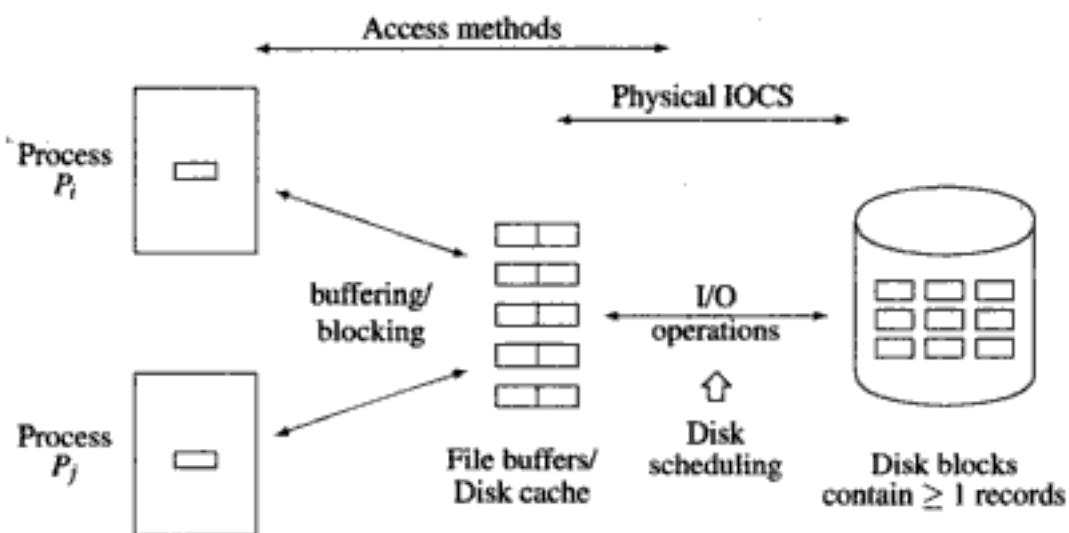


Fig. 12.1 Physical organization in access methods and physical IOCS

The IOCS is structured into two layers called the *access method* and the *physical IOCS*. The access method layer provides efficient file processing and the physical IOCS layer provides high device throughput. This structure of the IOCS separates process-level concerns in efficient implementation of an I/O operation from device-level concerns.

Figure 12.2 shows the hierarchy of file system and IOCS layers. The number of IOCS layers and their interfaces vary across different operating systems. In most operating systems, the physical IOCS is a part of the kernel. However, we will differentiate between the two in interest of clarity. We will assume that the physical IOCS is invoked through system calls, and it invokes other functionalities of the kernel also through system calls.

Table 12.1 summarizes significant mechanisms and policies implemented by IOCS layers in a conventional two-layer IOCS design. The physical IOCS layer implements device-level I/O. Its policy modules determine the order in which I/O operations should be performed to achieve high device throughput. These modules invoke physical IOCS mechanisms to perform I/O operations. The access method layer provides policy modules that ensure efficient file I/O and mechanisms that sup-

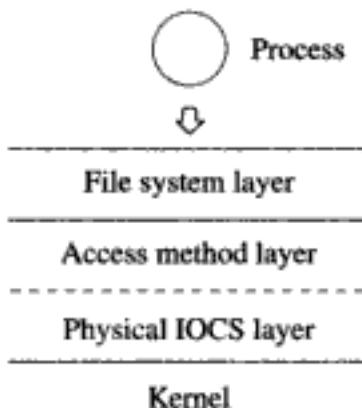


Fig. 12.2 Layers of the file system and IOCS

port file level I/O. These mechanisms use physical IOCS policy modules to achieve efficient device-level I/O. The file system layer implements sharing and protection of files using the policy and mechanism modules of the access method.

Table 12.1 Mechanisms and Policies in file system and IOCS layers

Physical IOCS	<ul style="list-style-type: none"> <i>Mechanisms:</i> I/O initiation, Providing I/O operation status, I/O completion processing, Error recovery <i>Policy:</i> Optimization of I/O device performance
Access methods	<ul style="list-style-type: none"> <i>Mechanisms:</i> File open and close, Read and write <i>Policy:</i> Optimization of file access performance
File System (FS)	<ul style="list-style-type: none"> <i>Mechanisms:</i> Allocation of disk blocks, Directory maintenance, Set/check file protection information <i>Policies:</i> Disk space allocation for access efficiency, Sharing and protection of files

Note that Table 12.1 lists only those mechanisms that can be meaningfully accessed from a higher layer. Other mechanisms, which are ‘private’ to a layer, are not listed here. For example, I/O buffering and blocking mechanisms exist in the access method layer. However, they are available only to access method policy modules; they are not accessed directly from the file system layer..

Physical organization in access methods and physical IOCS Figure 12.1 shows the physical organization. Read or write operations in processes result in calls on appropriate access methods. Access methods perform buffering and blocking to speed-up file processing. These actions result in commands to perform I/O operations to trans-

fer data between a disk block and a memory area used as a file buffer or a disk cache. Data is then copied from here into the address space of a process. The physical IOCS performs disk scheduling to determine the order in which I/O operations should be performed, and initiates I/O operations.

12.2 OVERVIEW OF I/O ORGANIZATION

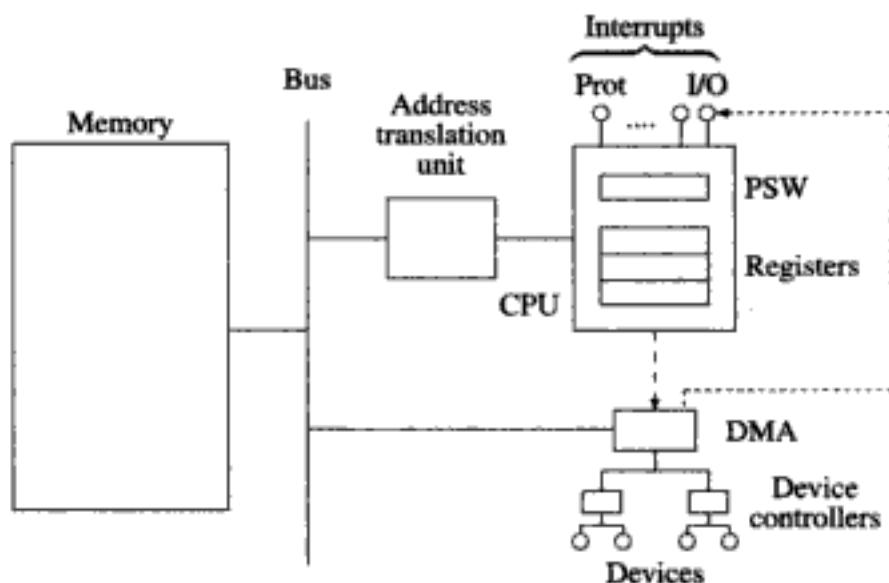


Fig. 12.3 Model of a computer system

Section 2.1 contained an overview of I/O organization. Figure 12.3 is reproduced from there. Each device controller is given a unique numeric id in the system. Similarly, each device connected to it has a unique numeric device id. A device address is a pair (*controller.id*, *device.id*).

An I/O operation consists of a sequence of elementary I/O tasks. It is initiated by executing an *I/O instruction*. The CPU, the DMA controller, the device controller and the I/O device participate to realize an I/O instruction. An I/O task is implemented by an *I/O command*, which requires participation of the DMA controller, the device controller and the I/O device, but not the CPU.

Example 12.1 The I/O operation to read a disk record with the id (*track.id*, *record.id*) involves the following I/O instructions and commands:

I/O instruction	:	Start I/O on (<i>controller.id</i> , <i>device.id</i>)
I/O commands	:	Position disk heads on track <i>track.id</i> Read record <i>record.id</i>

The I/O command ‘Position disk heads’ does not involve data transfer, but ‘read record’ involves data transfer.

Three modes of performing I/O operations—programmed mode, interrupt mode

and direct memory access (DMA) mode—were summarized in Table 2.1. We begin by describing operation of the DMA.

Direct memory access (DMA) In the DMA mode, data is transferred between an I/O device and the memory without involving the CPU. When an I/O instruction is executed, CPU transfers the following details to the DMA controller:

- Address of the I/O device
- Operation to be performed, viz. read or write
- Number of bytes of data to be transferred
- Address in memory and, if applicable, address on the I/O device, from where data transfer is to start.

This information is typically put into an area of memory. An I/O instruction uses the address of this area as an operand. When the instruction is executed, the CPU passes this address to the DMA interface. The CPU is not involved in the I/O operation beyond this point; it is free to execute instructions while the I/O operation is in progress.

The arrangement called *third party DMA* works as follows: Device controllers are connected to the DMA controller as shown in Figure 12.3. When an I/O instruction is executed, the DMA controller passes some details of the I/O operation to the device controller of the I/O device so that it can initiate the I/O operation. The device controller and the DMA controller co-ordinate transfer of data between the I/O device and the memory. At the end of data transfer, the DMA controller raises an I/O completion interrupt with the address of the device as the interrupt code. The interrupt handling routine analyzes the interrupt code to determine which device has finished its I/O operation, and takes appropriate actions.

Implementation of data transfer between an I/O device and the memory takes place as follows: The device controller sends a DMA-request signal when it is ready to perform a data transfer. On seeing this signal, the DMA controller obtains control of the bus, puts address of the memory location that is to participate in the data transfer on the bus and sends a DMA-acknowledgement signal to the device controller. The device controller now transfers the data to/from the memory.

The CPU executes instructions while an I/O operation is in progress, hence the CPU and the DMA controller are in competition for use of the bus. The technique of *cycle stealing* is used to ensure that both can use the bus without facing large delays. The CPU defers to the DMA controller for use of the bus at some specific points in its instruction cycle, typically when it is about to read an instruction or its data from memory. When the DMA wishes to transfer data to/from memory, it waits until the CPU reaches one of these points. It then steals a memory cycle from the CPU to implement its data transfer.

First party DMA achieves higher data transfer rates than third party DMA. In this arrangement, the device controller and the DMA controller are rolled into one unit.

It obtains control of the bus when it is ready for a data transfer. This technique is called *bus mastering*.

12.3 I/O DEVICES

I/O devices use a variety of principles like electro-mechanical principles of generating signals and electromagnetic or optical principles of recording data. They use a variety of I/O media, serve different purposes and use different methods of data organization and access. I/O devices can be classified according to the following criteria:

- *Purpose*: Input, print and storage devices
- *Nature of access*: Sequential and random access devices.
- *Data transfer mode*: Character and block mode devices.

A sequential access device uses its I/O medium in a sequential manner, so an operation is always performed on a data element that adjoins the data element accessed in the previous operation. Access to any other data element requires additional commands to skip over intervening data elements. A random access device can perform a read or write operation on data located in any part of the I/O medium.

The data transfer mode of a device depends on its speed of data transfer. A slow I/O device operates in the *character mode*, i.e., it transfers one character at a time between memory and the device. The device contains a small buffer to store the character. The device controller raises an interrupt after an input device reads a character or an output device writes a character. Device controllers of such devices can be connected directly to the bus. Keyboard, mouse and printer are character mode devices.

A device capable of a high data transfer rate operates in the *block mode* of data transfer. It is connected to a DMA controller. Tapes and disks are block mode devices. A block mode device needs to read or write data at a specific speed. Data would be lost during a read operation if the bus is unable to accept data from an I/O device at the required rate for transfer to memory, and a write operation would fail if it is unable to deliver data to the I/O device at the required rate. Such loss of data could arise if a data transfer is delayed due to contention for the bus.

To prevent loss of data during an I/O operation, data is not transferred over the bus during the operation. During an input operation, the data delivered by the I/O device is stored in a buffer in the DMA controller. It is transferred to the memory after the I/O operation completes. To perform an output operation, data to be written onto the I/O device is first transferred from the memory to the DMA buffer. During the I/O operation, it is transferred from the DMA buffer to the I/O device.

Table 12.2 lists I/O devices classified according to purpose and access mode. A unit of I/O medium is called an *I/O volume*. Thus a tape cartridge and a disk can be called a tape volume and disk volume respectively. I/O volumes in some I/O devices are detachable, e.g., floppy disks, compact disks (CDs), or digital audiotape (DAT)

Table 12.2 I/O Devices

Purpose	Access mode	Examples
Input	Sequential	Keyboard, Mouse, Network connections, Tapes, Disks
	Random	Disks
Print	Sequential	Printers, Network connections, Tapes, Disks
	Random	Disks
Storage	Sequential	Tapes, Disks
	Random	Disks

cartridges, while those in other I/O devices like hard disks are permanently fixed in the device. The information written (or read) in one I/O operation is said to form a *record*.

We use the following notation while discussing I/O operations.

- t_{io} : *I/O time*, i.e., time interval between the execution of an *io-init* instruction and the completion of I/O.
- t_a : *access time*, i.e., time interval between the issue of a read or write command and the start of data transfer.
- t_x : *transfer time*, i.e., time taken to transfer the data to/from an I/O device during a read or write operation

Figure 12.4 illustrates the factors influencing t_{io} . The I/O time for a record is the sum of its access time and transfer time, i.e.,

$$t_{io} = t_a + t_x \quad (12.1)$$

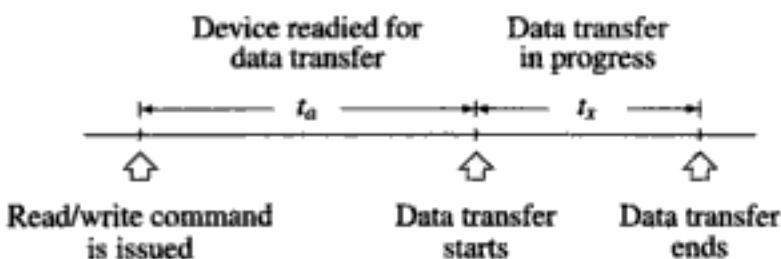


Fig. 12.4 Access and transfer times in an I/O operation

Actions of a sequential device are relative to its current position. It can operate either in the forward or in the backward direction with respect to this position. Due to this characteristic, a sequential device can only read records that are located on either side of its current position, and it can only write the next record in its forward direction. A random access device can read or write *any* record in an I/O volume. To implement this capability, all records on a volume are assigned addresses or ids.

While performing a read or write operation, the device must locate the desired record before commencing a read or write operation. Due to this peculiarity, the access time in a sequential device (see Eq. (12.1)) is a constant whereas it varies for different records in a random access device.

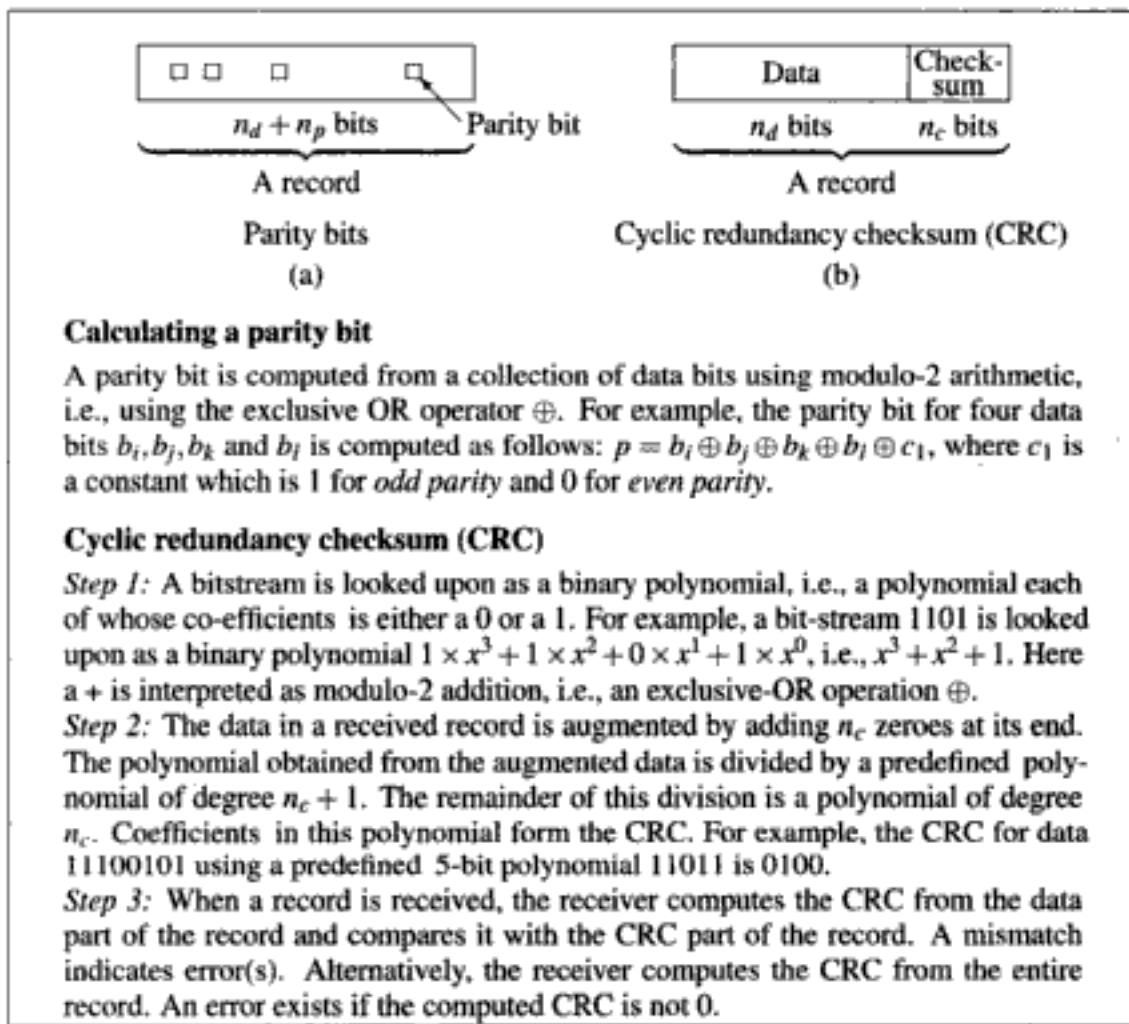
Error detection Data recording errors might occur while data is recorded on, or read from, an I/O medium. Similarly, data transmission errors might occur when data is transferred between an I/O medium and memory. To facilitate error detection, data being recorded or transmitted is viewed as a *bitstream*, i.e., as a stream of 1s and 0s, and special codes are used to represent the bitstream while recording or transmitting the bitstream.

Error detection is performed using redundancy. Some error detection information is associated with data and is recorded with it. This information is derived from the data using a standard technique. When data is read off an I/O medium, this information is also read off the medium. Now, error detection information is generated again from the read data using the same technique and it is compared with the error detection information read off the medium. A mismatch indicates some recording errors.

Figure 12.5 describes two approaches to error detection and correction. In the *parity bits* approach, n_p parity bits are computed for n_d bits of data. The parity bits are put in prefixed locations in a record. They are indistinguishable from data, except to the error detection/correction algorithm. In the *cyclic redundancy checksum* (CRC) approach, an n_c bit checksum is recorded at the end of the data. n_p depends on n_d , while n_c is independent of n_d .

Both approaches use modulo-2 arithmetic. This arithmetic ignores carries or borrows generated in any bit position, which makes it faster than binary arithmetic. A modulo-2 addition is represented as an exclusive-OR operation \oplus . It uses the following rules: $0 \oplus 0 = 0$, $1 \oplus 0 = 1$, $0 \oplus 1 = 1$, and $1 \oplus 1 = 0$. A popular variant of the parity bits approach used in RAMs and older magnetic tapes associates a single parity bit with a byte of data. As described in Figure 12.5, the parity bit is generated from all bits of a byte using the \oplus operation. It can detect a single error in a byte, but fails if two errors occur. It also cannot correct any errors. The error detection overhead is one parity bit for eight bits of data, i.e., 12.5 percent.

A *cyclic redundancy checksum* (CRC) is computed from data that is to be transmitted or recorded, and it is put into the CRC field of a record. It can indicate whether one or more errors have occurred in *any* byte of data, or whether bytes have been swapped or reordered. As described earlier, when a record is read a CRC is computed from its data field and compared with the code in its CRC field. An error exists if the two do not match. A practical value of n_c is 16 or 32 bits irrespective of the value of n_d . With $n_c < n_d$, error-detection is not foolproof because two bitstreams, say s_1 and s_2 , could generate the same CRC. If one of them is transformed into the other due to an error, CRC cannot detect this error. The probability of this



Calculating a parity bit

A parity bit is computed from a collection of data bits using modulo-2 arithmetic, i.e., using the exclusive OR operator \oplus . For example, the parity bit for four data bits b_i, b_j, b_k and b_l is computed as follows: $p = b_i \oplus b_j \oplus b_k \oplus b_l \oplus c_1$, where c_1 is a constant which is 1 for *odd parity* and 0 for *even parity*.

Cyclic redundancy checksum (CRC)

Step 1: A bitstream is looked upon as a binary polynomial, i.e., a polynomial each of whose co-efficients is either a 0 or a 1. For example, a bit-stream 1101 is looked upon as a binary polynomial $1 \times x^3 + 1 \times x^2 + 0 \times x^1 + 1 \times x^0$, i.e., $x^3 + x^2 + 1$. Here a + is interpreted as modulo-2 addition, i.e., an exclusive-OR operation \oplus .

Step 2: The data in a received record is augmented by adding n_c zeroes at its end. The polynomial obtained from the augmented data is divided by a predefined polynomial of degree $n_c + 1$. The remainder of this division is a polynomial of degree n_c . Coefficients in this polynomial form the CRC. For example, the CRC for data 11100101 using a predefined 5-bit polynomial 11011 is 0100.

Step 3: When a record is received, the receiver computes the CRC from the data part of the record and compares it with the CRC part of the record. A mismatch indicates error(s). Alternatively, the receiver computes the CRC from the entire record. An error exists if the computed CRC is not 0.

Fig. 12.5 Error detection approaches: (a) Parity bits, (b) Checksum

happening is $\frac{1}{2^n}$, so reliability of CRC is $1 - \frac{1}{2^n}$. For a 16-bit CRC, the reliability is 99.9985 percent. For a 32-bit CRC, reliability is 99.9999 percent.

12.3.1 Magnetic Tapes and Cartridges

The I/O medium in a tape or cartridge is a strip of magnetic material on which information is recorded in the form of 1s and 0s using principles of electromagnetic recording. The recording on a tape is multi-track. A read/write head is positioned on each track. It records or reads information from the bits of a byte and also some additional information used for error detection and correction.

Tape drives are sequential access devices. The operations that can be performed on these devices are: `read/write (count, addr (I/O area))`, `skip` and `rewind`, where `count` is the number of data bytes to be transferred and `addr (I/O area)` is the address of a memory area to which data transfer is to take place from the I/O medium (or vice versa). Tapes and DAT cartridges are popularly used for archival purposes. In

A start-of-track position is marked on each track, and records of a track are given serial numbers with respect to this mark. The disk can access any record using a record address of the form (*track_no, record_no*). The access time for a disk record is given by

$$t_a = t_s + t_r \quad (12.2)$$

where t_s : *seek time*, i.e., time to position the head on the required track

t_r : *rotational latency*, i.e., time to access desired record on the track

The seek time is caused by the mechanical motion of the head. Rotational latency arises because an I/O operation can start only when the required record is about to pass under the head. The average rotational latency is the time taken for half a disk revolution. Representative values of the average rotational latency are 3–4 msec, seek times are in the range of 5–15 msec, and data transfer rates are of the order of 10 Mbytes per second.

Variations in disk organization have been motivated by the desire to reduce the access time, increase the capacity of a disk and to make optimum use of disk surface. To provide fast access, a head can be provided for every track on the disk surface. Such disks, known as Head Per Track (HPT) disks, were used as paging devices in early virtual memory systems.

Higher disk capacities are obtained by mounting many platters on the same spindle to form a *disk pack*. One read/write head is provided for each circular surface of a platter. All heads in the disk pack are mounted on a single disk arm, hence at any moment the heads are located on identically positioned tracks of different surfaces. This property of a disk pack is exploited using the notion of a *cylinder*.

A *cylinder* is a collection of identically positioned tracks of different surfaces (see Figure 12.6). Hence I/O operations on records situated in the same cylinder can be performed without incurring seek times. The disk pack can be looked upon as consisting of a set of concentric cylinders between the innermost cylinder, which consists of the innermost tracks of all surfaces, and the outermost cylinder, which consists of the outermost tracks of all surfaces. A record's address is specified as (*cylinder number, surface number, record number*). The commands supported by a disk device are *read/write* (*record address, addr(I/O area)*) and *seek* (*cylinder number, surface number*).

Sectorized disks A sectorized disk organization is used to optimize use of the disk surface. A sector is a disk record with a standard size. The sector size is chosen to ensure minimum wastage of recording capacity due to inter-record gaps on the surface. Sectoring can be made a part of the disk hardware (hard sectoring), or could be implemented by the software (soft sectoring).

Data staggering techniques As described earlier, a DMA controller buffers the data involved in an I/O operation and transfers it to or from the memory as a single

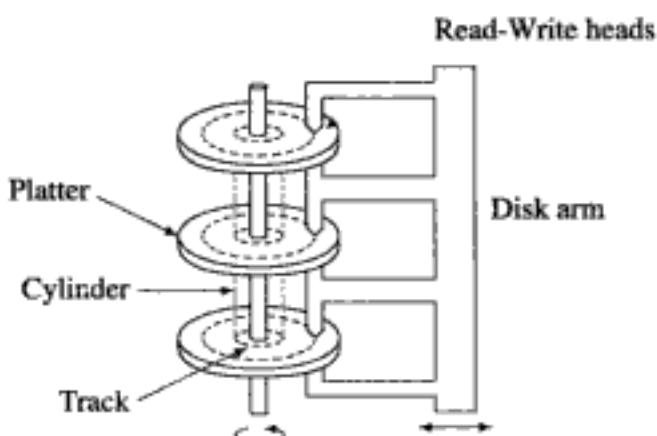


Fig. 12.6 Notion of a cylinder in a disk pack

block of data. For a read operation, the DMA controller performs the data transfer after the complete data is read off the device. While this transfer is being performed, the disk continues to revolve and one or more sectors may pass under the head by the time the transfer is completed. Hence an attempt to read the next sector cannot succeed in the same revolution of the disk. During a write operation data transfer takes place before recording of the data is initiated, however the effect is the same—data cannot be written into the next sector in the same revolution. In both cases, throughput of the disk device suffers.

A similar problem is caused when the first sector of a platter is to be read immediately after reading the last sector of the previous platter. Now, the read operation has to be switched between heads positioned on different platters; it causes a delay called the *head switching time*. In sequential file processing, this time has to elapse after reading the last sector on a platter, before the next read operation can be performed. By this time a few sectors of the next platter have passed under the read–write head. The seek time to move the head to the next cylinder also causes a similar problem. All these problems adversely affect the throughput of a disk.

The techniques of sector interleaving, head skewing and cylinder skewing address the problems caused by data transfer time, head switch time, and seek time, respectively. These techniques ensure that the read–write head will be able to perform the next operation before next consecutively numbered sector passes under it. This way, a read/write operation on the next sector can be performed in the current rotation of the disk. The technique of *sector interleaving* separates consecutively numbered sectors on a track by putting some other sectors between them. The expectation is that the data transfer involved in reading a sector can be completed before the next consecutively numbered sector passes under the head. *Head skewing* staggers the start of tracks on different platters of a cylinder so that the times when the last sector of a track and the first sector of the next track pass under their respective heads are separated by the head switch time. *Cylinder skewing* analogously staggers

the start of a track on consecutive cylinders to allow for the seek time.

Sector interleaving had a dramatic impact on the throughput of older disks. Sector interleaving is not needed in modern disks because modern disks have controllers that transfer data to and from memory at high rates; modern disks use only head and cylinder skewing. However, we discuss sector interleaving because it provides an insight into optimizing the peak disk throughput through data staggering.

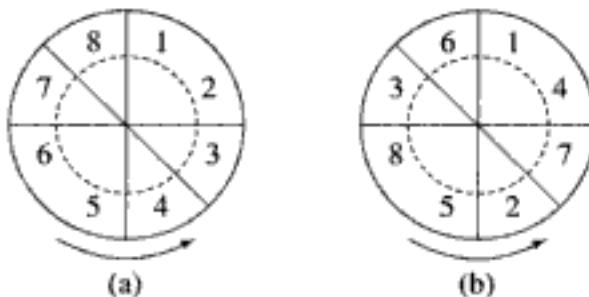


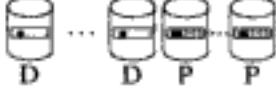
Fig. 12.7 Sector interleaving: (a) no interleaving, (b) interleaving factor = 2.

Figure 12.7 illustrates the sector interleaving technique. Interleaving factor (inf) is the number of sectors that separate consecutively numbered sectors on the same disk track. Part (b) of Figure 12.7 illustrates the arrangement when $inf = 2$, i.e., consecutively numbered sectors have two other sectors between them. This interleaving is exact, that is, each pair of consecutively numbered sectors is separated by two other sectors, because $n + 1$, where n is the number of sectors on a track, is a multiple of $inf + 1$. Note that interleaving with $inf = 1$ or 3 would not be exact since $inf + 1$ is a factor of n . In these cases some consecutive sectors would have to be separated by more than inf sectors, which leads to a performance penalty.

Disk throughput depends on the interleaving factor in an obvious way. Let t_{dt} be the time taken to transfer one sector's data between the DMA controller and the memory, and let t_{sect} be the time taken for one sector to pass under the disk head. Optimal performance is obtained if $t_{dt} = inf \times t_{sect}$, since I/O on the next sector can be started immediately after DMA finishes transferring the previous sector's data. $t_{dt} > inf \times t_{sect}$ implies that the next sector had passed under the head before the DMA finished data transfer for the previous sector, so the next sector can only be accessed in the next revolution of the disk. $t_{dt} < inf \times t_{sect}$ implies that the disk is idle for some time before it accesses the next sector in the same revolution. Disk throughput suffers in both these cases. Example 12.2 illustrates variation of peak disk throughput with the sector interleaving factor.

Example 12.2 A disk completes one rotation in 8 msec and has 8 sectors on a track, each containing 1K bytes of data. The values of t_{dt} and t_{sect} satisfy the relation $t_{sect} < t_{dt} < 2 \times t_{sect}$. To obtain the peak disk throughput for a value of inf , we read the sectors in the order 1 ... 8 over and over again and observe the number of bytes transferred in one second. Figure 12.8 shows variation of the peak disk throughput for different values of inf .

Table 12.3 RAID levels

Level	Technique	Description
Level 0	Disk striping 	Data is interleaved on several disks. During an I/O operation, the disks are accessed in parallel. Potentially, this organization can provide an n -fold increase in data transfer rates when n disks are used.
Level 1	Disk mirroring 	Identical data is recorded on two disks. While reading data, the copy that is accessible faster is used. One of the copies is accessible even after a failure occurs. Read operations can be performed in parallel if errors do not arise.
Level 2	Error correction codes 	Redundancy information is recorded to detect and correct errors. Each bit of data or redundancy information is stored on a different disk and is read or written in parallel. Provides high data transfer rates.
Level 3	Bit-interleaved parity 	Analogous to level 2, except that it uses a single parity disk for error correction. An error that occurs while reading data from a disk is detected by its device controller. The parity bit is used to recover lost data.
Level 4	Block-interleaved parity 	Writes a <i>block</i> of data, i.e., consecutive bytes of data, into a strip and computes a single parity strip for strips of a stripe. Provides high data transfer rates for large read operations. Small read operations have low data transfer rates; however, many such operations can be performed in parallel.
Level 5	Block-interleaved distributed parity 	Analogous to level 4, except that the parity information is distributed across all disk drives. Prevents the parity disk from becoming an I/O bottleneck as in level 4. Also provides better read performance than level 4.
Level 6	P + Q redundancy 	Analogous to RAID Level 5, except that it uses two independent distributed parity schemes. Supports recovery from failure of two disks.

(Note: D and P indicate disks that contain only data and only parity information, respectively. • marks indicate bits of a byte that are stored on different disks, and their parity bits. — indicates a strip containing parity information)

RAID 0 + 1, a single error in a copy of a stripe makes the entire copy inaccessible, so errors in both copies of a stripe would make the stripe inaccessible. In RAID 1 + 0, an error on one disk would be tolerated by accessing its mirror disk. A stripe would become inaccessible only if both a disk and its mirror disk have errors.

RAID level 2 This RAID organization uses *bit striping*, i.e., it stores each bit of data or redundancy information on a different disk. When data is to be written, the i^{th} data strip contains the i^{th} bit of each byte and a parity strip contains one of the parity bits computed from corresponding bits in all strips of the stripe. An *error correcting code* is used to compute and store redundancy information for each byte (see Section 12.3). Thus, eight disks are used to record the bits of a byte, and a few more disks are used to record redundancy information. For example, the (12, 8) Hamming code, which is adequate for recovery from a single failure, would require four redundancy bits. The RAID 2 arrangement employing this code would consist of 8 data disks and 4 disks containing redundancy information, each storing one bit of data or parity information. This RAID arrangement can read/write data eight times faster than a single disk. However, it is expensive because several disks are needed to store redundancy information.

RAID level 3 Level 3 employs disk striping with a *bit-interleaved parity* scheme, i.e., it employs *bit interleaving*—it writes the bits of a byte on different disks—and employs a single parity bit per byte. The data strips of a stripe are stored on 8 data disks and the parity strip is stored on the parity disk. Thus, RAID level 3 employs a significantly lesser amount of redundant information than RAID level 2. A read operation is performed as follows: The disk controller checks whether an error exists within a strip. If so, it ignores the entire strip and recovers the data in the strip using the parity strip—the value of a data bit is the modulo-2 difference between the parity bit and the modulo-2 sum of corresponding bits of other strips in the stripe.

All data disks participate in an I/O operation. This feature provides high data transfer rates. However, it also implies that only one I/O operation can be in progress at any time. Another drawback of RAID level 3 is that parity computation can be a significant drain of the CPU power. Hence parity computation is off-loaded to the RAID itself.

RAID level 4 Level 4 is analogous to level 3 except that it employs *block-interleaved parity*. Each strip accommodates a *block* of data, i.e., a few consecutive bytes of data. If an I/O operation involves a large amount of data, it will involve all data disks as in RAID level 3, hence RAID level 4 can provide high data transfer rates for large I/O operations. A read operation whose data fits into one block will involve only a single data disk, so small I/O operations have small data transfer rates; however, several such I/O operations can be performed in parallel.

A write operation involves computation of parity information based on data recorded in all strips of a stripe. This can be achieved by first reading data contained

in all strips of a stripe, replacing the data in some of the strips with new data that is to be written, computing the new parity information, and writing the new data and parity information on all disks. However, this procedure limits parallelism because all disks are involved in the write operation even when new data is to be written into a single block $block_i$ of stripe $stripe_i$. Hence, the priority information is computed by a simpler method that involves the exclusive OR of three items—the old information in the parity block, the old data in block $block_i$, and the new data to be written in block $block_i$. This way, only the disk(s) containing the block(s) to be written into and the parity block are involved in the write operation, and so several small read operations involving other disks can be performed in parallel with the write operation.

RAID level 5 Level 5 uses block level parity as in level 4, but distributes the parity information across all disks in the RAID. This technique permits small write operations that involve a single data block to be performed in parallel if their parity information is located on different disks. Small fault-free read operations can be performed in parallel as in RAID level 4. Hence this organization is particularly suitable for small I/O operations performed at a high rate. Larger operations cannot be performed in parallel; however, the organization provides high data transfer rates for such operations. It also provides higher peak disk throughput for read operations than level 4 because one more disk can participate in read operations.

RAID level 6 This organization uses two independent distributed parity schemes. These schemes support recovery from failure of two disks. Peak disk throughput is slightly higher than in level 5 because of the existence of one more disk.

12.4 DEVICE LEVEL I/O

We assume that I/O operations are performed in the DMA mode. As mentioned in Section 12.2, the DMA is a special purpose processor. It performs a direct memory access over the bus to implement a data transfer between the memory and an I/O device. The DMA implements one complete I/O operation, i.e., a sequence of I/O commands, and raises an I/O interrupt at the end of the I/O operation. We assume that a computer system contains the I/O-related features described in Table 12.4.

Example 12.3 The I/O operation of Example 12.1, viz. read record ($track_id$, $record_id$) on device ($controller_id$, $device_id$), is implemented by issuing the instruction

I/O-init (controller_id, device_id), addr

where *addr* is the start address of the memory area containing the two I/O commands

Position disk heads on track *track_id*

Read record *record_id*

Table 12.4 Computer system features supporting functions in device-level I/O

Function	Description of computer system feature supporting it
Initiating an I/O operation	Instruction <i>I/O-init</i> (cu, d), <i>command address</i> initiates an I/O operation. (cu, d) is the device address and <i>command address</i> is the address of the memory location where the first command of the I/O operation is stored. We assume that I/O commands describing an I/O operation are in consecutive memory locations. (Some systems fetch the I/O command address from a standard memory location when the <i>I/O-init</i> instruction is executed.) The <i>I/O-init</i> instruction sets a condition code to indicate whether the I/O operation has started.
Checking device status	Instruction <i>I/O-status</i> (cu, d) obtains status information for the I/O device with address (cu, d). The information indicates whether the device is busy, free or in an error state, and cause of the error, if any.
Performing I/O operations	I/O commands for device specific I/O operations implement operations like positioning of heads over a track, and reading of a record.
Handling interrupts	The interrupt hardware implements the interrupt action described in Section 2.1.1.

12.4.1 I/O programming

We use the term *I/O programming* to describe all actions concerning initiation and completion of an I/O operation. I/O initiation is performed using an *I/O-init* instruction. I/O completion actions are performed when an I/O interrupt occurs indicating completion of an I/O operation. To see the details of I/O programming, we consider an application program that uses a bare machine, i.e., a computer system that does not have any software layers between the application program and the machine's hardware. Such a program must itself perform all actions concerning initiation and completion of an I/O operation.

I/O initiation When an *I/O-init* instruction is executed, the CPU sends the device address to the DMA. The DMA interrogates the device to check its availability. This process is called device selection. The DMA informs the result of device selection actions to the CPU, which sets an appropriate condition code in its condition code register. The *I/O-init* instruction is now complete; the CPU is free to execute other instructions.

If device selection is successful, the DMA immediately starts the I/O operation by accessing and decoding the first I/O command. If device selection fails, the condition code set by the *I/O-init* instruction describes the cause of failure, e.g., if the device is busy with some other operation, or if a hardware fault has occurred. If the device was busy, the program can retry I/O initiation sometime in future. If

a hardware fault has occurred, the program can report the condition to the system administrator.

I/O completion processing Since the program executes on a bare machine, unlike in a multiprogramming or time sharing system (see Chapter 2), it cannot be blocked until the I/O operation completes. Hence the CPU remains available to the program even during the I/O operation and continues to execute instructions. However, it does not perform any work until the I/O operation completes!

The program addresses this problem by using a flag to indicate whether the I/O operation has completed. To start with, its value is set to 'in progress'. The interrupt processing routine changes the value of the flag to 'completed'. After starting the I/O operation, the CPU enters a loop where it repeatedly checks this flag. (This is a busy wait—see Section 9.2.2.) When the interrupt processing routine changes the flag to 'completed', the CPU exits from the loop and resumes execution of the program.

Example 12.4 describes details of I/O programming.

Example 12.4 Figure 12.9 illustrates the basic actions involved in I/O programming. IO_FLAG is set to '1' to indicate that the I/O operation is in progress, and the *I/O-init* instruction is executed. The conditional branch (BC) instructions test the condition code set by the *I/O-init* instruction. Condition code *cc₁* is set if I/O initiation is successful. In that event the I/O operation would have already started, hence the program goes on to execute the instruction with label PROCEED. The two instructions COMP (i.e., compare) and BC at PROCEED implement a busy wait; the CPU compares value of IO_FLAG with 1 and continues to loop if this is the case. Condition code *cc₂* indicates that the device is busy, hence the program retries the I/O instruction until I/O initiation succeeds. Condition code *cc₃* indicates an I/O error situation. The error is reported to the system administrator. These details are not shown in the program.

	SET	IO_FLAG, '1'	I/O in progress
RETRY:	<i>IO_init</i>	(cu, d), COMMANDS	
	BC	cc ₁ , PROCEED	Initiation successful?
	BC	cc ₂ , RETRY	Is the device busy?
	BC	cc ₃ , ERROR	Error, inform operator
PROCEED:	COMP	IO_FLAG, '1'	I/O still in progress?
	BC	EQ, PROCEED	
	...		
COMMANDS:	...	{I/O commands}	
	...		
IO_INTRPT:	SET	IO_FLAG, '0'	Interrupt processing
	...		

Fig. 12.9 Device level I/O

I/O completion processing involves the following actions: When an I/O interrupt occurs, control is transferred to the instruction with the label IO_INTRPT by the interrupt action (see Section 2.1.1). This is the start of the I/O interrupt processing routine. The I/O interrupt routine changes IO_FLAG to '0', and returns. This action brings the

program out of the busy wait at PROCEED.

12.4.2 The Physical IOCS (PIOCS)

Functioning of an OS component has three important aspects—offering a convenience of some kind, ensuring efficient use of resources under its control, and helping to ensure overall efficiency of the operating system. All these three aspects are present in the case of physical IOCS. The first aspect involves offering a convenient interface for performing device level I/O. The other two aspects of its functioning are somewhat less obvious. Table 12.5 contains an overview of these three aspects. The physical IOCS layer provides the following functionalities to implement these three aspects:

1. I/O initiation, completion and error recovery
2. Awaiting completion of an I/O operation
3. Optimization of I/O device performance.

Table 12.5 Aspects of Physical IOCS functions

Aspect	Description
Interface for device level I/O	Physical IOCS provides simple and easy-to-use means to perform I/O operations and handle I/O interrupts. Thus, a program does not require intricate knowledge of I/O initiation and interrupt processing in order to perform I/O.
Good system performance	Performing I/O at the bare machine level involves a busy wait following an I/O initiation, which lasts until an I/O-interrupt occurs (see Example 12.4). This wastes valuable CPU time. Physical IOCS avoids busy waits by interfacing with the process management component to put a program in the <i>blocked</i> state while it waits for completion of an I/O operation.
Good device performance	Good device performance is obtained by scheduling I/O operations aimed at a device in a suitable order. This activity is called Device I/O scheduling or simply <i>I/O scheduling</i> .

I/O initiation, completion and error recovery Physical IOCS support for I/O programming makes the DMA and device intricacies transparent to a process. The process needs to specify only the device address and details of the I/O operation. The I/O operation is performed asynchronously with operation of the process. If the process does not contain computations that can be performed concurrently with the I/O operation, it can await completion of the I/O operation using synchronization provided by PIOCS. When an interrupt pertaining to the device arises, physical IOCS analyses the interrupt to determine the status of the I/O operation. If the I/O operation has completed successfully, it notes this fact for use in implementing synchronization of a process with completion of an I/O operation. If the interrupt indicates that an

a functionality to perform the assignment of a physical device to a logical device. This functionality may be invoked through the OS command processor before starting the execution of a program. Alternatively, a process may invoke it dynamically by issuing a system call during its execution.

A process may assign a physical device (cu_i, d_j) of an appropriate device class to a logical device `mydisk`. It may now make a request *I/O-init mydisk, command address*. The physical IOCS would implement this command on the device (cu_i, d_j). Logical devices are also used to implement the notion of virtual devices described in Section 1.3.2.1. To create many virtual devices on a disk, the disk can be assigned to many logical devices.

12.4.4 Physical IOCS Data Structures

Physical IOCS uses the following data structures:

- Physical device table (PDT)
- Logical device table (LDT)
- I/O control block (IOCB)
- I/O queue (IOQ).

Figure 12.11 illustrates the relationship between these data structures. Physical device table (PDT) is a system-wide data structure that contains information concerning all physical devices in the system. The important fields of this table are *device address*, *device type* and *IOQ pointer*. The *IOQ pointer* field of a PDT entry contains a pointer to the queue of I/O requests for the device. Information in the queue is used for I/O scheduling.

Logical device table (LDT) is a per-process data structure describing assignments to the logical devices used by the process. One copy of LDT exists for every process in the system. This copy is accessible from the PCB of the process. The LDT contains one entry for each logical device. Fields *logical device name*, and *physical device address* in the entry contain information concerning the current assignment, if any, for the logical device. Note that for certain classes of devices (e.g., disks), the mapping from logical devices to physical devices can be many-to-one. Thus, many logical devices, possibly belonging to different processes, can share the same physical device.

The physical IOCS checks for assignment conflicts while making new assignments. For example, only one assignment can be in force for a non-sharable device like a printer. The device type field of the PDT is used to determine whether sharing is feasible for a device. A device assignment may be nullified by a process through a physical IOCS call. In addition, the physical IOCS nullifies all device assignments of a process at the end of its execution.

An I/O control block (IOCB) contains all information pertaining to an I/O operation. The important fields in an IOCB are *logical device name*, *I/O details* and *status*

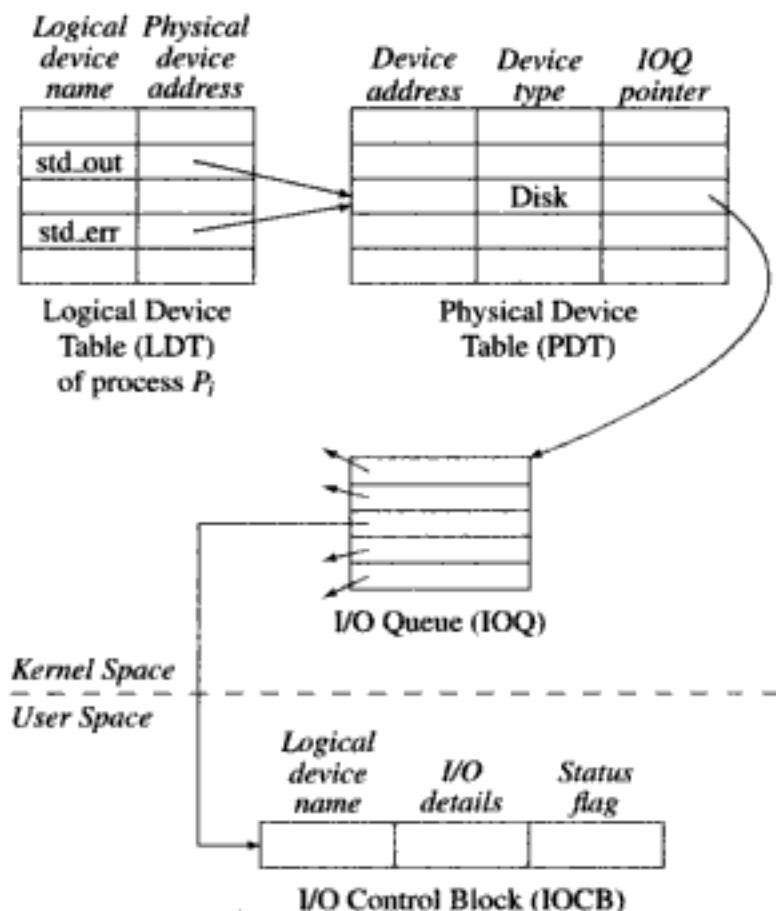


Fig. 12.11 Physical IOCS data structures

flag. The I/O details field contains start address of the memory area that holds all I/O commands involved in the I/O operation. The status flag indicates whether an I/O operation is ‘in progress’ or ‘completed’.

The I/O queue (IOQ) is a list of all I/O operations pending on a physical device. Each entry of IOQ contains a pointer to an I/O control block. Information in the IOQ is used for optimizing the performance of the I/O device. Note that the PDT entry of a physical device, rather than the LDT entry of a logical device, points to the IOQ. This is significant when a physical device is assigned to many logical devices. Information concerning I/O operations directed at all logical devices to which it is assigned are now available in a single place for I/O scheduling.

The physical and logical device tables are accessed with device numbers as keys. PDT is a fixed table; it can be formed at system boot time by obtaining details of all devices connected to the system. LDT is generally a fixed size table whose size is specified at system generation time or boot time. These tables are organized as linear tables. An I/O control block is allocated on demand when an I/O operation is to be initiated and is destroyed when the I/O operation completes, so it is best to

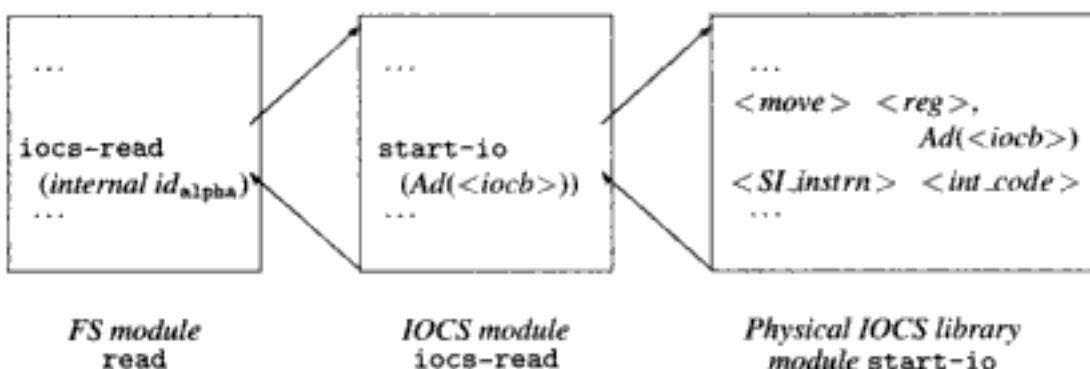


Fig. 12.13 Invocation of physical IOCS library module `start-io`

To enter the I/O control block address in the correct IOQ, physical IOCS extracts the logical device id from the IOCB, and obtains the address of the physical device assigned to the logical device from the logical device table of the process. It then obtains the address of IOQ for the physical device from the physical device table and adds the IOCB address at the end of IOQ. The I/O operation can be initiated straightforwardly if no other entries exist in the IOQ. If other entries exist, presumably one of the previous I/O operations is in progress, so the new request simply gets entered in the IOQ. It would be initiated sometime in future (see description of I/O completion for details).

I/O initiation is performed as described in Section 12.4. The *status flag* field of the I/O control block is used in a manner analogous to the use of `IO_FLAG` in Example 12.4.

I/O completion processing The I/O completion processing function is implicitly invoked at the occurrence of an I/O completion interrupt. This function performs the following actions:

1. Invoke the I/O error recovery function if the I/O operation was unsuccessful.
2. Set the status flag of the IOCB to ‘completed’ and remove address of the IOCB from IOQ of the device.
3. If I/O operations are pending on the device, initiate one of them.
4. If the process issuing the I/O operation is blocked awaiting its completion, change its state to *ready*.

I/O completion processing is implemented as follows: The interrupt hardware provides two items of information—address of the physical device raising the I/O interrupt, and an I/O status code describing the cause of the interrupt. These items of information are available to physical IOCS when control is passed to it. It uses the device address to find the physical device table entry of the device. It now locates the I/O control block corresponding to the I/O operation that caused the interrupt. If

the I/O status code indicates an error condition, physical IOCS consults the *device type* field of the PDT entry and invokes an appropriate I/O error recovery routine with the IOCB address as a parameter. If I/O completion was normal, the physical IOCS changes the *status flag* field in the IOCB to 'completed' and removes the IOCB address from the IOQ. If IOQ is not empty, another I/O operation is now initiated.

Actions of the error recovery routine are device-specific. Transient I/O errors can arise for some devices like tapes and disks, hence the recovery routine retries the faulty operation a few times before declaring it to be irrecoverable. The process is notified of the failure if it has set appropriate parameters in the `read` or `write` call; otherwise, the process is aborted. If error recovery is successful, the error recovery routine returns control to the I/O completion processing function, which now performs all actions pertaining to a successful completion of I/O operation. This arrangement makes occurrence of transient I/O failures transparent to a process.

One vital aspect of I/O completion processing is the manner in which physical IOCS locates the I/O control block (IOCB) for the I/O operation that caused an interrupt. The address of the physical device causing an interrupt is available to the physical IOCS, so it should be possible to locate this address from the PDT entry of this I/O device. Either the I/O initiation function can put the IOCB address in some special field of the PDT entry when it initiates an I/O operation on the device, or it can move the IOCB to the start of IOQ while initiating the I/O operation. I/O completion processing simply picks up the IOCB address from the PDT entry or the IOQ of the device for posting completion of the I/O operation or for passing it to the error recovery procedure.

Awaiting completion of an I/O operation A process invokes this function through the physical IOCS library call `await-io (<IOCB_address>)` where the IOCB describes the awaited I/O operation. The physical IOCS merely tests the status flag in the IOCB, and returns to the process if the flag value is 'completed'. If not, the physical IOCS library routine requests the kernel to block it on the event 'successful I/O completion'. The kernel creates an ECB for the I/O completion event and enters it in the ECB list. This ECB contains the id of the process waiting for completion of the I/O operation. When the I/O completion event occurs, the event handling mechanism locates its ECB, extracts the id of the process, and marks an appropriate change in its state. This arrangement ensures that the library routine is activated at the completion of the I/O operation. (See Example 3.5 for an explanation of this arrangement.) Control now returns to the process.

Optimization of I/O device performance The physical IOCS can optimize the performance of an I/O device in two ways. It can perform smart scheduling of I/O operations directed at an I/O device to achieve higher throughput. Alternatively, it can reduce the number of I/O operations to be performed by caching some part of the data stored on a device. We discuss these techniques in Sections 12.5 and 12.9, respectively.

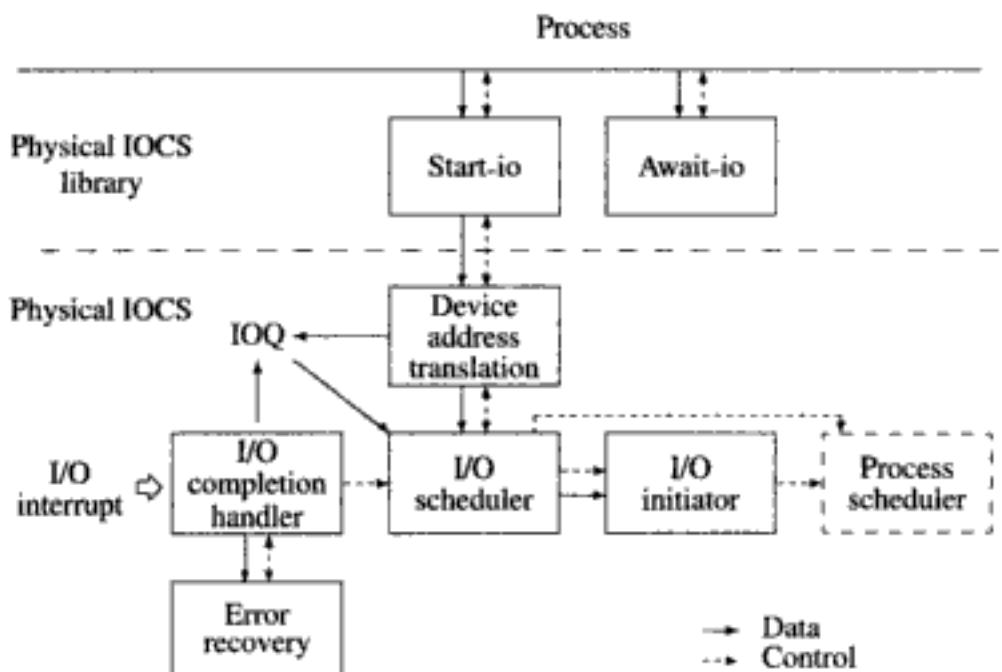


Fig. 12.14 Summary of physical IOCS functionalities

Summary of physical IOCS functionalities Figure 12.14 summarizes the physical IOCS functionalities. Each block represents a physical IOCS module. Modules above the dashed line execute in the user mode, while those below this line execute in the kernel mode. Module 'I/O scheduler' performs smart disk scheduling of I/O operations to obtain good device performance. The physical IOCS functionalities are activated in one of two ways: Calls on the physical IOCS library modules `start-io` or `await-io` by a process, or occurrence of an I/O completion interrupt. When a user process invokes `start-io`, `start-io` invokes the I/O initiation functionality of the physical IOCS. After device address translation, the request is entered in the IOQ of the physical device and control is passed to the I/O scheduler. I/O scheduler invokes the I/O initiator to start the I/O operation immediately if no other I/O operations exist in the IOQ of the device. Control is then passed to the process scheduler, which returns it to the process making the request.

When the `await-io` module of physical IOCS is invoked, it checks the status flag of the IOCB. If the I/O operation is complete, control is immediately returned to the process; otherwise, the process makes a kernel call to block itself. At an I/O completion interrupt, an error recovery routine is invoked if an I/O error has occurred, otherwise the status flag of the IOCB is set to 'completed'. The ECB-PCB arrangement is used to activate a process (if any) awaiting I/O completion, and the I/O scheduler is called to initiate the next I/O operation. Control is then handed over to the process scheduler.

12.4.6 Device Drivers

The design of the physical IOCS described so far has one drawback—it assumes that the physical IOCS has sufficient knowledge to handle I/O initiation, completion processing and error recovery for all kinds of I/O devices existing in the system. Addition of a new I/O device requires changes to the physical IOCS , which can be both complex and expensive. This feature restricts the classes of I/O devices that can be connected to the system, thus limiting the scope for growth and evolution of the system.

To overcome this problem, physical IOCS provides only generic support for I/O operations. It invokes a specialized *device driver* (DD) module for handling device level details for a specific class of devices. Thus device drivers are not a part of the physical IOCS . This arrangement enables new I/O devices to be added to the system without having to reconfigure the kernel. DDs are loaded by the system boot procedure depending on the classes of I/O devices that exist in the system. Alternatively, device drivers can be loaded whenever needed during operation of the OS.

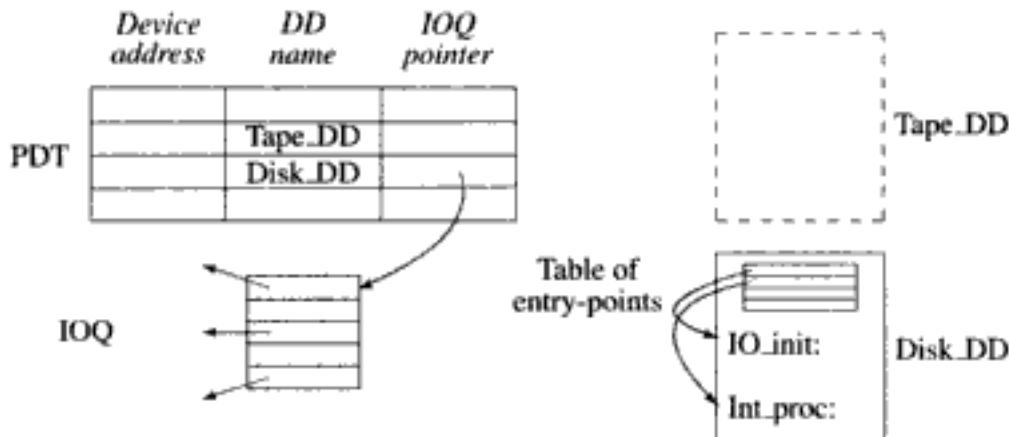


Fig. 12.15 Use of device drivers

Figure 12.15 illustrates the arrangement used to support dynamic loading of DDs. The PDT entry of a device shows the name of its device driver instead of the name of its device class. When I/O is to be invoked on a particular device, the name of its DD is obtained from its PDT entry. Disk.DD, the device driver for the system disk, has been loaded at system boot time. Other DDs, viz. Tape.DD, would be loaded on demand.

A device driver contains functionalities of the four modules shown in Figure 12.14, viz. I/O scheduler, I/O initiator, I/O completion handler and I/O error recovery, for one class of I/O devices. It contains entry-points for each of these functionalities, and provides a table of entry-points at the start of its code. The physical IOCS refers to this table to pick the address of the entry-point for specific a operation.

As described before, physical IOCS invocations can be classified into explicit

invocations and implicit invocations. When the physical IOCS is invoked explicitly for initiating an I/O operation, it performs the generic function of entering details of the I/O operation into the IOQ of the concerned device. It now consults the *DD name* field of the PDT entry for the device, obtains the identity of the device driver and obtains the address of the entry-point for I/O init by following the standard conventions. (If DD does not already exist in memory, it is loaded at this point.) It now passes control to this entry-point of the DD. The DD performs I/O scheduling followed by I/O initiation and returns control to the physical IOCS . The physical IOCS passes control to the process scheduler. When the physical IOCS is invoked implicitly at an I/O interrupt, it performs similar actions to identify the DD entry-point for handling interrupts and passes control to it. After processing the interrupt, DD returns control to the physical IOCS , which passes it to the process scheduler.

12.5 DISK SCHEDULING

The seek time of a disk block depends on its position relative to the current position of the disk heads. Consequently, the total seek time involved in performing a set of I/O operations depends on the order in which the operations are performed. The throughput of a disk defined as the number of I/O operations performed per second, also depends on the order in which I/O operations are performed. Hence the physical IOCS and device drivers for disks employ a *disk scheduling* policy to perform disk I/O operations in a suitable order. We shall discuss the following disk scheduling policies before describing disk scheduling in modern systems:

- *First come-first served (FCFS) scheduling*: Select the I/O operation that was requested earliest.
- *Shortest seek time first (SSTF) scheduling*: Select the I/O operation whose seek time from the current position of disk heads is the shortest.
- *SCAN scheduling*: This policy moves the disk heads from one end of the platter to the other, servicing I/O operations on each track or cylinder before moving on to the next one. This is called a scan. When the disk heads reach the other end of the platter, they are moved in the reverse direction and newly arrived requests are processed in a reverse scan. A variant called *look* scheduling reverses the direction of disk heads when no more I/O operations can be serviced in the current direction. It is also called the *elevator algorithm*.
- *Circular SCAN or CSCAN scheduling*: This policy performs a scan as in SCAN scheduling. However, it never performs a reverse scan; instead, it moves the heads back to that end of the platter from where they started and initiates another scan. The *circular look* variant (we will call it *C-look* scheduling) variant moves the heads only as far as needed to service the last I/O operation in a scan before starting another scan.

The FCFS and SSTF disk scheduling policies are analogous to some process scheduling policies discussed in Section 4.2. The FCFS disk scheduling policy is easy to

implement but does not guarantee good disk throughput. The shortest seek time first (SSTF) policy is analogous to the shortest request next (SRN) policy. Hence it achieves good disk throughput but may starve some requests. To implement the SSTF policy, the physical IOCS uses a model of the disk that computes the seek time of an I/O operation given the current head position and the address of the disk block involved in the I/O operation. SSTF and the various scan policies can be efficiently implemented if the IOQs are maintained in sorted order by track number.

Example 12.6 describes the operation of various disk scheduling policies for a set of five I/O requests. The *look* policy performs better than other policies in this example because it completes all I/O operations in the shortest amount of time. However, none of these policies is a clear winner in practice because the pattern of disk accesses cannot be predicted.

Example 12.6 Figure 12.16 summarizes the performance of the FCFS, SSTF, Look and C-Look disk scheduling policies for five I/O requests directed at a hypothetical disk having 200 tracks. The requests are made at different instants of time. It is assumed that the previous I/O operation completes when the system clock reads 160 msec. The time required for the disk heads to move from $track_1$ to $track_2$ is assumed to be a linear function of the difference between their positions:

$$t_{hm} = t_{const} + |track_1 - track_2| \times t_{pt}$$

where t_{const} is a constant, t_{pt} is the per-track head movement time and t_{hm} is the total head movement time. We assume the rotational latency and data transfer times to be negligible, $t_{const} = 0$ msec and $t_{pt} = 1$ msec. A practical value of t_{const} is 2 msec. Also, the formula for t_{hm} is not linear in practice.

Figure 12.16 shows the following details for each decision: time at which the decision is made, pending requests and head position at that time, the scheduled request, and its seek time. The last column shows the total seek time for each policy. The plots show the disk head movement for each policy. Note that the total seek times in different scheduling policies vary greatly. SSTF is better than FCFS; however *look* has the smallest total seek time in this example. It is better than *C-look* because it can reverse the direction of traversal after completing the I/O operation on track 100, and service the requests on tracks 75, 40 and 12, whereas C-Look starts a new scan with the request on track 12.

Scheduling in disks A SCSI disk can accept upto 32 commands concurrently. It stores these commands in a table and uses scheduling to decide the order in which it should perform them. The physical IOCS associates a tag with each I/O command to indicate how the disk should handle the command. This tag is entered in the command table along with other details of the command. It is used while making scheduling decisions. This feature is called *tagged command queuing*.

Scheduling in the disk can surpass scheduling in the physical IOCS because the disk uses a more precise model that considers the seek time as well as the rotational latency of a disk block. Consequently, a disk can make fine distinctions between two I/O commands that appear equivalent to the physical IOCS. As an example, consider

t_{const} and t_{pr}	= 0 msec and 1 msec, respectively
Current head position	= Track 65
Direction of last movement	= Towards higher numbered tracks
Current clock time	= 160 msec

I/O requests:

Serial number	1	2	3	4	5
Track number	12	85	40	100	75
Time of arrival	65	80	110	120	175

Scheduling details:

Policy	Details	Scheduling decisions					Σ Seek time
		1	2	3	4	5	
FCFS	Time of decision	160	213	286	331	391	
	Pending requests	1,2,3,4	2,3,4,5	3,4,5	4,5	5	
	Head position	65	12	85	40	100	
	Selected request	1	2	3	4	5	
	Seek time	53	73	45	60	25	256
SSTF	Time of decision	160	180	190	215	275	
	Pending requests	1,2,3,4	1,3,4,5	1,3,4	1,3	1	
	Head position	65	85	75	100	40	
	Selected request	2	5	4	3	1	
	Seek time	20	10	25	60	28	143
Look	Time of decision	160	180	195	220	255	
	Pending requests	1,2,3,4	1,3,4,5	1,3,5	1,3	1	
	Head position	65	85	100	75	40	
	Selected request	2	4	5	3	1	
	Seek time	20	15	25	35	28	123
C-Look	Time of decision	160	180	195	283	311	
	Pending requests	1,2,3,4	1,3,4,5	1,3,5	3,5	5	
	Head position	65	85	100	12	40	
	Selected request	2	4	1	3	5	
	Seek time	20	15	88	28	35	186

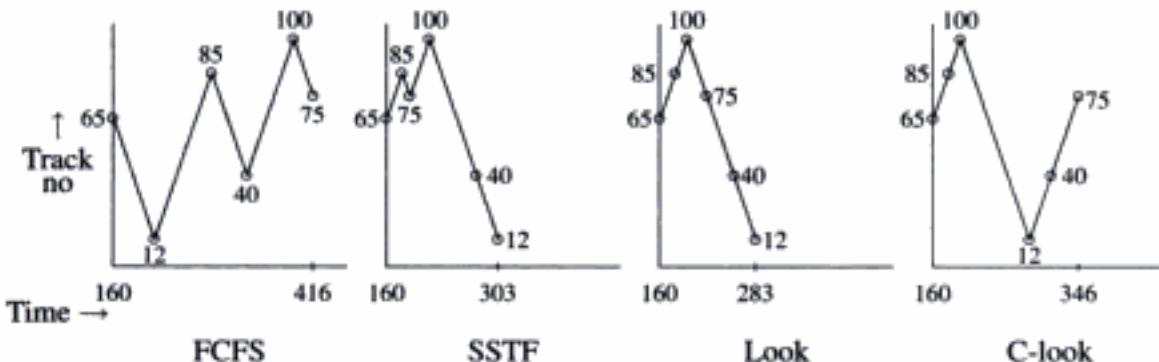


Fig. 12.16 Disk scheduling summary using the FCFS, SSTF, Look and C-look policies

I/O commands that concern disk blocks that are $+n$ and $-n$ tracks away from the current position of disk heads. Both commands have equal seek times, hence the physical IOCS would have to make a random choice between them. However, disk scheduling may discover that one of the commands has a larger rotational latency than the other one. Given the current rotational position of platters, and the position of the required disk block or sector, the disk may find that the disk block that is $+n$ tracks away may be already passing under the heads by the time the heads are positioned on the track. Hence, the disk block can be read only in the next rotation of the disk. The disk block that is $-n$ tracks away may pass under the heads sometime after the heads have been positioned on the track, so its rotational latency would be smaller than that of the disk block that is $+n$ tracks away. Such finer distinctions can contribute to higher throughput of the disk.

The tag in a command can be of three kinds—simple queue tag, ordered queue tag and head-of-queue tag. A *simple queue tag* in a command indicates that the command can be reordered to optimize disk throughput. A command with an *ordered queue tag* indicates that all commands that were entered in the queue earlier should be scheduled before it is scheduled. Such a command should be issued periodically to ensure that I/O operations do not starve, i.e., do not stay indefinitely in the command table. A command with a *head-of-queue tag* should be performed immediately by the disk, i.e., it should be performed ahead of any other command. This feature may be used to ensure that meta-data are written to the disk before file data.

Scheduling in a disk also has its drawbacks. The disk considers all I/O operations as equals, so it might interfere with file-level optimizations performed by access methods. Consider processing of a sequential file with a few buffers. When the file is opened, the access method issues commands to read the first few records of the file in its buffers. To exploit the advantages of buffering, these read commands should be performed in the order in which they are issued; however, the disk might reorder them based on their seek and rotational latencies. Consequently, a later record of the file may be read in while a process waits to access an earlier record!

Drawbacks of disk scheduling lead to the obvious question—should disk scheduling be performed in the disk, in the physical IOCS or in both? Use of a more precise model to compute seek and rotational latencies indicates that scheduling should be performed in the disk. Command ordering requirements to support file-level access optimization implies that scheduling should also be performed in the physical IOCS. An OS designer has to ensure that these schedulers work harmoniously.

12.6 BUFFERING OF RECORDS

The I/O programming techniques discussed in Section 12.4 and illustrated in Figure 12.9 provide the basic features for handling I/O initiation and I/O completion processing. When a process uses these techniques to process a sequential file, its performance depends on the amount of time spent in reading/writing a record and the amount of CPU time spent in processing it.

We use the following notation to discuss the performance of a process that processes a sequential input file.

- t_{io} : *I/O-time per record* (see Eq. (12.1))
- t_c : *copying time per record* (i.e., time required to copy a record from one memory area to another)
- t_p : *processing time per record* (i.e., the CPU time spent by the process in processing a record)
- t_w : *I/O-wait time per record* (i.e., the time during which the process is blocked awaiting completion of a *read* operation for a record)
- t_{ee} : *effective elapsed time per record* (i.e., the clock time since the process issued a *read* operation for a record to the end of processing of the record)

t_w and t_{ee} are analogously defined for an output file. We call the ratio $1 - \frac{t_w}{t_{io}}$, the *CPU-I/O overlap in a program*.

When $t_w = t_{io}$, the CPU-I/O overlap is 0. Performance of a process can be improved by making $t_w < t_{io}$. The best performance of the process would be obtained if $t_w = 0$, which is possible only if a record is available in the memory whenever the process needs it. The basic approach used to achieve this condition is to overlap processing of one record with the reading of the next record (or writing of the previous record). Using this arrangement, t_w would become 0 if I/O for the next record would be completed by the time the previous record is processed. The techniques of buffering and blocking of records are used to achieve this condition.

An *I/O buffer* is a memory area temporarily used to store data involved in an I/O operation. Buffering is used to provide overlap of the I/O and CPU activities in a process—that is, to overlap t_{io} and t_p —so as to reduce t_w . This is achieved by

- *Pre-fetching* an input record into an I/O buffer, or
- *Post-writing* an output record from an I/O buffer.

In pre-fetching, the I/O operation to read the next record is started on a buffer sometime before the record is needed by the process. It may be started while CPU is processing the previous record. This arrangement overlaps a part of t_{io} for reading the next record with t_p for the previous record. This way the process wastes less time waiting for completion of an I/O operation. In post-writing, the record to be written is simply copied into a buffer when the process issues a write operation. Actual output is performed from the buffer sometime later. It can overlap with (a part of) processing for the next record.

To see the effect of buffering on the elapsed time of a process, we consider a program that reads and processes 100 records from a sequential file F. We consider three versions of the program named *Unbuf.P*, *Single.buf.P* and *Multi.buf.P*, which

use zero, one and n buffers, $n > 1$, respectively. We assume $t_{io} = 75$ msec, $t_p = 50$ msec and $t_c = 5$ msec.

Figure 12.17 illustrates operation and performance of processes that represent executions of programs *Unbuf.P*, *Single_buf.P* and *Multi_buf.P*. For convenience, we assume a process to have the same name as the program it executes. For each process, the figure shows the code of the program, illustrates the steps involved in reading and processing a record and shows a timing chart depicting its performance. *Unbuf.P* uses a single area of memory named *Rec_area* to read and process a record of file F (see Figure 12.17(a)). It issues a read operation and awaits its completion before processing a record. The timing diagram shows that I/O is performed on *Rec_area* from $t = 0$ to $t = 75$ msec, and CPU processing occurs between $t = 75$ msec and $t = 125$ msec. Hence $t_w = t_{io}$, $t_{ee} = t_{io} + t_p$ and no CPU–I/O overlap exists in the process. The elapsed time of the process is $100 \times (75 + 50)$ msec = 12.5 seconds.

Figure 12.17(b) illustrates operation of *Single_buf.P*, which uses a single buffer area named *Buffer*. The process issues a read operation to read the first record into *Buffer* and enters the main loop of the process that repeats the following 4-step procedure 99 times:

1. Await end of I/O operation on *Buffer*.
 2. Copy the record from *Buffer* into *Rec_area*.
 3. Initiate an I/O operation on *Buffer*.
 4. Process the record existing in *Rec_area*.
- (12.3)

As shown in the timing diagram of Figure 12.17(b), the process faces an I/O wait in the first step of (12.3) until the read operation on *Buffer* completes. It now performs Steps 2–4 of (12.3). Hence it copies the first record into *Rec_area*, initiates a read operation for the second record, and starts processing the first record. These two activities proceed concurrently, thus achieving CPU–I/O overlap. We depict this fact by drawing a rectangular box to enclose processing of the first record and reading of the second record in the activities part of Figure 12.17(b). Similar overlap is obtained while processing records 2–99. Step 2 of (12.3), i.e., copying of the next record from *Buffer* to *Rec_area*, is performed only after both, reading of the next record and processing of the current record, complete. Thus, for records 2–99, effective elapsed time per record (t_{ee}) is given by

$$t_{ee} = t_c + \max(t_{io}, t_p) \quad (12.4)$$

No processing is in progress when the first record is being read, and no I/O is in progress when the last record is being processed. Thus the operation of the process goes through three distinct phases—the start-up phase when the first record is read, the steady state when records are both read and processed, and the final phase when the last record is processed. Hence the total elapsed time of the process is given by

$$\text{Total elapsed time} = t_{io} + (\text{number of records} - 1) \times t_{ee} + (t_c + t_p) \quad (12.5)$$

Programs

Program Unbuf_P

```

start an I/O operation for
  read (F, Rec_area);
await I/O completion;
while (not end_of_file (F))
begin
  process Rec_area;
  start an I/O operation for
    read (F, Rec_area);
  await I/O completion;
end

```

Program Single_buf_P

```

start an I/O operation for
  read (F, Buffer);
await I/O completion;
while (not end_of_file (F))
begin
  copy Buffer into Rec_area;
  start an I/O operation for
    read (F, Buffer);
  process Rec_area;
  await I/O completion;
end

```

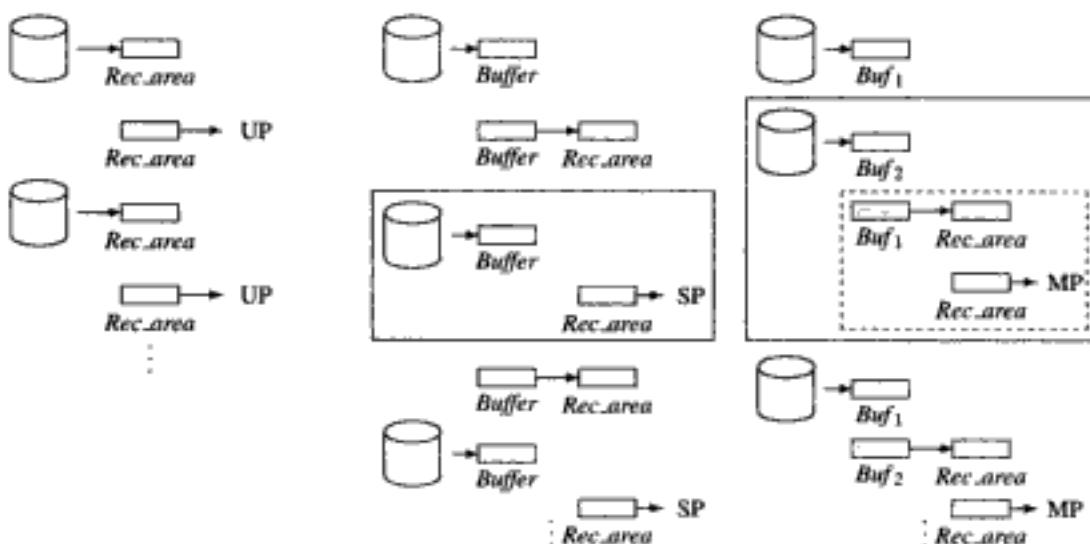
Program Multi_buf_P

```

for i := 1 to n
  start an I/O operation
    for read (F, Bufi);
  await I/O completion on
    Bufi;
k := 1;
while (not end_of_file (F))
  copy Bufk into Rec_area;
  start an I/O operation for
    read (F, Bufk);
  process Rec_area;
  k := (k mod n) + 1;
  await I/O completion on
    Bufk;
end

```

I/O, Copying and Processing activities during execution (UP:Unbuf_P, SP:Single_buf_P, MP:Multi_buf_P)



Timing Diagrams (I: I/O operation, C: Copying, P: Processing)

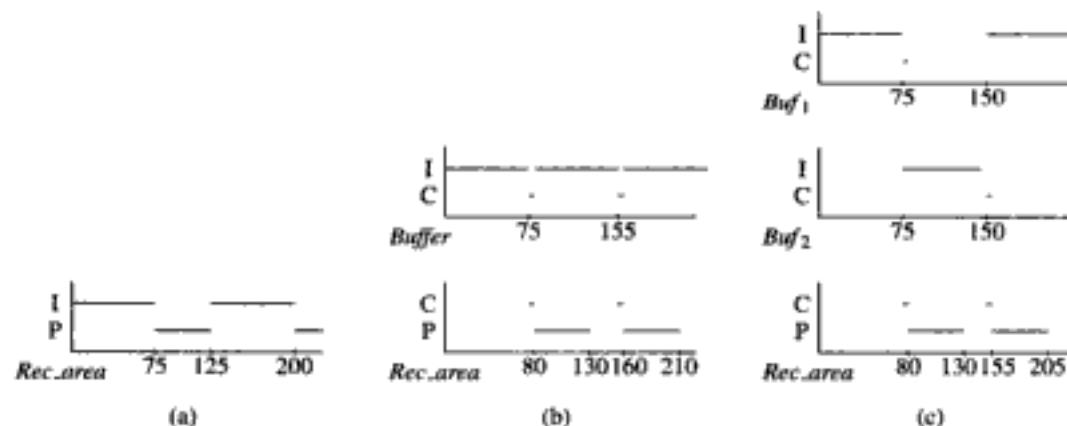


Fig. 12.17 Unbuffered and buffered file processing (note: the *end_of_file* condition is set when the statement *await I/O completion* is executed for an operation that tries to read past the end of a file)

where the three terms in Eq. (12.5) account for the start-up, steady state and final phases of the process. For *Single_buf.P*, t_{ee} is 80 msec and total elapsed time of the process is $75 + 99 \times 80 + 55$ msec = 8.05 seconds.

In the steady state, $t_w = t_{io} - (t_c + t_p)$. Hence buffering would be more effective for smaller values of t_{io} . In fact, $t_w = 0$ if $t_{io} < t_c + t_p$. For example, if t_{io} had been 50 msec, the total elapsed time of the process would have been 5.55 seconds.

Figure 12.17(c) illustrates operation of process *Multi_buf.P*, which uses buffer areas named *Buf₁*, *Buf₂*, ..., *Buf_n*. At the start of file processing, *Multi_buf.P* initiates I/O on all n buffers. Inside the file processing loop, it uses the buffers in turn, following the steps of (12.3) to process a record in a buffer. The statement $k := (k \bmod n) + 1$; ensures that the buffers are used in a cyclic manner. The process waits for I/O to complete on the next buffer, copies the record from the buffer into *Rec_area*, initiates I/O on the buffer, and then processes the record in *Rec_area*.

There is one significant difference between *Multi_buf.P* and *Single_buf.P*. Consider processing of first two records in Figure 12.17(c). When I/O on *Buf₁* completes, *Multi_buf.P* copies the first record from *Buf₁* into *Rec_area* and starts processing the contents of *Rec_area*. At the same time, it initiates a read operation on *Buf₂*. Hence both operations—copying and processing of the record from *Buf₁* and the I/O on *Buf₂*—are performed concurrently. In Figure 12.17(c), we depict this fact by showing a dashed rectangular box around copying and processing of the record from *Buf₁* and enclosing the dashed rectangular box and the I/O operation on *Buf₂* inside another rectangular box. The elapsed time of the process is therefore given by

$$t_{ee} = \max(t_{io}, t_c + t_p) \quad (12.6)$$

$$\text{Total elapsed time} = t_{io} + (100 - 1) \times t_{ee} + (t_c + t_p) \quad (12.7)$$

Total elapsed time of *Multi_buf.P* is $75 + 99 \times 75 + 55$ msec = 7.555 seconds, which is marginally better than *Single_Buf.P*'s elapsed time of 8.05 seconds.

The ratio of the elapsed times of *Unbuf.P* and *Multi_buf.P* is the speed-up factor due to buffering. Considering the steady state, the speed-up factor is approximately $\frac{t_{io} + t_p}{t_{ee}}$. From Eq. (12.6), it can be seen that its best value is obtained when $t_{io} = t_c + t_p$. This value has the upper bound of 2.

Consider the operation of *Multi_Buf.P* when > 1 buffer is used. Figure 12.18 illustrates a typical situation during execution of *Multi_Buf.P*. The CPU has recently copied the record from *Buf_{i-1}* into *Rec_area* and started an I/O operation on *Buf_{i-1}*. Thus, I/O has been initiated on all n buffers. Some of the I/O operations, viz. those on *Buf_j* ... *Buf_{j-1}*, are already complete. I/O is currently in progress for *Buf_j*, while *Buf_{j+1}* ... *Buf_n*, *Buf₁* ... *Buf_{i-1}* are currently in the queue for I/O initiation. Thus ($j-i$) buffers are full at the moment, I/O is in progress for one buffer and ($n-j+i-1$) buffers are in the queue for I/O.

The value of ($j-i$) would depend on values of t_{io} and t_p . If $t_{io} < t_p$, i.e., I/O for a record is faster than its processing, we can see that buffers *Buf_{i+1}* ... *Buf_n*, *Buf₁* ... *Buf_{i-2}* would be full, and *Buf_{i-1}* would be either under I/O or full when CPU

$t_w > 0$ if $t_{io} > t_p$, or $t_{io} > t_c + t_p$ (see Eqs. (12.4) and (12.6)). Thus both unbuffered and buffered processing of files would benefit from a reduction in t_{io} . The technique of blocking of records is used to reduce the effective I/O time per record.

From Eq. (12.1), $t_{io} = t_a + t_x$, where t_a, t_x are the access time per record and transfer time per record, respectively. Now consider a process that reads or writes two records together in a single I/O operation. The data transfer time for the I/O operation is $2 \times t_x$, so the I/O time for the operation, t_{io}^* , is given by

$$t_{io}^* = t_a + 2 \times t_x \quad (12.8)$$

The I/O operation reads or writes two records, so

$$\text{Effective I/O time per record} = \frac{t_{io}^*}{2} = \frac{t_a}{2} + t_x$$

Thus the effective I/O time has been reduced by writing more than one record in an I/O operation.

Logical and physical records A *logical record* is the unit of data processed by a process. A *physical record* or *block* is the unit of data for transfer to or from an I/O medium. A file is said to employ *blocking* of records if a physical record contains more than one logical record.

Definition 12.2 (Blocking factor) The blocking factor of a file is the number of logical records in one physical record.

A file using blocking factor > 1 is said to contain blocked records. Figure 12.19 shows a file that uses a blocking factor of 3.

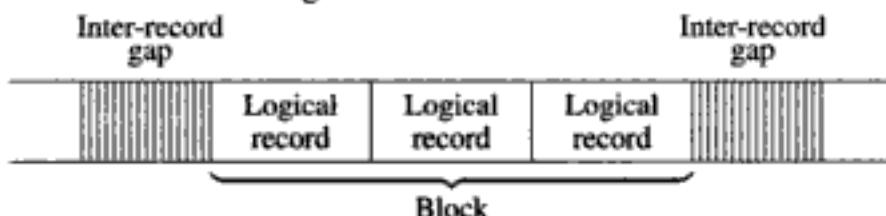


Fig. 12.19 A file with blocking factor = 3

Deblocking actions A read operation on a file containing blocked records transfers m logical records to the memory, where m is the blocking factor. Actions for extracting a logical record from a block for processing in a process are collectively called *deblocking actions*.

Figure 12.20 shows a program that manipulates a file with blocked records in an unbuffered manner. The main loop of the program does not initiate a read operation after processing every logical record. A running index l points at the next logical record within the block being processed. When l exceeds m , the blocking factor, l is reset to 1 and the next block is read. A similar logic can be incorporated into the

programs of Figure 12.17(b),(c) to achieve buffered processing of a file containing blocked records.

```

start an I/O operation for read (F, Rec.area);
await I/O completion;
while (not end_of_file (F))
    for i := 1 to m
        { extract ith record in Rec.area and process it }
    start an I/O operation for read (F, Rec.area);
    await I/O completion;
end

```

Fig. 12.20 Processing of a file with blocked records in an unbuffered manner

Choice of blocking factor Generalizing on the previous discussion, if s_{lr} and s_{pr} represent the size of a logical and physical record, respectively, then $s_{pr} = m \cdot s_{lr}$. The total I/O time per physical record, $(t_{io})_{pr}$, and the I/O time per logical record, $(t_{io})_{lr}$, are given by

$$(t_{io})_{pr} = t_a + m \times t_x \quad (12.9)$$

$$(t_{io})_{lr} = \frac{t_a}{m} + t_x \quad (12.10)$$

Thus blocking reduces the effective I/O time per logical record. If $t_x < t_p$, with an appropriate choice of m it may be possible to reduce $(t_{io})_{lr}$ such that $(t_{io})_{lr} \leq t_p + t_c$. Once this is achieved, from Eqs. 12.4 and 12.6 it follows that buffering can be used to reduce the wait time per record.

Table 12.7 Variation of $(t_{io})_{lr}$ with blocking factor

blocking factor (m)	block size	t_a msec	$m \times t_x$ msec	$(t_{io})_{pr}$ msec	$(t_{io})_{lr}$ msec
1	200	10	0.25	10.25	10.25
2	400	10	0.50	10.50	5.25
3	600	10	0.75	10.75	3.58
4	800	10	1.00	11.00	2.75

Example 12.8 Table 12.7 shows the variation of $(t_{io})_{lr}$ with m for a disk device with $t_a = 10$ msec, transfer rates of 800 K bytes/sec and $s_b = 200$ bytes. t_x , the transfer time per logical record is $\frac{200}{800}$ msec, i.e., 0.25 msec. $(t_{io})_{pr}$ and $(t_{io})_{lr}$ are computed as in Eqs. (12.9) and (12.10). If $t_p = 3$ msec, $m \geq 4$ makes $(t_{io})_{lr} < t_p$.

The value of m is bounded on the lower side by the desire to make $(t_{io})_{lr} \leq t_p + t_c$. On the higher side, it is bounded by the following considerations:

- Memory commitment for file buffers

- Size of a disk track or sector.

A practical value of the blocking factor is the smallest value of m that makes $(t_{io})_{br} \leq t_p + t_c$. Two buffers are now sufficient to eliminate I/O waits for the second and subsequent blocks. As discussed in Section 12.6, a larger number of buffers may be considered to meet the peak requirement of records in a process.

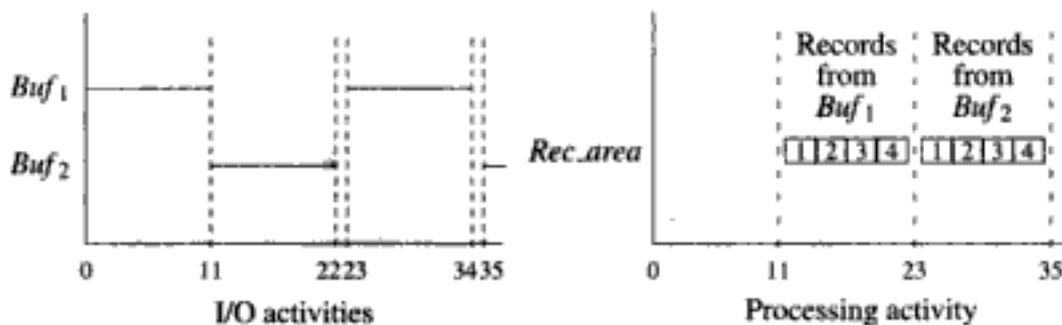


Fig. 12.21 Buffered processing of blocked records using blocking factor = 4 and two buffers

Example 12.9 Figure 12.21 shows the timing chart of processing the blocked file of Example 12.8 with a blocking factor of 4, using two buffers Buf_1 and Buf_2 . We assume t_c to be negligible. Processing starts with read operations on Buf_1 and Buf_2 . The operation on Buf_1 completes at $t = 11$ msec. The process now processes the four logical records copied from Buf_1 . Each logical record requires 3 msec of CPU time, so processing of the four records of Buf_1 consumes 12 msec. This processing overlaps with the read operation on Buf_2 , which consumes 11 msec. Consequently, a block has been read into Buf_2 before processing of records in Buf_1 is completed. CPU starts processing the logical records copied from Buf_2 at $t = 23$ msec. Thus, the process does not suffer any I/O waits after the start-up phase.

12.8 ACCESS METHODS

The IOCS provides a library of access method modules. As mentioned in Section 7.3.4, an access method provides support for efficient processing of a class of files that use a specific file organization. For the fundamental file organizations discussed in Section 7.3, the IOCS may provide access methods for the following kinds of processing:

- Unbuffered processing of sequential-access files
- Buffered processing of sequential-access files
- Processing of direct-access files
- Unbuffered processing of index sequential-access files
- Buffered processing of index sequential-access files.

Access methods for buffered processing of sequential-access and index sequential-access files incorporate the buffering technique illustrated in Figure 12.17(c). Access methods for processing of sequential-access and index sequential-access files

also provide an option to perform blocking of records using the technique shown in Figure 12.20.

We assume that each access method module provides three entry points with the following parameters:

1. AM-open (<internal_id>)
2. AM-close (<internal_id>)
3. AM-read/write (<internal_id>, <record_info>, <I/O_area_addr>)

Modules of the file system and IOCS invoke these functionalities to implement file processing. AM-open is invoked by iocs-open after information has been copied from the directory entry of the file into its FCB. Similarly, AM-close is invoked by iocs-close. AM-read/write are invoked by a file system module. The entry point AM-read is actually the IOCS library module seq-read of Figure 12.13.

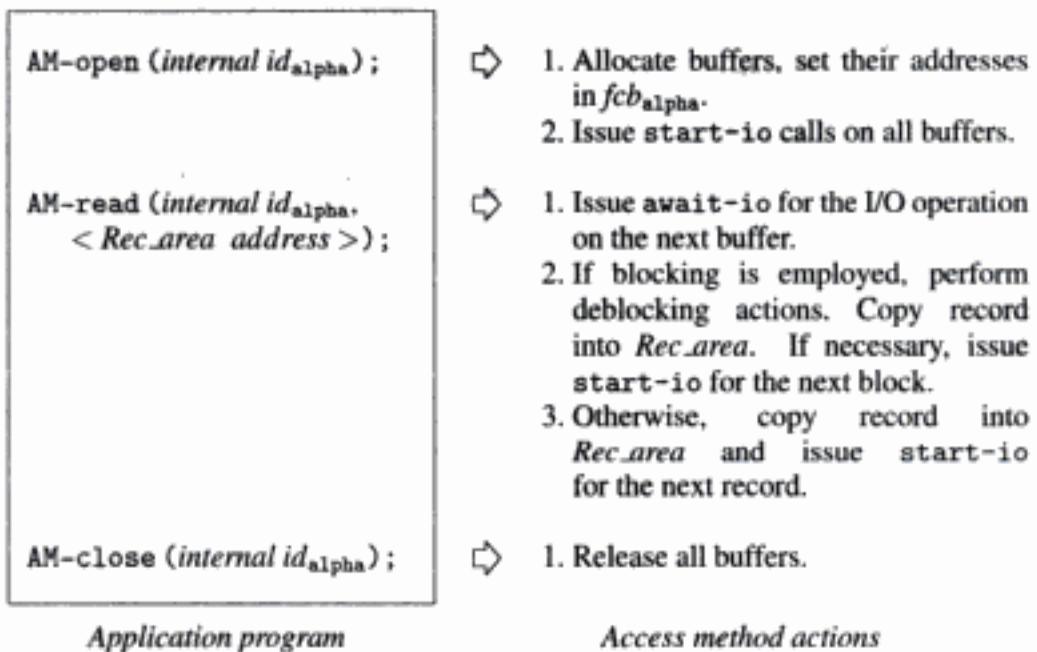


Fig. 12.22 Actions of an access method for buffered reading of a file

Figure 12.22 shows actions of the access method for buffered processing of a sequential-access file alpha. AM-read uses the information in *fcb_{alpha}*, including *fmt_{alpha}*, to form a pair (*record id*, *byte id*) for the next block in the file. When blocking is employed, this action would be needed only after the last record of a block is delivered to the process. A few actions of the access method would be different if alpha was an output file. AM-write would be invoked to perform write operations. In step 2, it would invoke a module of the file system to allocate more disk space to alpha and enter its address into *fmt_{alpha}*.

12.9 UNIFIED DISK CACHE

An operating system maintains a *disk cache* to speed up repeated accesses to file data and meta-data like file map tables stored on the disk. The cache contains copies of disk blocks that were recently read off files by processes, and copies of output data that a process wishes to write into a file, but are yet to be written to the disk. When a read or write operation is to be initiated on a file, the file system converts the offset of a data byte in the file into an address of a disk block and an offset into the block. It uses the address of the disk block to check whether the block exists in the disk cache. If so, it is accessed to perform the I/O operation; otherwise, the disk block is read into the cache and the I/O operation is completed. If the cache is full, the LRU replacement policy is used to make room for the new page. A modified disk block is written into the file when it is to be removed from the cache.

Use of the disk cache speeds up file accesses. Studies mentioned in Section 12.10.2 indicate that a hit ratios of 0.9 and upwards can be obtained by committing a limited amount of memory to the disk cache. Use of the disk cache has some drawbacks too. An I/O operation involves two copy operations, one between the disk and the disk cache and the other between the disk cache and the address space of the process that initiated the I/O operation. Another drawback is that its performance is sensitive to the file access behavior of processes. For example, reading of a large sequential file by a process would fill the disk cache with parts of this file, so accesses to data in other files would suffer until parts of this file are replaced by blocks from other files. The disk cache also leads to poor reliability of the file system because modified disk blocks contained in it would be lost in the event of a system crash.

The OS also maintains another cache, implicitly or explicitly, called the *page cache*. It contains all code and data pages of processes that exist in memory, including any memory mapped files. A new page is loaded into the cache when a page fault occurs. The page size is typically a few disk blocks, so this operation involves reading a few blocks from a program file or a swap file. This is file I/O. Thus, the disk blocks are first read into the disk cache, and then copied into the page cache. When a modified page is to be removed from memory, it is first copied into the disk cache. From there, it is written to the disk sometime in the future.

Figure 12.23(a) is a schematic showing the disk cache and the page cache. Use of these two caches implies two copy operations when a page-in or page-out operation is to be performed—a copy operation between a disk and the disk cache, and a copy operation between the disk cache and the page cache. Two copies of the page may exist in memory for some time until either the copy in the disk cache or the copy in the page cache is overwritten.

Double copying of data in pages and duplicate copies of pages cause performance problems. An OS designer or a system administrator also has to face a difficult design decision—how much memory to commit to each cache. The decision can affect performance of the system because an undercommitment of memory to the page cache would lead to a reduced degree of multiprogramming and possible thrashing,

modules.

The OS defines a standard device driver interface consisting of a set of pre-defined entry points into the device driver routines. Some of these are:

1. $\langle ddname \rangle .init$: DD initialization routine
2. $\langle ddname \rangle .read/write$: Routines to Read or Write a character
3. $\langle ddname \rangle .int$: Interrupt handler routine.

The $\langle ddname \rangle .init$ routine is called at system boot time. It initializes various flags used by the DD. It also checks for the presence of various devices, sets flags to indicate their presence and may allocate buffers to them. Character I/O is performed by invoking the $\langle ddname \rangle .read$ and $\langle ddname \rangle .write$ routines. The device driver has to provide a strategy routine for block data transfers. The strategy routine is roughly equivalent to the I/O scheduler shown in Figure 12.14. A call on the strategy routine takes the I/O control block address as a parameter. The strategy routine adds this IOCB to an IOQ, and initiates the I/O operation if possible. If immediate initiation is not possible, the I/O operation is initiated subsequently by the interrupt handler.

12.10.2 Buffer Cache

Unix supports only byte-stream files, i.e., sequential files without any structure. In Section 7.12, we discussed the arrangement consisting of inode, file descriptor and file structure that is set up when a file is opened for processing (see Figure 7.30). The buffer cache is a disk cache (see Section 12.9), which is used to provide the advantages of buffering and blocking of data, and also minimize disk operations for frequently accessed data.

The buffer cache consists of a pool of buffers. A buffer is the same size as a disk block. Buffers in the cache are not allocated on a per process or per file basis. Instead, a buffer is allocated to a disk block and it is used by every process that needs to access the data in the disk block. When a file is shared by many processes, the data required by a process may already exist in a buffer if another process has accessed it recently, so a disk access is not needed at every read operation. Similarly, disk accesses are avoided if a process accesses some data more than once.

A header is maintained for each buffer. The header contains three items of information: A pair (*device address, disk block address*) gives the address of the disk block that presently exists in the buffer. A status flag indicates whether I/O is in progress for the buffer, and a *busy* flag indicates whether some process is currently accessing the contents of the buffer.

A hash table is used to speed up search for a required disk block (see Figure 12.24). The hash table consists of a number of buckets, where each bucket points to a list of buffers. A hashing function h is used to access the table. It has the basic property that given any number x , it computes a result that is in the range $1 \dots n$, where n is the number of buckets in the hash table. When a disk block with address

aaa is loaded into a buffer with the address bbb , aaa is hashed with function h to compute a bucket number $e = h(aaa)$ in the hash table. The buffer is now entered in the list of buffers in the e^{th} bucket. Thus, all buffers containing disk blocks whose addresses hash into the e^{th} bucket are located in this list.

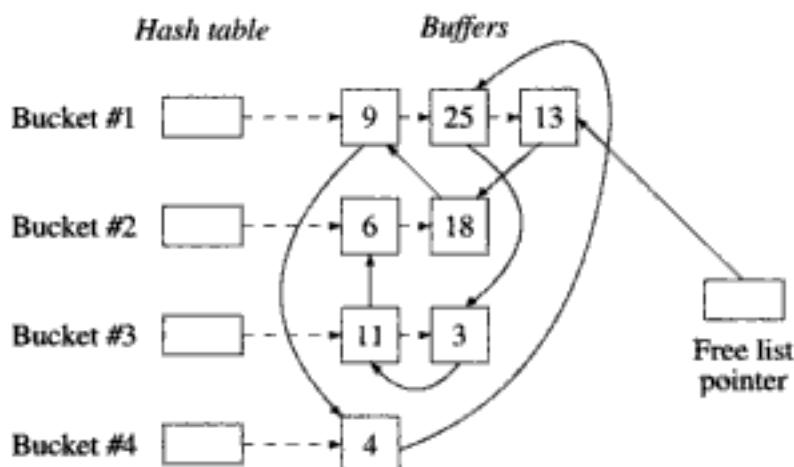


Fig. 12.24 Unix buffer cache

The following procedure is used when a process P performs a read operation on file alpha:

1. Form the pair (*device address, disk block address*) for the byte required by P.
2. Hash *disk block address* to obtain a bucket number. Search buffers in the bucket to check whether a buffer has a matching pair in its header.
3. If a buffer with a matching header does not exist, allocate a free buffer, put the (*device address, disk block address*) information in its header, enter the buffer in the list of an appropriate bucket, set its status flag to 'I/O in progress', queue the buffer for I/O and put P to sleep on completion of I/O.
4. If a buffer with matching header exists, return to P with its address if flags indicate that the I/O operation on the buffer is complete and the buffer is 'not busy'; otherwise, put P to sleep on completion of a read operation on the buffer or buffer 'not busy' condition.
5. If free buffers exist, check whether the next disk block allocated to alpha already exists in memory. If not, allocate a free buffer to it and queue the buffer for a read operation.

This procedure performs pre-fetching of data on a per-process basis by initiating an I/O for the next disk block of the file (Step 5). The advantages of blocking of records are inherent in the fact that a complete disk block is read/written when any byte in it is accessed.

Buffers in the buffer pool are re-used on an LRU basis as follows: All buffers are entered in a free list. A buffer is moved to the end of the list whenever its contents are referenced. Thus least recently used buffers move to the head of the free list. In Step 3, the buffer at the start of the free list is allocated unless it contains some modified data that is yet to be written into the file. In that case, a write operation for the buffer is queued and the next buffer in the list is allocated.

Example 12.10 Figure 12.24 illustrates the Unix buffer cache. Disk blocks 9, 25 and 13 hash into the first entry of the hash table, hence they are entered in the linked list starting on this entry. Similarly 6, 18 and 11, 3 form the linked lists starting on the second and third entries of the hash table. All buffers are also entered in the free list. The position of a buffer in the free list indicates how recently the buffer was accessed by any process. If process P needs to access data residing in disk block 18, the buffer containing block 18 would be moved to the end of the free list. If P now needs data in disk block 21, the first buffer in the free list, i.e., the buffer containing block 13, would be allocated to it if the buffer contents have not been modified since block 13 was loaded. After loading disk block 21 in the buffer, the buffer would be added to an appropriate list in the hash table. This buffer would also be moved to the end of the free list.

12.11 FILE PROCESSING IN LINUX

The organization of Linux IOCS is analogous to that of Unix IOCS. Thus, block and character type I/O devices are supported by individual device drivers, devices are treated like files, and a buffer cache is used to speed-up file processing. However, many IOCS specifics are different. We list some of them before discussing details of disk scheduling in Linux 2.6.

1. Linux kernel modules are dynamically loadable, so a device driver has to be registered with the kernel when loaded and de-registered before being removed from memory.
2. For devices, the *vnode* data structure of the virtual file system (VFS) (see Section 7.11) contains pointers to device-specific functions for the file operations *open*, *close*, *read* and *write*.
3. Each buffer in the disk cache has a buffer header that is allocated in a slab of the slab allocator (see Section 5.10).
4. Dirty buffers in the disk cache are written to the cache when the cache is too full, when a buffer has been in the cache for a long time, or when a file directs the file system to write out its buffers in the interest of synchronization.

I/O scheduling in Linux 2.6 uses some innovations to improve I/O scheduling performance. A read operation needs to be issued to the disk when a process makes a *read* call and the required data does not already exist in the buffer cache. The process would get blocked until the read operation is completed. On the other hand, when a process makes a *write* call, the data to be written is copied into a buffer and

the actual write operation takes place sometime later. Hence the process issuing a *write* call does not get blocked; it can go on to issue more *write* calls. Therefore, to provide better response times to processes, the IOCS performs read operations at a higher priority than write operations.

The I/O scheduler maintains a list of pending I/O operations and schedules from this list. To improve disk throughput, it combines new I/O operations with pending I/O operations whenever possible. When a process makes a *read* or a *write* call, the IOCS checks whether any pending I/O operation involves performing the same operation on some adjoining data. If this check succeeds, it combines the new operation with the pending operation, so that disk heads have to be positioned on the concerned track only once rather than twice, which reduces the movement of disk heads.

Linux 2.6 provides four I/O schedulers. The system administrator can choose the one that best suits the workload in a specific installation. The *No-op* scheduler is simply an FCFS scheduler. The *deadline* scheduler uses Look scheduling as its basis but also incorporates a feature to avoid large delays. It implements Look scheduling by maintaining a scheduling list of requests sorted by track numbers and selecting a request based on the current position of disk heads. However, Look scheduling faces a problem when a process performs a large number of write operations in one part of the disk—I/O operations in other parts of the disk would be delayed. If a delayed operation is a read, it would cause considerable delays in the requesting process. To prevent such delays, the scheduler assigns a deadline of 0.5 second to a read operation and a deadline of 5 seconds to a write operation, and maintains two queues—one for read requests and one for write requests—according to deadlines. It normally schedules requests from the scheduling list; however, if the deadline of a request at the head of the read or write queue expires, it schedules this request, and a couple of more requests from its queue, out of sequence before resuming normal scheduling. The *completely fair queuing* scheduler maintains a separate queue of I/O requests for each process and performs round robin between these queues. This approach avoids large delays for processes.

A process that performs synchronous I/O is blocked until its I/O operation completes. Such a process typically issues the next I/O operation immediately after waking up. When Look scheduling is used, the disk heads would most probably have passed over the track that contains the data involved in the next I/O operation, so the next I/O operation of the process gets serviced only in the next scan of the disk. This causes delays in the process and may cause more movement of the disk heads. The *anticipatory* scheduler addresses this problem. After completing an I/O operation, it waits a few milliseconds before issuing the next I/O operation. This way, if the process that was activated when the previous I/O operation completed issues another I/O operation immediately, that operation may also be serviced in the same scan of the disk.

12.12 FILE PROCESSING IN WINDOWS

Windows uses a centralized cache manager, which is used by both the I/O manager and the virtual memory (VM) manager. Figure 12.25 shows a schematic of the arrangement. The I/O manager consists of the NTFS and the device driver for disks.

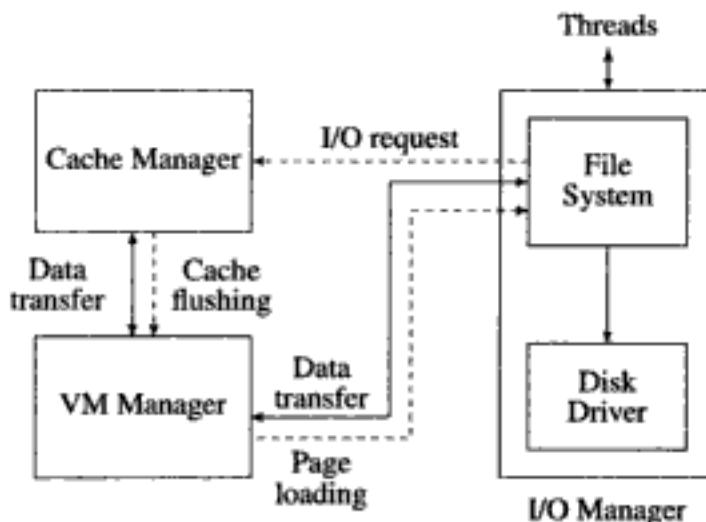


Fig. 12.25 Cache management in Windows 2000

The cache is organized as a set of cache blocks. Each cache block is 256 Kbytes in size. The part of file held in a cache block is called a *view*. A *virtual address control block* (VACB) is used to describe each view. Each entry in a VACB also indicates the number of threads using that view. A set of VACBs is created by the cache manager. It helps to quickly determine whether a required part of a file is in the cache at any moment. It also readily indicates whether a view is currently in use. To facilitate this arrangement, a file is considered to be a sequence of views, each of 256 Kbytes, and a VACB array is set up for it when it is opened.

When a thread makes a request to read/write from a file, the I/O manager passes this request to the cache manager. The cache manager consults the VACB array for the file and determines whether the required portion of the file is a part of some view in the cache. If not, it allocates a cache block and maps the view in the cache block. If the request was a read request, it copies the required data into the caller's address space. This operation involves the VM manager to handle accesses to the caller's address space. If a page fault arises during the copy operation, the VM manager invokes the disk driver through NTFS to read the required page into the memory. For obvious reasons, this operation is performed in a non-cached manner.

The cache manager performs buffering of a file by analyzing the read/write requests. If it finds that the previous few requests indicate sequential access to the file, it prefetches subsequent data blocks. File updates take place in an interesting manner. The data to be written into a file is reflected into the view of the file held in the cache manager. The cache manager periodically nudges the VM manager to

- (a) If two buffers are used, find the value of the smallest blocking factor that can eliminate I/O waits for the second and subsequent blocks.
- (b) If two buffers and a blocking factor of 5 is used, what is the peak requirement of records in a process that this blocking and buffering arrangement can satisfy?
9. A sequential file is recorded using blocking. It is processed using two buffers. The I/O and processing times are as follows:
- | | | |
|--|---|-----------|
| Access time (average) | = | 20 msec |
| Transfer time per record | = | 4 msec |
| Copying time per record | = | 0.5 msec |
| Peak requirement of records by the process | = | 5 records |
| Processing time per record | = | 10 msec |
- Determine optimal values of the blocking factor and the number of buffers.
 What changes, if any, would you make in your design if the peak requirement of records were (i) 3 records, (ii) 8 records?
10. One buffer is used in processing file `info` of Problem 7 in Exercise 7. Calculate the elapsed time of the process if the transfer time per record is 0.5 msec. Explain your calculations.
11. Comment on validity of the following statement: "By judicious selection of the blocking factor and the number of buffers, it is always possible to eliminate I/O waits completely for the second and subsequent blocks in the file."
12. A process is expected to open a file before accessing it. If it tries to access a file without opening, FS performs an open before implementing the access. A system programmers' handbook warns all programmers to open a file before accessing it or suffer a performance penalty. Explain the nature and causes of the performance penalty.
13. Discuss influence of disk scheduling algorithms on effectiveness of I/O buffering.
14. An input file is processed using many buffers. Comment on validity of the following statements:
- "Of all the disk scheduling algorithms, FCFS disk scheduling is likely to provide the best elapsed time performance for the program."
 - "Sector interleaving is useful only while reading the first few records in the file; it is not useful while reading other records."
15. A magnetic tape has a recording density of 8 bits per millimeter along a track. The tape moves at the velocity of 2 metres per second while reading/writing data. The inter-record gap is 0.5 cm wide, and the access time of the tape is 5 msec. A sequential file containing 5000 records, each of size 400 bytes, is stored on this magnetic tape. Calculate the length of the magnetic tape occupied by the file and the total I/O time required to read the file if the file is recorded (a) without blocking, and (b) with a blocking factor of 4.
16. A process uses several buffers while processing a file containing blocked records. A system failure occurs during its execution. Is it possible to resume execution of the process from the point of failure?
17. The speed-up factor resulting from the use of a special I/O technique is the ratio of the elapsed time of a process without using blocking or buffering to the elapsed of the same process using the special I/O technique. In Section 12.6, the speed-up factor due

- to buffering was shown to have an upper bound of 2.
- Develop a formula for speed-up factor when a process does not use buffers while processing a file containing blocked records. Can the value of this speed-up factor exceed 2? If so, give an example.
18. Develop a formula for speed-up factor when a process uses 2 buffers while processing a file containing blocked records and $t_p \geq t_s$.
 19. Describe implications of the Unix buffer cache for file system reliability. Unix supports a system call flush to force the kernel to write buffered output onto the disk. Can a programmer use flush to improve reliability of his files?
 20. The lseek command of Unix indicates offset of the next byte in a sequential file to be read/written. When a process wishes to perform a read/write operation, it issues an lseek command. This command is followed by an actual read/write command.
 - (a) Comment on advantages of using the lseek command in a process.
 - (b) Indicate the sequence of actions that should be executed when a process issues an lseek command.

BIBLIOGRAPHY

Tanenbaum (1990) describes I/O hardware. Ruemmler and Wilkes (1994) present a disk drive model which can be used for performance analysis and tuning. Teorey and Pinkerton (1972) and Hofri (1980) compare various disk scheduling algorithms, while Worthington *et al* (1994) discuss disk scheduling for modern disk drives. Lumb *et al* (2000) discuss how background activities like disk reorganization can be performed during mechanical positioning of disk heads for servicing foreground activities, and the effect of disk scheduling algorithms on effectiveness of this approach.

Chen and Patterson (1990) and Chen *et al* (1994) describe RAID organizations, while Wilkes *et al* (1996) and Yu *et al* (2000) discuss enhancements to RAID systems. Alvarez *et al* (1996) discuss how multiple failures can be tolerated in a RAID architecture, while Chau and Fu (2000) discuss a new layout method to evenly distribute parity information for declustered RAID. Gibson *et al* (1997) discuss file servers for network-attached disks. Nagle *et al* (1999) discuss integration of user-level networking with network-attached storage (NAS). Curtis Preston (2002) discusses NAS and storage area networks (SANs), while Clark (2003) is devoted to the SAN technology. Toigo (2000) discusses modern disks and future storage technologies.

Disk caching is discussed in Smith (1985). Braunstein *et al* (1989) discuss how file accesses are speeded up when virtual memory hardware is used to lookup the file buffer cache.

McKusick *et al* (1996) discuss the Berkeley Fast File system for Unix 4.4BSD. Bach (1986) and Vahalia (1996) discuss other Unix file systems. Ruemmler and Wilkes (1993) present performance studies concerning various characteristics of disk accesses made in the Unix file system. Beck *et al* (2002) and Bovet and Cesati (2003) discuss the I/O schedulers of Linux. Love (2004, 2005) describes the I/O schedulers in Linux 2.6. Custer (1994) describes the Windows NT file system, while Russinovich and Solomon (2005) discuss NTFS for Windows.

1. Alvarez, G. A., W. A. Burkhard, F. Cristian (1996): "Tolerating multiple failures in RAID architectures with optimal storage and uniform declustering." *Proceedings of*

- the 24th Annual International Symposium on Computer Architecture*, 62–72.
2. Bach, M. J. (1986): *The design of the Unix operating system*, Prentice-Hall, Englewood Cliffs.
 3. Beck, M., H. Bohme, M. Dziadzka, U. Kunitz, R. Magnus, C. Schroter, and D. Verworner (2002): *Linux Kernel Programming*, Pearson Education,
 4. Bovet, D. P., and M. Cesati (2003): *Understanding the Linux Kernel*, O'reilly, Sebastopol.
 5. Braunstein, A., M. Riley, and J. Wilkes (1989): "Improving the efficiency of Unix buffer caches," *ACM Symposium on OS Principles*, 71–82.
 6. Chau, A., and A. W. Fu (2000): "A gracefully degradable declustered RAID architecture with near optimal maximal read and write parallelism," *Cluster Computing*, 5 (1), 97–105.
 7. Chen, P. M., and D. Patterson (1990): "Maximizing performance in a striped disk array," *Proceedings of Seventeenth Annual International Symposium on Computer Architecture*, May 1990.
 8. Chen, P. M., E. K. Lee, G. A. Gibson, R. H. Katz, and D. A. Patterson (1994): "RAID—highperformance, reliable secondary storage," *Computing Surveys*, 26 (2), 145–186.
 9. Clark, T. (2003): *Designing Storage Area Networks: A Practical Reference for Implementing Fibre Channel and IP SANs*, (2nd edition), Addison Wesley Professional.
 10. Curtis Preston, W. (2002): *Using SANs and NAS*, O'reilly, Sebastopol.
 11. Custer, H. (1994): *Inside the Windows NT File System*, Microsoft Press, Redmond.
 12. Gibson, G. A., D. Nagle, K. Amiri, F. W. Chang, E. M. Feinberg, H. Gobioff, C. Lee, B. Ozceri, E. Riedel, D. Rochberg, and J. Zelenka (1997): "File server scaling with network-attached secure disks," *Measurement and Modeling of Computer Systems*, 272–284.
 13. Hofri, M. (1980): "Disk scheduling: FCFS vs. SSTF revisited," *Communications of the ACM*, 23 (11), 645–53.
 14. Iyer, S. and P. Druschel (2001): "Anticipatory scheduling: a disk scheduling framework to overcome deceptive idleness in synchronous I/O," *Proceedings of the 18th ACM Symposium on Operating Systems Principles*.
 15. Lampson, B. (1981): "Atomic transactions," in *Distributed Systems—Architecture and Implementation: An Advanced Course*, Goos, G. and J. Hartmanis (Eds), Springer Verlag, Berlin, 246–265.
 16. Love, R. (2004): "I/O schedulers," *Linux Journal*, 118.
 17. Love, R. (2005): *Linux Kernel Development*, Second edition, Novell Press.
 18. Lumb, C. R., J. Schindler, G. R. Ganger, and D. F. Nagle (2000): "Towards higher disk head utilization: extracting free bandwidth from busy disk drives," *Proceedings of the 4th Symposium on Operating Systems Design and Implementation*.
 19. McKusick, M. K., K. Bostic, M. J. Karels, and J. S. Quarterman (1996): *The Design and Implementation of the 4.4BSD Operating System*, Addison Wesley, Reading.
 20. Nagle D., G. Ganger, J. Butler, G. Gibson and C. Sabol (1999): "Network support for network-attached storage," *Proceedings of Hot Interconnects*.
 21. Ruemmler, C., and J. Wilkes (1993): "Unix disk access patterns," *Proceedings of the Winter 1993 USENIX Conference*, 405–420.

Synchronization and Scheduling in Multiprocessor Operating Systems

A multiprocessor system can provide three advantages over a uniprocessor system—higher throughput, computation speed-up and graceful degradation. A multiprocessor OS can execute several processes on the CPUs at the same time, thus achieving a higher throughput. If an application is structured into a set of concurrent processes, this feature can also provide computation speed-up for the application. If a processor fails, the system can continue to operate with its remaining processors. This feature is called *graceful degradation*. It is vital for supporting mission critical and real time applications.

A multiprocessor OS must fully exploit the presence of multiple CPUs to obtain these advantages. It achieves this by addressing three issues. The kernel is structured so that several CPUs can execute the kernel code and perform control functions in parallel. This way the kernel does not become a performance bottleneck. Special synchronization locks are used to reduce preemption and process switching due to synchronization. Processes of an application are scheduled such that they can execute efficiently and harmoniously on several CPUs.

We begin with an overview of the architecture of multiprocessor systems, which provides the background for a discussion of synchronization and scheduling issues. We then discuss the three issues described above.

13.1 ARCHITECTURE OF MULTIPROCESSOR SYSTEMS

Performance and reliability of a uniprocessor system depends on the performance and reliability of the CPU and the memory. CPU and memory performance can be enhanced by using faster chips, and several levels of caches; however, performance

cannot be improved beyond technological limits. Further improvements in performance can be obtained by using multiple CPUs. Hence multiprocessor architectures gained in importance in the late 1990's.

Table 13.1 Advantages of multiprocessors

Advantage	Description
High throughput	Several processes can be serviced by the CPUs at the same time. Hence more work is accomplished.
Computation speed-up	Several processes of an application may be serviced at the same time, leading to faster completion of computation and better response times.
Graceful degradation	Failure of a processor does not halt operation of the system. The system can continue to operate with somewhat reduced capabilities.

Multiprocessor architectures provide the three advantages summarized in Table 13.1. The throughput increases because the OS can schedule several processes in parallel. However, throughput does not vary linearly with the number of processors due to memory contention. We will discuss this aspect later in this section. A computation can complete faster if its processes are scheduled in parallel. The actual speed-up of a computation depends on several factors like the scheduling policy and overhead of the operating system and the inherent parallelism within an application. A fault in the processor brings the operation of a uniprocessor to a halt. However, in a multiprocessor system other processors would be able to continue their operation. This feature provides a vital ability needed in mission critical applications like on-line services.

The memory hierarchy of a uniprocessor computer system consists of the disk, the main memory and the cache (see Section 2.1.1). In multiprocessor systems, the main memory itself may consist of two levels—the local memory and the non-local memory. The speeds with which a processor can access the local and non-local memories depends on the processor–memory associations. This feature is crucial in process synchronization because synchronization is achieved using locks or semaphores stored in shared memory. It is equally important in scheduling because considerable performance degradation would occur if a CPU executes a process that exists in a non-local memory.

Classification of multiprocessor systems Multiprocessor systems are classified into three kinds of systems based on the manner in which processors and memory units are associated with one another.

- *Uniform memory architecture (UMA architecture)* : All CPUs in the system can access the entire memory in an identical manner, i.e., with the same access speed. Some examples of UMA architectures are the Balance system

by Sequent and VAX 8800 by Digital. This architecture is called the *tightly coupled multiprocessor architecture* in older literature. It is also called *symmetrical multiprocessor (SMP) architecture*.

- *Non-uniform memory architecture (NUMA architecture)* : The system consists of a number of nodes. Each node consists of one or more CPUs, a memory unit, and an I/O subsystem. The memory unit of a node is said to be *local* to the CPUs in that node. Other memory units are said to be *non-local*. All memory units together constitute a single address space. Each CPU can access the entire address space, but it can access the local memory unit faster than it can access non-local memory units. Some examples of the NUMA architecture are the HP AlphaServer and the IBM NUMA-Q.
- *No-remote-memory-access architecture (NORMA architecture)* : Each CPU has its local memory. Processors can access remote memory units, however this access is on the network, hence it is slow compared to access to the local memory. The Hypercube system by Intel is an example of a NORMA architecture. A NORMA system is a distributed system according to Def. 2.10, hence we shall not discuss architecture of NORMA systems in this Chapter.

Table 13.2 Features of various interconnection networks

Network	Features
Bus	Low cost. Reasonable access speed at low traffic density. Only one conversation can be in progress at any time.
Cross-bar switch	High cost. Low expandability. Processors and memory units are connected to the switch. A conversation is implemented by selecting a path between a processor and a memory unit. Permits many conversations concurrently.
Multistage interconnection network (MIN)	A compromise between a bus and a cross-bar switch. It consists of many stages of 2×2 cross-bar switches. A conversation is set-up by selecting a path through each stage. Permits some concurrent conversations.

Interconnection networks CPUs in a multiprocessor system use an interconnection network to access memory units. Two important attributes of an interconnection network are cost and effective access speed. Table 13.2 illustrates three popular interconnection networks. Figure 13.1 contains schematics of these networks.

Use of a *bus* in a multiprocessor system is simply an extension of its use in a uniprocessor system. All memory units and all CPUs are connected to the bus. Thus the bus supports data traffic between any CPU and any memory unit. However, only one CPU–memory conversation can be in progress at any time. The organization is simple and inexpensive but it is slow due to bus contention at medium or high traffic densities.

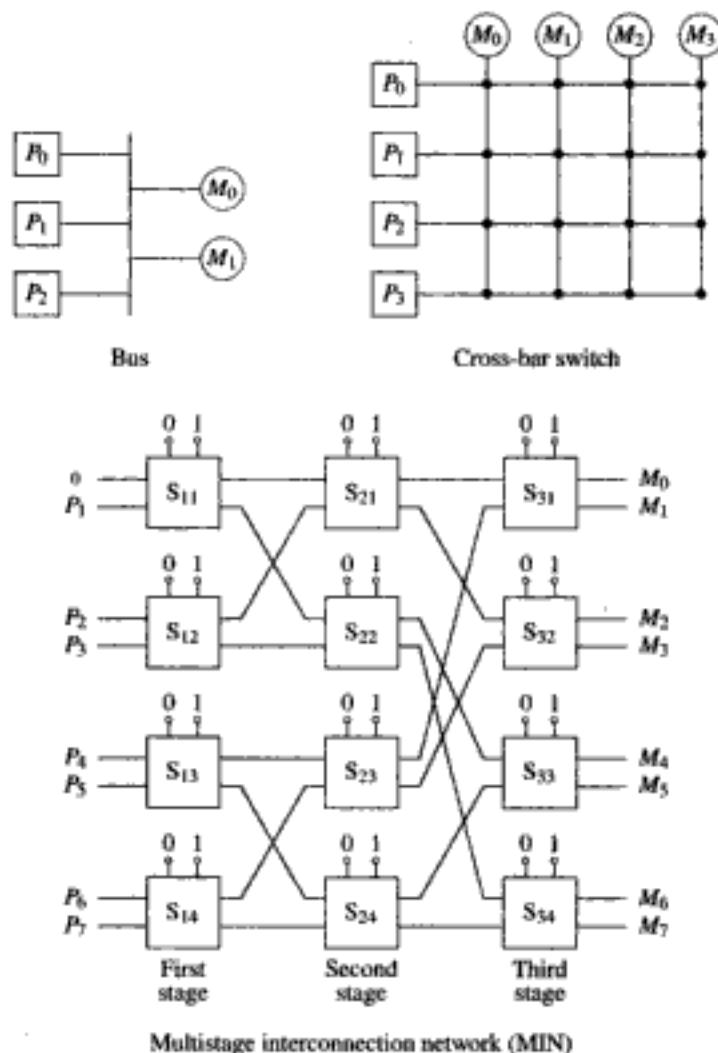


Fig. 13.1 Interconnection networks

A *cross-bar switch* eliminates the contention problem by using a matrix organization wherein CPUs are arranged along one dimension and memory units along the other dimension (see Figure 13.1). Every CPU and memory unit has its own independent bus. When a CPU C_i wishes to access a word located in memory unit M_j , the switch connects the bus of P_i with the bus of M_j . The CPU–memory conversation now takes place over this bus. Naturally this conversation does not suffer any contention due to a conversation between any other (CPU, memory) pair. Contention arises only if two or more CPUs wish to converse with the same memory unit. The cross-bar switch is expensive due to the large number of CPU–memory paths; it also suffers from poor expandability.

A *multistage interconnection network (MIN)* is a compromise between a bus and a cross-bar switch in terms of cost and parallelism. It consists of many stages

containing 2×2 cross-bar switches. Switching in each stage is governed by one bit in the address of the memory unit being accessed. The value of the bit, whether 0 or 1, connects the input of a switch to different switches in the next stage—we say that it selects different switches in the next stage—and so on until a switch in the last stage that selects one of two memory units. In effect, the address bits of a memory unit select a path between a CPU and the memory unit that it wishes to access. Many CPU-to-memory conversations can be in progress at a time provided their paths in MIN do not intersect. A multistage interconnection network has been used in the BBN Butterfly, which has a NUMA architecture.

Figure 13.1 shows an 8×8 Omega network. A column contains switches corresponding to a stage in the interconnection network. For each 2×2 cross-bar switch, a row represents a CPU and a column represents the value of one bit in the binary address of the memory unit to be accessed. If an address bit is 0, the upper output of the cross-bar switch is selected. If the bit is 1, the lower output of the switch is selected. These outputs lead to switches in the next stage. When bits in the address of a memory unit are numbered with bit 0 as the most significant bit, switches in stage i concern bit $(i - 1)$ of the address. The j^{th} switch in stage i is denoted by S_{ij} .

When CPU C_1 wishes to access memory unit M_4 , the interconnection takes place as follows: The binary address of the memory unit is 100. Hence the lower output of switch S_{11} is selected. This leads to S_{22} whose upper output gets selected because the next address bit is 0. This leads to S_{33} whose upper output is selected. It leads to M_4 as desired. Switches S_{13} , S_{24} and S_{34} would be selected if CPU C_4 wishes to access memory unit 7. The interconnection network uses twelve 2×2 switches. In general, an $N \times N$ multistage network uses $\log_2 N$ stages, each stage containing $\frac{N}{2} 2 \times 2$ switches.

Other interconnection networks use combinations of these three fundamental interconnection networks. For example, the IEEE scalable coherent interface (SCI) uses a ring-based network that provides bus-like services but uses fast point-to-point unidirectional links to provide high throughput. A cross-bar switch is used to select the correct unidirectional link connected to a processor.

13.1.1 SMP Architecture

Figure 13.2 shows alternative architectures of SMP systems. In Figure 13.2(a), the CPUs, the memory and the I/O subsystem are connected to the system bus. Each CPU chip contains a level 1 cache that holds the last few instructions and operands accessed by the CPU. In addition, a CPU may use a level 2 cache that is external to it. The memory contains a level 3 cache that holds the last few instructions and operands fetched from the memory. As discussed earlier, only one conversation can be in progress over the bus at any time, hence the bus may become a bottleneck and limit the performance of the system. CPUs would face unpredictable delays while accessing the memory.

Figure 13.2(b) shows a system that uses a cross-bar switch to connect the CPUs

13.1.2 NUMA Architecture

Figure 13.3 illustrates the architecture of a NUMA system. The large rectangle encloses a node of the system. It could consist of a single CPU system; however, it is common to use an SMP system as a node of a NUMA system. Thus a node consists of CPUs, local memory units and an I/O subsystem connected by a local interconnection network. A global port also exists on the local interconnection network. The global ports of all nodes are connected to a high speed global interconnection network capable of providing transfer rates upwards of 1 Gbytes per second. They are used for the traffic between CPUs and non-local memory units. A global port may also contain a cache to hold instructions and data from non-local memories accessed by CPUs of a node. The global interconnection network shown in Figure 13.3 resembles the IEEE scalable coherent interface (SCI). It uses a ring-based network that provides fast point-to-point unidirectional links between nodes.

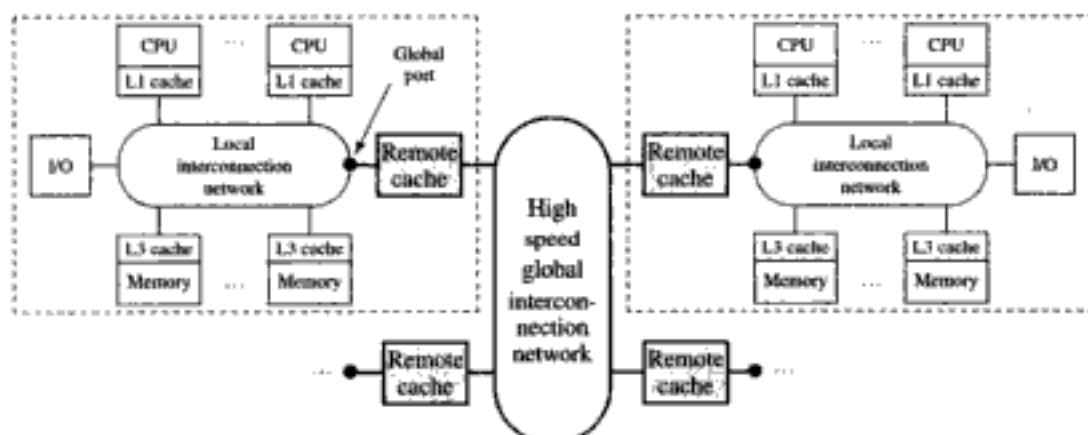


Fig. 13.3 NUMA architecture

As in an SMP, the hardware of a NUMA system must ensure coherence between caches in CPUs of a node. It must also ensure coherence between non-local caches. This requirement can slow down memory accesses and consume part of the bandwidth of interconnection networks.

Ignoring delays in the local and non-local interconnection networks, the effective memory access time to a local memory would depend on the hit ratios in the L1, L2 and L3 caches, and the memory access time. The access time to a non-local memory would depend on hit ratios in the L1, L2, and the remote cache in the global port; and the memory access time.

The nodes in a NUMA architecture are typically high performance SMPs containing 4 or 8 CPUs. Due to the high speed non-local interconnection network, performance of such NUMA architectures is scalable when more nodes are added. The actual performance of a NUMA system would depend on the non-local memory accesses made by processes during their execution. This is an OS issue, which we

discuss in the next section.

13.2 MULTIPROCESSOR OPERATING SYSTEMS

To realize the advantages of high throughput and computation speed-up offered by a multiprocessor system, the CPUs must be used effectively and processes of an application should be able to interact harmoniously. These two considerations will, of course, influence process scheduling and process synchronization. They also affect the operating system's own methods of functioning in response to interrupts and system calls. Table 13.3 highlights the three fundamental issues raised by these considerations. We describe the importance of these issues in this section, and discuss their details in the sections that follow.

Table 13.3 Issues in synchronization and scheduling in a multiprocessor OS

Issue	Description
Kernel structure	Execution of kernel functions should not become a bottleneck. Many CPUs should be able to execute kernel code concurrently.
Process synchronization	Presence of multiple CPUs should result in reducing the overhead of switching between processes, and in reducing synchronization delays.
Process scheduling	Scheduling policy should exploit the presence of multiple CPUs to provide computation speed-up for applications. It should also ensure high cache hit ratios.

Early multiprocessor operating systems functioned in the *master-slave* mode. In this mode, one CPU is designated as the master, and all other CPUs operate as its slaves. Only the master CPU executes the kernel code. It handles interrupts and system calls, and performs scheduling. It communicates its scheduling decisions to other CPUs through *interprocessor interrupts* (IPIs). The primary advantage of the master-slave kernel structure is its simplicity. Its drawback is that a process making a system call is delayed until the system call is handled. The CPU on which it operates is idle until either the process resumes its operation or the master CPU assigns new work to the CPU, both of which can happen only after the master CPU handles the system call and performs scheduling. Hence execution of kernel functions by the master is a bottleneck, which affects system performance. This problem can be solved by structuring the kernel so that many CPUs can execute its code in parallel.

Presence of multiple CPUs can be exploited to reduce synchronization delays in a multiprocessor system. In a uniprocessor system, letting a process loop until a synchronization condition is met denies the CPU to other processes and may lead to priority inversion (see Section 9.2.2). Hence synchronization is performed through blocking of a process until its synchronization condition is met. However, in a multi-

the process it was executing before the interrupt, is still the highest priority process, hence there is no change in its own workload. However, process P_k has a higher priority than process P_j , which was being executed by CPU C_2 , so CPU C_1 has to raise an IPI in CPU C_2 to indicate that its workload has changed. CPUC₂ would now preempt P_j and start executing P_k , and CPU C_1 would resume execution of P_i , which it was executing before the interrupt.

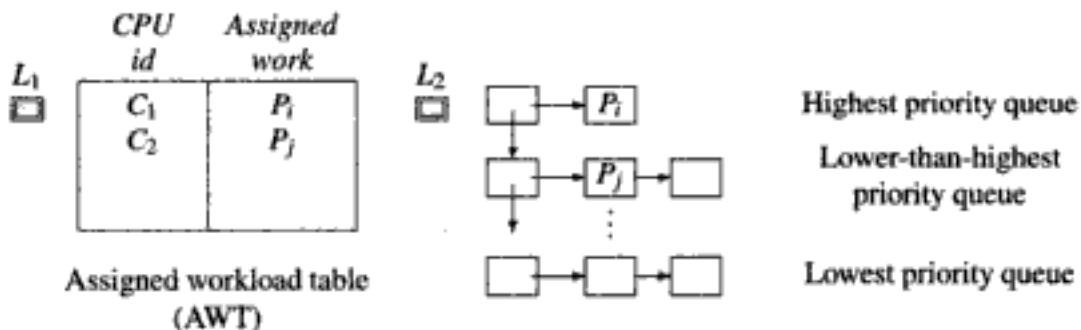


Fig. 13.4 Locking of data structures in an SMP kernel

Figure 13.4 illustrates simple data structures for use by an SMP kernel. The PCB list is analogous to that used in Section 4.4.5, the only difference being that more than one process may be in the *running* state. The kernel maintains an additional data structure named *assigned workload table* (AWT) in which it records the workload assigned to various CPUs.

Example 13.1 Figure 13.4 shows contents of AWT and the PCB list when CPU C_1 and C_2 execute processes P_i and P_j , respectively. The state of both P_i and P_j is shown as *running*. The PCB list contains PCBs in decreasing order by process priority. Process P_k is currently in the *blocked* state. When the interrupt that activates P_k arises, CPU C_1 changes the state of P_i to *ready* since P_i is no longer in execution. It changes the state of P_k to *ready* when it processes the interrupt. It now performs scheduling and realizes that the workload of C_2 must change. So it puts P_k 's state in the communication area and sends an IPI to C_2 .

When C_2 receives the IPI, it stops executing process P_j and records its state in the communication area. It now loads the state of P_k from the communication area, sends an IPI to C_1 and starts executing P_k . When C_1 receives the IPI, it changes the states of P_k and P_j to *running* and *ready*, respectively, moves the state stored by C_2 to the PCB of P_j , and changes the workload of C_2 to P_k in AWT.

SMP kernel Since many CPUs can execute code of the SMP kernel in parallel, its code has to be reentrant (see Section 5.11.3.2). Hence it is essential to ensure mutual exclusion over kernel data structures through use of binary semaphores (see Section 9.8)—we will refer to them as mutex locks. Granularity of locks influences system performance in an obvious manner. If a single lock controls all kernel data structures only one processor could be using the data structures at any time. If separate locks control individual data structures, processors could access different

data structures in parallel. However, care must be taken to ensure that deadlocks do not arise when the kernel needs to access more than one data structure.

Following Section 11.8, resource ranking can be used to prevent deadlocks when several processors wish to access scheduling data structures of the kernel. As shown in Figure 13.4, the AWT and PCB list data structures are protected using locks L_1 and L_2 , respectively. Any processor wishing to perform scheduling tries to set these locks in the order L_2 followed by L_1 .

An SMP kernel provides graceful degradation because it continues to operate despite failures, even though its efficiency may be affected. For example, failure of a processor when it is not executing kernel code does not interfere with operation of other processors in the system, so they would continue to execute normally. Non-availability of the failed processor would affect the process that was executing on it when the failure occurred. It would also affect throughput and response times in the system to some extent.

NUMA kernel In a NUMA system, CPUs experience different memory access times for local and non-local memories, so a process executing on a CPU must make predominant use of local memory units. Also, every node in the system must have its own kernel that schedules processes existing in local memories in CPUs within the node.

Historically, such an arrangement is called a *separate kernel*. The system is divided into several domains, and a separate kernel administers each domain. In a NUMA system, use of a separate kernel for each node ensures that processes incur short memory access times since most of their accesses are to local memories. The kernel should schedule a process on the same CPU all the time. This would ensure good performance of the process, and contribute to good system performance, by ensuring high hit ratios in the CPU's cache, i.e., in the L1 cache. Good hit ratios in the L3 cache would result if the memory allocated to a process exists in a single local memory unit.

The kernel of a node can ensure good performance of an application by allocating memory to all its processes in the same memory unit and devoting a few CPUs to their execution. The notion of an *application region* embodies this idea. An application region consists of a resource partition and an instance of the kernel. The resource partition contains one or more CPUs, some local memory units and a few I/O devices. The kernel of the application region executes processes of only one application. The advantage of this arrangement is that the kernel can optimize the performance of the application through clever scheduling and high cache hit ratios. This can be done without interference from processes of other applications. Operating systems for most NUMA systems offer this or equivalent features.

Use of a separate kernel for a node of a NUMA system or for an application region also has some disadvantages. Accesses to non-local memories become more complex, since they span the domains of more than one kernel. The separate kernel

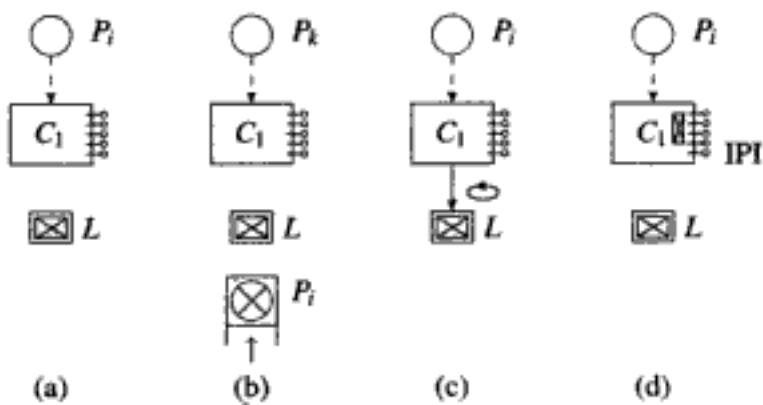


Fig. 13.5 Queued, spin and sleep locks in multiprocessor operating systems

Figure 13.5(b) illustrates the situation after preemption. The id of P_i is entered in the queue of lock L and CPU C_j switches to execution of some other process P_k . When the process that had set lock L completes its use of the critical section the process at the head of L 's queue is activated and the lock is awarded to it.

A process that cannot set a queued lock relinquishes the CPU on which it is executing. Such a process does not use a CPU and does not access memory while it waits to set the lock. The average length of the queue for a lock determines if the solution is scalable. If processes do not require lock L frequently, the queue length is bounded by a constant c (that is, it is never larger than c), so increasing the number of CPUs or processes in the system does not increase the average delay in acquiring the lock. The solution is scalable under these conditions. If processes require lock L frequently, the length of the queue may be proportional to the number of processes. In this case the solution is not scalable.

Spin lock A spin lock differs from a queued lock in that a process that makes an unsuccessful attempt to set a lock does not relinquish the CPU on which it is executing. Instead it enters into a loop in which it makes repeated attempts to set the lock until it succeeds (see Figure 13.5(c)). Hence the name *spin lock*. We depict the situation in which CPU C_j spins on lock L by drawing an arrow from C_j to L . Thus we have a busy wait situation, which a conventional OS tries so hard to avoid! Thus CPU C_j repeatedly accesses the value of the lock and tests it using an indivisible instruction like a Test-and-Set instruction (see Section 9.8.4). This action creates traffic on the memory bus or across the network.

Use of spin locks may degrade system performance on two counts: First, the CPU remains with the process looping on the spin lock hence other processes are denied use of the CPU. Second, memory traffic is generated as the CPU spins on the lock. The latter drawback may not be significant if the memory bus or the network is lightly loaded, but it would cause performance degradation in other situations.

it is able to obtain the primary lock.

A CPU tests a primary lock only once when it wishes to set the lock. If it is unable to set the lock, it spins on its own shadow lock. This way spinning does not generate memory or network traffic. When a CPU that had set the primary lock wishes to reset it, it has to enable a CPU that is spinning on a shadow lock. It achieves this by resetting the shadow lock of the CPU.

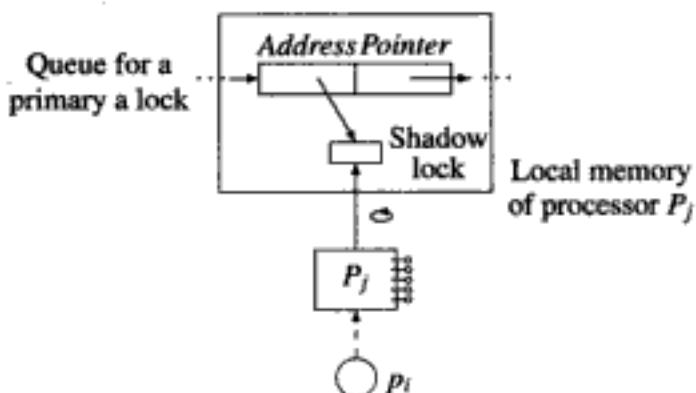


Fig. 13.7 An efficient software solution for process synchronization

Figure 13.7 illustrates an implementation of this scheme. A queue of shadow locks is maintained for each primary lock. Each entry in the queue contains the address of a shadow lock and a pointer to the next shadow lock in the queue. A CPU wishing to set a lock tries to set the primary lock. This step may generate some memory or network traffic since the primary lock may not be local to the requesting CPU. If this attempt fails, the CPU allocates a shadow lock in its own memory, enters its address in the queue of the primary lock and starts spinning on it. The queue may span different memory units in the system. So the step of entering the shadow lock in the queue generates memory or network traffic. The CPU spins on its shadow lock until the lock is reset by a CPU that completes its use of the primary lock. Resetting of a shadow lock also generates memory or network traffic. Needless to say that manipulation of the queue should itself be done under a lock.

13.5 PROCESS SCHEDULING

A process can execute on any CPU in a multiprocessor system. However, its execution performance can be improved by making an intelligent choice of the CPU for executing it, i.e., by deciding *where* to execute it. Execution performance of a group of processes that synchronize and communicate with one another can be improved by deciding *how* and *when* to execute the processes. This Section discusses issues involved in making these decisions.

Choice of the CPU When a process P_i operates on a CPU, say, CPU C_1 , some parts of its address space are loaded into the L1 cache of the CPU. When the CPU is

switched to execution of another process, some of these parts are overwritten by parts of the address space of the new process, however some other parts of P_i 's address space may survive in C_1 's cache memory for some time. These parts are called the *residual address space* of a process. A process is said to have an *affinity* for a CPU if it has a residual address space in its cache. The process would have a higher cache hit ratio on this CPU than on a CPU for which it does not have affinity.

Affinity based scheduling schedules a process on a CPU for which it has an affinity. This technique provides a good cache hit ratio, thereby speeding up operation of the process and reducing the memory bus traffic. Another way to exploit the affinity is to schedule the threads of a process on the same CPU in close succession. However, affinity based scheduling interferes with load balancing across CPUs since processes and threads become tied to specific CPUs. Section 13.6.3 describes how it also leads to scheduling anomalies in the Windows system.

In Section 13.3, we discussed how the SMP kernel permits each CPU to perform its own scheduling. This arrangement prevents the kernel from becoming a performance bottleneck; however, it leads to scheduling anomalies in which a higher priority process is in the *ready* state even though a low priority process has been scheduled. Correcting this anomaly requires shuffling of a processes between CPUs, as indicated in the next example.

Example 13.2 An SMP system contains two CPUs C_1 and C_2 , and three processes P_i , P_j , and P_k with priorities 8, 6, and 5, respectively. Figure 13.8(a) shows the situation in which process P_i is in the *blocked* state due to an I/O operation, and processes P_j and P_k are in operation using CPUs C_1 and C_2 , respectively. When the I/O operation of P_i completes, the I/O interrupt is processed by CPU C_1 , which changes P_i 's state to *ready* and switches itself to service process P_j . So, process P_j , which is the process with the next higher priority, is in the *ready* state, and P_k , whose priority is the lowest, is in operation. To correct this situation, process P_k should be preempted and process P_j should be scheduled on CPU C_2 . Figure 13.8(b) shows the situation in after this is performed.

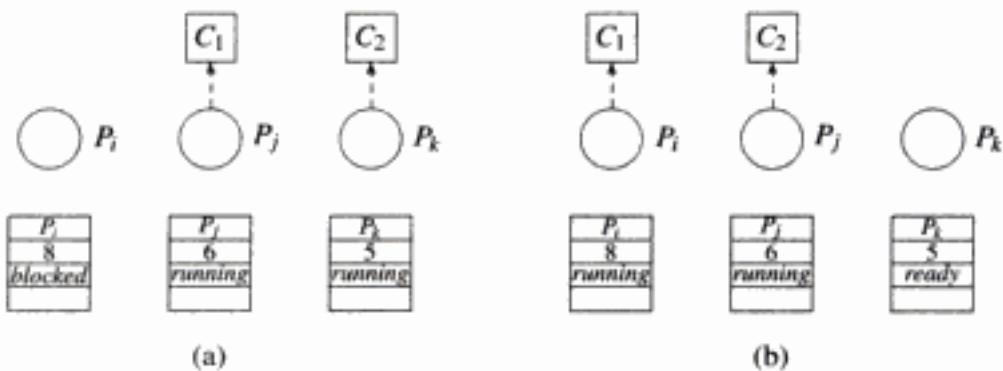


Fig. 13.8 Process P_j is shuffled from CPU C_1 to CPU C_2 when process P_i becomes *ready*

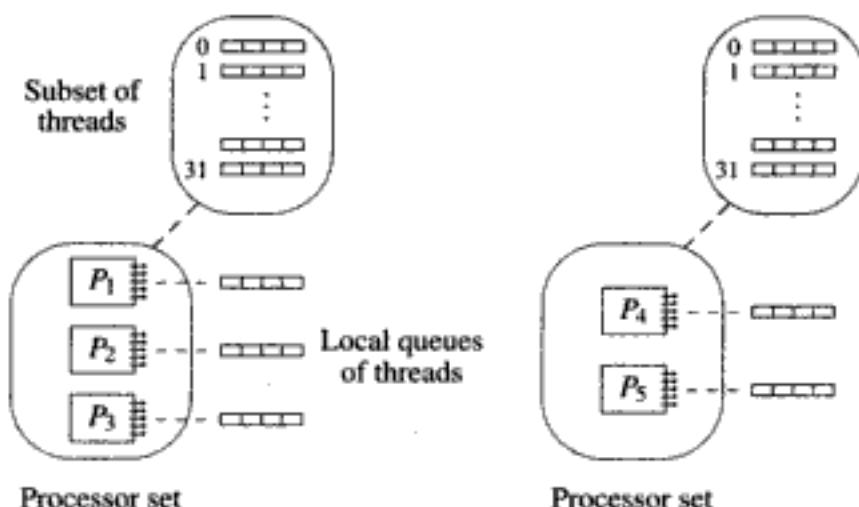


Fig. 13.9 Scheduling in Mach

application. A thread may use a hint to ensure better scheduling when threads of an application need to use synchronization or communication. A discouragement hint reduces the priority of a thread; it is used by a thread that has to spin on a lock that has been set by some other process. A hands-off hint is given by a thread to indicate that it wishes to relinquish the processor to another thread. It can also indicate the identity of a thread to which it wishes to hand over the processor. On receiving such a hint, the scheduler switches the processor to execution of the named thread irrespective of its priority. This feature can be used effectively when a thread spins on a lock while the holder of the lock is preempted. The spinning thread can hand-off its processor to the preempted thread. This action will speed up its execution and lead to an early release of the lock.

13.6.2 Linux

Multiprocessing support in Linux was introduced in the Linux 2.0 kernel. Locking with coarse granularity was employed to prevent race conditions over kernel data structures. Locking granularity became finer with later releases; however, the kernel was still non-preemptible. With Linux 2.6 kernel, the Linux kernel became pre-emptible (see Section 14.10). The Linux 2.6 kernel also employs very fine grained locking.

The Linux kernel provides spin locks for locking of data structures. It also provides a special *reader-write spin lock* which would permit any number of reader processes—that is, processes that do not modify any kernel data—to access protected data at the same time, however, it would permit only one writer process to update the data at any time.

The Linux kernel uses another lock called the *sequence lock* that incurs low overhead and is scalable. The sequence lock is actually an integer that is used as

a sequence counter through an atomic, i.e., *indivisible*, increment instruction. Whenever a process wishes to use a kernel data structure, it simply increments the integer in the sequence lock associated with the data structure, notes its new value, and performs the operation. After completing the operation, it checks whether the value in sequence lock has changed. If the value has changed, the operation is deemed to have failed, so it annuls the operation it just performed and attempts it once again, and so on until the operation succeeds.

Linux uses per-CPU data structures to reduce contention for locks on kernel data structures. As mentioned in Section 13.3, a per-CPU data structure of a CPU is accessed only when the kernel code is executed by that CPU; however, even this data structure needs to be locked because concurrent accesses may be made to it when an interrupt activates an interrupt servicing routine while kernel code was being executed to service a system call. Linux eliminates this lock by disabling preemption of this CPU due to interrupts—the code executed by the CPU makes a system call to disable preemption when it is about to access the per-CPU data structures, and makes another system call to enable preemption when it finishes accessing the per-CPU data structures.

Scheduling for a multiprocessor incorporates considerations of affinity—a user can specify a *hard affinity* for a process by indicating a set of processors on which it must run, and a process has a *soft affinity* for the last processor on which it was run. Since scheduling is performed on a per-CPU basis, the kernel performs *load balancing* to ensure that computational loads directed at different CPUs are comparable. This task is performed by a CPU which finds that its ready queues are empty; it is also performed periodically by the kernel—every 1 msec if the system is idle, and every 200 msecs, otherwise.

The function `load_balance` is invoked to perform load balancing with the id of an underloaded CPU. `load_balance` finds a ‘busy CPU’ that has at least 25 percent more processes in its ready queues than the ready queues of the underloaded CPU. It now locates some processes in its ready queues that do not have a hard affinity to the busy CPU, and moves them to the ready queues of the underloaded CPU. It proceeds as follows: It first moves the highest priority processes in the the *exhausted list* of the busy CPU, because these processes are not likely to have a soft affinity to the busy CPU. If more processes are needed to be moved, it moves the highest priority processes in the *active list* of the busy CPU.

13.6.3 SMP support in Windows

The windows scheduling policy incorporates affinity scheduling, which helps to achieve good memory access performance for a thread by utilizing its residual address space held in the cache of a processor. The *thread processor affinity* attribute of a thread object, in conjunction with the *default processor affinity* attribute of the process object of the process containing the thread defines an affinity set for a thread. If this affinity set is non-null, a thread is always executed on some processor in the

14. Tucker, A., and A. Gupta (1989) : "Process control and scheduling issues for multi-programmed shared memory multiprocessors," *Proceedings of 12th ACM Symposium on Operating System Principles*, 159–166.

chapter 14

Structure of Operating Systems

The main concerns of an OS are management of user computations and resources to achieve effective utilization of a computer system. Table 14.1 summarizes the functions performed by an OS in pursuit of this goal. Elements of the first four functions were discussed in Chapter 2.

Table 14.1 Functions performed by an OS

Function	Details
User interface	Processing of user commands to create computational structures and to access resources
Process management	Initiation and termination of processes, Scheduling, Communication between processes
Memory management	Allocation and deallocation of memory, Swapping, Virtual memory management
I/O management	I/O interrupt processing, initiation of I/O operations
File management	Creation, storage and access of files
Protection & security	Preventing interference with resources and processes

During the lifetime of an operating system, several changes take place in computer systems and computing environments. An operating system must cope with these changes by adapting to new hardware, by being implemented on new computers and by extending its scope to new computing environments. We discuss different philosophies for structuring the modules of an OS to provide these features. We begin by recapitulating some details of OS operation from Chapters 1 and 2.

14.1 OPERATION OF AN OS

Table 14.2 describes three aspects that are important to understand the operation of an OS. The operation of an OS is initialized by the boot procedure, which is loaded into the computer's memory using a combination of a hardware/firmware feature in the computer system, and the software technique of *bootstrapping*. A computer is designed to perform two functions automatically when power is switched on—it instructs the I/O device containing the boot medium to read a program stored on a special track or sector in the boot medium, and starts executing the program. This program loads some other programs in memory which, in turn, load other programs until the entire boot software is loaded in memory.

Table 14.2 Primary aspects of OS operation

Aspect	Description
Booting	Booting is performed when a computer system is switched on. It loads the kernel in memory, initializes its data structures, and hands over control to it.
Interrupt handling	Each event causes an interrupt, which is serviced by taking appropriate actions. Resource allocation, I/O and initiation/termination of programs are performed as aspects of event handling.
Scheduling	The kernel selects one program for execution.

The boot software performs the following functions:

1. Determine the configuration of the system, viz. CPU type, memory size, I/O devices and details of other hardware connected to the system.
2. Load OS programs constituting the kernel in memory.
3. Initialize data structures of the OS.
4. Pass control to the OS.

At the end of the boot procedure, the operating system is in control of the system. The operation of the operating system is interrupt-driven. An event that requires attention of the OS causes an interrupt. The interrupt action passes control to the OS (see Section 2.1.1). Events like I/O completion and end of a time slice are recognized by the hardware, and lead to interrupts of an appropriate kind. When a process makes a system call to request resources, start I/O operations or create/terminate other processes; it too leads to an interrupt called a software interrupt (see Section 2.1.2.2).

Figure 14.1 contains a schematic of the arrangement used to perform event handling. The interrupt code indicates the nature and cause of an interrupt (see Section 2.5.1). The interrupt handler module of the OS analyses the interrupt code to identify the event that has occurred and activates an event handler, which takes appropriate actions and passes control to the scheduler. The scheduler selects a process for execution and passes control to it.

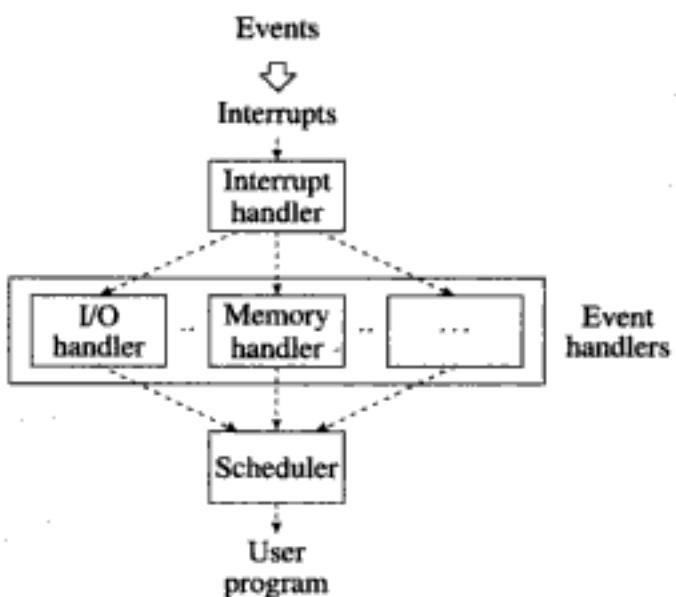


Fig. 14.1 Overview of OS operation

CPU switching occurs twice during the handling of an event. The first switch occurs when the interrupt action passes control to the OS. The interrupt handler saves the CPU state of the process that was interrupted. The second switch occurs when control is to be passed to a user process. The CPU state of the selected process is loaded into the CPU.

The OS operates in an interrupt-driven manner, hence some of the functions of Table 14.1 are invoked by various interrupts. An I/O interrupt invokes some of the I/O management functions. Some part of the process management function is embedded in the scheduler and the dispatcher, which are invoked while exiting from the OS after processing an interrupt. All other functions are invoked through system calls hence their logic can be found in various event handlers (see Figure 14.1).

User processes are loaded into the memory as users log in and start using the system. Thus a collection of OS and user processes exists in the computer's memory at any time during its operation. The memory is divided into two areas—the *system area* and the *user area*—to accommodate the OS and user processes (see Figure 14.2). Some active user processes are situated in the memory while some other user processes may have been swapped out of the memory.

14.2 STRUCTURE OF AN OPERATING SYSTEM

14.2.1 Policies and Mechanisms

A function performed by an OS manages a class of resources or a class of entities. For example, the scheduling function manages processes in the system, and the memory management function manages its memory. While designing a function, A designer needs to think at two distinct levels:

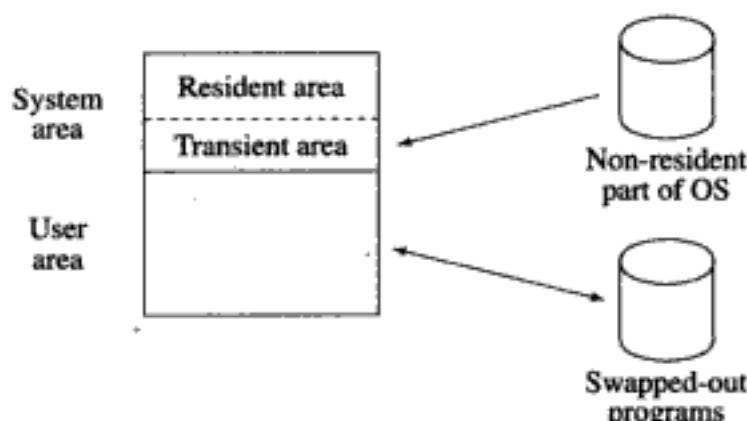


Fig. 14.2 User and OS programs in memory

- **Policy:** A policy specifies the guiding principle that should be used while managing a class of resources or entities.
- **Mechanism:** A mechanism is a specific operation needed to implement a policy.

The distinction between a policy and a mechanism is best expressed as follows: A policy specifies *what* should be done, while a mechanism determines *how* something should be done. A mechanism is implemented as a module that performs a specific task, whereas A policy is implemented as a decision-making module that decides which mechanism modules to call. Example 14.1 discusses policies and mechanisms in the scheduling function.

Example 14.1 An OS uses the round-robin scheduling policy. The following mechanisms are needed to implement it:

Maintain a queue of ready processes

Select the first process from the queue for execution

Switch the CPU to execution of the selected process

A designer has a choice of several scheduling policies. Interestingly, these policies need some common mechanisms, e.g., the dispatching mechanism.

14.2.2 Portability and Extensibility of Operating Systems

Design and implementation of operating systems involves huge financial investments. To protect these investments, an operating system should have a lifetime of more than a decade. Several changes will take place in computer architecture, I/O device technology and application environments during this time, so it should be possible to adapt an OS to these changes.

Two features are important in this context—portability and extensibility. *Portability* refers to the ease with which an OS can be implemented on a computer system with a different architecture. Intrinsically, an OS has poor portability because its code depends on the architecture of a computer system in several ways, for example

Layer 2 implements communication between a process and the operator's console by allocating a virtual console to each process. Layer 3 performs I/O management. It provides easy-to-use functions to perform input and output operations on the devices existing in the system. Intricacies of I/O programming (see Section 12.4) are thus hidden from the higher layer (layer 4), which is occupied by user processes.

The layered approach to OS design suffers from two problems. The operation of a system may be slowed down by the layered structure. Recall that each layer can only interact with adjoining layers. This implies that a request for OS service made by a user process must trickle down from the highest numbered layer to the lowest before the required action is performed by the computer system.

The second problem concerns difficulties in developing a layered design. A layer can access only the immediately lower layer, so all features and facilities needed by it must be available in lower layers. This requirement may pose a problem in ordering the layers. This problem is often solved by splitting a layer into two and putting other layers between them. For example, process handling functions are performed by the lowest layer in the THE operating system (see Table 14.3). The next higher layer performs memory management. This arrangement may be inconvenient because the OS needs to perform memory allocation as a part of process creation. This difficulty can be overcome by splitting process handling into two layers. One layer would perform process management functions like context save, switching, scheduling and process synchronization. This layer would continue to be the lowest layer in the structure. The other layer would be located above the memory management layer. This layer would perform process creation.

14.5 VIRTUAL MACHINE OPERATING SYSTEMS

Different classes of users need different kinds of user service. Hence running a single OS on a computer system can disappoint many users. This problem is solved using a *virtual machine operating system* (VM OS) to control the computer system. The VM OS creates several *virtual machines*. Each virtual machine is allocated to one user, who can use any OS of his own choice on the virtual machine and run his programs under this OS. This way users of the computer system can use different operating systems at the same time. We call each of these operating systems a *guest OS* and call the virtual machine OS the *host OS*.

A *virtual machine* is a virtual resource (see Section 1.3.2.1). It has the same architecture as the computer used by the VM OS, which we call the *host machine*, i.e., it has a virtual CPU capable of executing the same instructions, virtual memory and virtual I/O devices. It may, however, differ from the host machine in terms of some elements of its configuration like memory size and I/O devices. Due to identical architecture of the virtual and host machines, no semantic gap exists between them, so use of a virtual machine does not introduce any overhead and does not cause any performance loss (contrast this with the use of the extended machine layer described in Section 14.4); software intervention is also not needed to run a guest OS on a

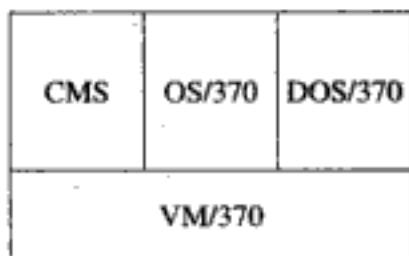


Fig. 14.6 Virtual machine operating system VM/370

OS susceptible to corruption by a user process. The Intel 80x86 family of computers has a feature that provides a way out of this difficulty. The 80x86 computers support 4 CPU modes. Hence the host OS can run with the CPU in the privileged mode, a guest OS can execute processes running under it with the CPU in the user mode but can itself run with the CPU in one of the two remaining modes.

Virtualization is the process by which capabilities of a host machine are made available as capabilities in a virtual machine. Full virtualization would imply that the host machine and a virtual machine have identical capabilities, hence an OS can operate identically while running on a bare machine and on a virtual machine supported by a VM OS. However, full virtualization may lead to lack of security. In Example 14.3, we saw how VM/370 lets a guest OS execute a privileged instruction, but its execution causes an interrupt and VM/370 itself executes the instruction on behalf of the guest OS. This arrangement is insecure because the VM OS cannot differentiate between legitimate use of the privileged instruction by a guest OS and an illegal use of the privileged instruction in a user process running under a guest OS. In modern virtual machine environments, this difficulty is solved by modifying the code of an OS that is to be used as a guest OS. The modified code causes a software interrupt and passes information about the privileged instruction it wished to execute to the VM OS, and the VM OS executes the instruction on its behalf.

In modern computing environments, virtual machines have been used for reducing hardware and operational costs by using existing servers for new applications that require use of different operating systems. VMware and XEN are two virtual machine products that aim at implementing upto 100 guest OSs on a host machine with only a marginal performance degradation when compared to their implementation on a bare machine. Since the VM OS prevents interference between virtual machines, the virtual machine environment can also be used to test a modified OS or a new version of application code on the same server that is used for production runs of the OS or the application.

Virtual machine environments have also been used to provide disaster management capabilities by transferring a virtual machine hosted on a server that has to shut down due to an emergency to another server available on the network. The transfer

of virtual devices (see Section 1.3.2.1). A process is another abstraction provided by the kernel.

Table 14.4 Typical mechanisms in a kernel

Function	Mechanism examples
Interrupt processing	Save CPU state
Scheduling	Dispatch a program Manipulate scheduling lists
Memory management	Set memory protection information Swapping-in/swapping-out Virtual memory mechanisms, e.g., page fault handling, page loading
I/O	I/O initiation I/O completion processing Error recovery
Communication	Inter-process communication mechanisms Networking mechanisms

A kernel-based design may suffer from stratification analogous to the layered OS design (see Section 14.4) because the code to implement an OS command may contain an architecture dependent part, which would be included in the kernel, and an architecture independent part, which would be kept outside the kernel. These parts would have to communicate with one another through system calls, which would add to OS overhead due to interrupt servicing actions. Consider the command to initiate execution of the program in a file named alpha. As discussed at the start of this section, the non-kernel program that implements the command would make four system calls to allocate memory, open file alpha, load the program contained in it into memory and initiate its execution; which would incur considerable overhead. Some operating system designs reduce OS overhead by including the architecture independent part of a function's code also in the kernel. Thus, the non-kernel program that initiated execution of a program would become a part of the kernel. Other such examples are process schedulers (including scheduling policies), I/O scheduling parts of device drivers, core file system services and memory management policies. These inclusions reduce OS overhead; however, they also reduce portability of the OS.

Kernel-based operating systems offer poor extensibility because addition of new functionalities to the OS would require modification of the kernel to provide new functions and services.

14.6.1 Dynamically Loadable Kernel Modules

A recent trend in kernel design is to structure the kernel as a set of dynamically loadable modules. Each module has a well-specified interface through which it interacts

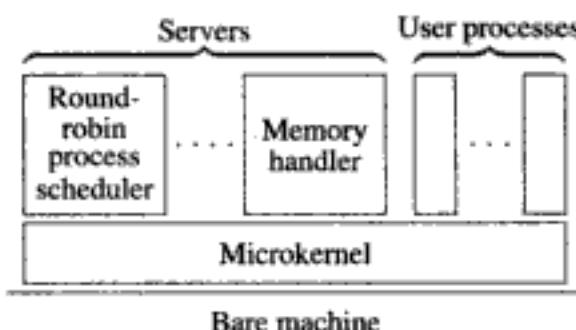


Fig. 14.8 Structure of microkernel-based operating systems

trum of computer systems, from palm-held systems to large parallel and distributed systems. This vision has been realized to some extent. The Mach microkernel has been used to implement several different versions of Unix. The distributed operating system Amoeba uses an identical microkernel on all computers in a distributed system including workstations, servers and large multiprocessors.

Interpretation of the term ‘essential core of OS code’ has been a matter of some debate, and as a result there is considerable variation in the services included in a microkernel. For example, IBM’s implementation of the Mach microkernel leaves the process scheduling policy and device drivers outside the kernel—these functions run as servers. The QNX microkernel includes interrupt servicing routines, process scheduling, inter-process communication and core network services. The L4 microkernel includes memory management and supports only seven system calls. Both QNX and L4 are only 32K bytes in size.

The *exokernel* uses a radically different philosophy: Resource management need not be centralized; it can be performed by applications themselves in a distributed manner. Accordingly, an exokernel merely provides efficient multiplexing of hardware resources, but does not provide any abstractions. Thus an application process sees a resource in the computer system in its raw form. This approach results in extremely fast primitive operations, 10–100 times faster than when a monolithic Unix kernel is used. For example, data that is read off an I/O device passes directly to the process that requested it; it does not go through the exokernel, whereas it would have gone through the Unix kernel. Since traditional OS functionalities are implemented at the application-level, an application can select and use an OS from a library of operating systems. The OS executes as a *process* in the non-kernel mode and uses features of the Exokernel.

It can be argued that certain services *must* be provided by a microkernel. These include memory management support, interprocess communication and interrupt servicing. Memory management and interrupt servicing would be invoked by higher level modules in the OS code that exist outside the microkernel. The interrupt servicing routine would accept interrupts and pass them to higher level modules for

processing.

The advantages of microkernels are somewhat offset by doubts concerning performance of operating systems based on them. The problem has its origin in the fact that some functionalities of a conventional kernel are split between a microkernel and an OS implemented using the microkernel—the familiar stratification problem again. For example, a kernel includes the complete process management function, which performs creation, scheduling and dispatching of processes, whereas a microkernel might include only process creation and dispatching, and process scheduling might run as a process under the microkernel. Communication between the two parts can cause performance problems.

14.8 CONFIGURING AND INSTALLING THE KERNEL

In 1960's and 1970's it was common for a computer manufacturer to market a computer series that consisted of a number of models with the same basic architecture. The models differed in the size of memory, speed of CPU and peripherals (and, of course, in price!). This difference in the capabilities of the models posed a difficult problem for OS designers. An identical kernel could not be used on all models as it would either under-utilize the capabilities of a model or provide poor performance.

An elaborate procedure called *system generation* was used to obtain a kernel that was well-suited to the configuration of a computer system. The procedure was coded in the form of a system generation tool. An *OS distribution* was a set of magnetic tapes that contained the system generation tool and other OS code. It was carried physically to the computer system where the OS was to be installed. A specification of the computer system, i.e., details of its CPU, memory and I/O devices; and a specification of the operating environment, were developed by a systems specialist. These specifications formed the input to the system generation tool. The tool generated a kernel version whose data structures contained the specification of the system. The generated kernel was loaded on the disk. Other parts of the OS were then copied onto the disk to form a usable version of the OS.

Operating systems have come a long way since then. Today distributions may be carried on CDs or accessed over the network. A distribution contains the following:

- A kernel containing a minimal set of facilities
- Modules of the OS
- A kernel configuration utility for adapting a kernel to the configuration of the computer system.

In many ways, the kernel configuration utility is a later day variant of a system generation program. The difference is not conceptual in nature, but merely a matter of detail. Given the sophistication of contemporary hardware, it is possible for an OS to find details regarding CPUs, memory sizes and I/O devices from the hardware itself. As described at the start of this Chapter, the boot procedure can query

14.10 THE KERNEL OF LINUX

The Linux operating system provides the functionalities of Unix System V and Unix BSD; it is also compliant with the POSIX standard. It was initially implemented on the Intel 80386 and has since been implemented on later Intel processors and several other architectures.

Linux has a monolithic kernel. The kernel is designed to consist of a set of individually loadable modules. Each module has a well-specified interface which indicates how its functionalities can be invoked and its data can be accessed by other modules. Conversely, the interface also indicates the functions and data of other modules that are used by this module. Each module can be individually loaded into memory, or removed from it, depending on whether it is likely to be used in near future. In principle, any component of the kernel can be structured as a loadable module, but typically device drivers become separate modules.

A few kernel modules are loaded when the system is booted; others are loaded dynamically when needed. At any time, a few of the kernel modules exist in memory. The kernel maintains information about their functions and data for use while loading a new module. When a new module is to be loaded, the kernel 'binds' the module to other modules by obtaining addresses of the functions and data of other modules that are used by the new module and inserting them in appropriate instructions of the new module. This step integrates the new module with modules already in memory, so that all these modules constitute a monolithic kernel. The kernel now adds information about the functions and data of the new module to the information maintained by it.

Use of kernel modules with well-specified interfaces provides several advantages. Existence of the module interface simplifies testing and maintenance of the kernel. An individual module can be modified to provide new functionalities or enhance existing ones. This feature overcomes the poor extensibility typically associated with monolithic kernels. Use of loadable modules also limits the memory requirement of the kernel, because some modules may not be loaded during an operation of the system. To enhance this advantage, the kernel has a feature to automatically remove unwanted modules from memory—it produces an interrupt periodically and checks which of its modules in memory have not been used since the last such interrupt. These modules are delinked from the kernel and removed from memory. Alternatively, modules can be individually loaded and removed from memory through system calls.

The Linux 2.6 kernel, which was released in 2003, removed many of the limitations of the Linux 2.5 kernel and also enhanced its capabilities in several ways. Two of the most prominent improvements were in making the system more responsive and capable of supporting embedded systems. Kernels upto Linux 2.5 were non-preemptible, so if the kernel was engaged in performing a low priority task, higher priority tasks of the kernel were delayed. The Linux 2.6 kernel is preemptible, which makes it more responsive to users and application programs. However, the kernel

should not be preempted when it is difficult to save its state, or when it is performing sensitive operations, so the kernel disables and enables its own preemptibility through special functions. The Linux 2.6 kernel can also support architectures that do not possess a memory management unit (MMU), which makes it suitable for embedded systems. Thus, the same kernel can now be used in embedded systems, desktops and servers. The other notable feature in the Linux 2.6 kernel is better scalability through an improved model of threads, an improved scheduler, and fast synchronization between processes; these features are described in later Chapters.

14.11 ARCHITECTURE OF WINDOWS

Figure 14.10 shows architecture of the Windows OS. The *hardware abstraction layer* (HAL) interfaces with the bare machine and provides abstractions of the I/O interfaces, interrupt controllers and interprocessor communication mechanisms in a multiprocessor system. The kernel uses the abstractions provided by HAL to provide basic services such as interrupt processing and multiprocessor synchronization. This way, the kernel is shielded from peculiarities of a specific architecture, which enhances its portability. The HAL and the kernel are together equivalent to a conventional kernel (see Figure 14.7). A device driver also uses the abstractions provided by the HAL to manage I/O operations on a class of devices.

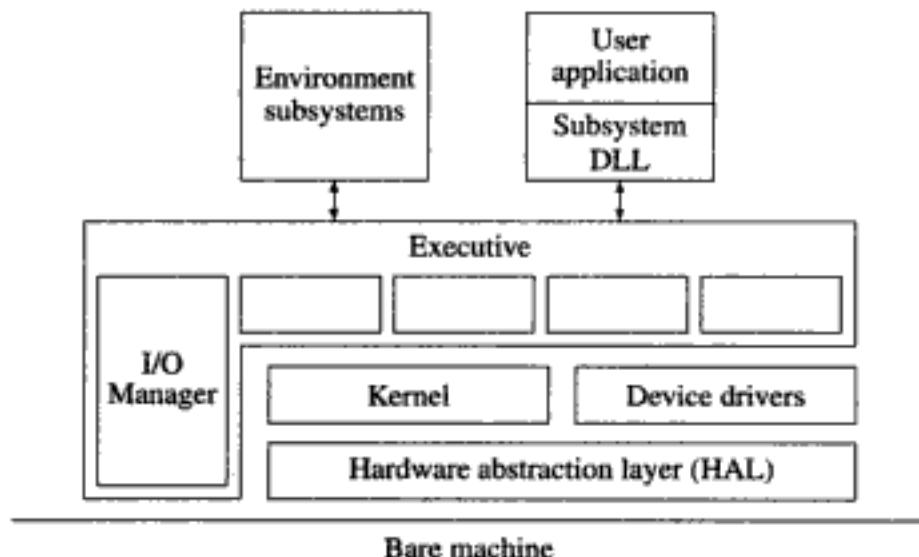


Fig. 14.10 Architecture of Windows

The kernel performs the process synchronization and scheduling functions (see Section 4.8). The executive comprises non-kernel programs of the OS; its code uses facilities in the kernel to provide services such as process creation and termination (see Section 3.5.4), virtual memory management (see Section 6.9), an interprocess message passing facility for client–server communication called the *local procedure call* (LPC), I/O management and a file cache to provide efficient file I/O (see Sec-

3. Beck, M., H. Bohme, M. Dziadzka, U. Kunitz, R. Magnus, C. Schroter, and D. Verworner (2002): *Linux Kernel Programming, Third edition*, Pearson Education.
4. Bovet, D. P., and M. Cesati (2003): *Understanding the Linux Kernel*, O'Reilly, Sebastopol.
5. Creasy, R. J. (1981): "The origin of the VM/370 time-sharing system," *IBM Journal of Research and Development*, **25** (5), 483–490.
6. Dijkstra, E. W. (1968): "The structure of THE multiprogramming system," *Communications of the ACM*, **11**, 341–346.
7. Engler D. R., M. F. Kaashoek, and J. O'Toole (1995): "Exokernel: An operating system architecture for application-level resource management," *Symposium on OS Principles*, 251–266.
8. Liedtke J. (1996): "Towards real microkernels," *Communications of the ACM*, **39** (9).
9. Love, R. (2005): *Linux Kernel Development, Second edition*, Novell Press.
10. Mauro, J., and R. McDougall (2001): *Solaris Internals—Core Kernel Architecture*, Prentice-Hall.
11. McKusick, M. K., K. Bostic, M. J. Karels, and J. S. Quarterman (1996): *The Design and Implementation of the 4.4 BSD Operating System*, Addison Wesley, Reading.
12. Meyer, J., and L. H. Seawright (1970): "A virtual machine time-sharing system," *IBM Systems Journal*, **9** (3), 199–218.
13. Russinovich, M. E., and D. A. Solomon (2005): *Microsoft Windows Internals, Fourth edition*, Microsoft Press, Redmond.
14. Sugarman, J., G. Venkitachalam, and B. H. Lim (2001): "Virtualizing I/O devices on VMware workstation's hosted virtual machine monitor," *2001 USENIX Annual Technical Conference*.
15. Tanenbaum, A. S. (2001): *Modern Operating Systems, Second edition*, Prentice-Hall, Englewood Cliffs.
16. Vahalia, U. (1996): "*UNIX Internals—the New Frontiers*", Prentice-Hall, New Jersey, 1996.
17. Warhol, P. D. (1994): "Small kernels hit it big," *Byte*, January 1994, 119–128.

Part III

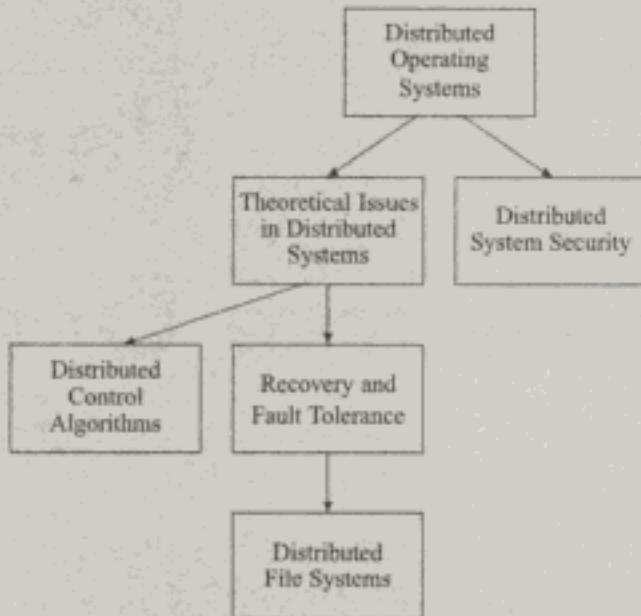
DISTRIBUTED OPERATING SYSTEMS

A distributed system consists of several nodes, where each node is a computer system with its own clock and memory, that can communicate among themselves through a network. A distributed operating system exploits these features as follows: It permits a user to structure his application as a *distributed computation*, which consists of several processes located in different nodes of the distributed system. To service all processes in the system efficiently, it balances computational loads in various computers by transferring processes between nodes, if necessary. This way, processes of an application can compete for CPUs in different nodes, which provides both *computation speed-up* within an application and better performance of the system. Also, the OS uses the redundancy of resources and network links in the system to provide high reliability.

To realize these advantages of computation speed-up, performance and reliability, the OS has to perform control functions like scheduling and deadlock handling on a system-wide basis. Due to the distributed nature of the system, a distributed OS cannot use the notions of *time* and *state* the way a conventional OS uses them to perform control functions, so it performs its control functions in a distributed manner, i.e., through processes in several computers that work in close coordination to make decisions.

Presence of the network has several implications for the distributed OS. A fault does not halt the complete system. It may affect only some computations, or only some parts of a computation, so the distributed OS uses special reliability techniques to minimize the impact of a fault. Communication over the network is slow, so it can seriously erode system performance if processes access their files over the network. To prevent this, distributed file systems employ techniques that reduce network traffic during file processing. The networking component also makes the OS susceptible to security attacks, so it employs special techniques to provide security.

Road Map for Part III



Chapter 15: Distributed Operating Systems

A distributed system consists of hardware components such as computer systems and the network, and software components such as *network protocols*, *distributed computations* and the operating system. This chapter discusses important features of these components and the manner in which these features influence the *computation speed-up, reliability and performance* that can be provided by a distributed operating system.

Chapter 16: Theoretical Issues in Distributed Systems

Time and *state* are two key notions used in a conventional OS. These notions are absent in a distributed system because it contains several computer systems, each with its own clock and memory, that communicate through messages which incur unpredictable communication delays. This chapter discusses practical alternatives to the traditional notions of time and state. These alternative notions are used in the design of distributed control algorithms and recovery schemes used in a distributed OS.

Chapter 17: Distributed Control Algorithms

A distributed OS uses a distributed control algorithm (DCA) to implement a control function. The algorithm involves actions in several nodes of the distributed system. This chapter discusses different classes of distributed control algorithms, and presents algorithms for performing five control functions in a distributed OS—*mutual exclusion, deadlock handling, scheduling, termination detection and leader election*.

election.

Chapter 18: Recovery and Fault Tolerance

A fault may disrupt operation in a system by damaging the states of some data and processes. The focus of *recovery* is to restore some data or process(es) to a consistent state such that normal operation can be restored. *Fault tolerance* provides un-interrupted operation of a system despite faults. This chapter discusses recovery and fault tolerance techniques used in a distributed operating system. *Resiliency*, which is a technique for minimizing the impact of a fault, is also discussed.

Chapter 19: Distributed File Systems

A distributed file system (DFS) stores files in several nodes of a distributed system, so a process and a file used by it might be in different nodes of a system. User convenience in the use of a DFS is determined by the manner in which a DFS arranges files and directories in the nodes of a distributed system. Performance and reliability of a DFS are determined by the manner in which it accesses a required file. This chapter discusses different models used to organize files and directories in nodes of a system, and techniques such as *file caching* and *stateless file servers* that are used to ensure good performance and reliability, respectively.

Chapter 20: Distributed System Security

Presence of the network makes a distributed system susceptible to security attacks launched through interprocess messages. This chapter discusses how processes can ensure message security and verify authenticity of data and messages to thwart such attacks.

15.2 NODES OF DISTRIBUTED SYSTEMS

A distributed system contains different types of nodes. A *minicomputer* node has a single CPU that is shared to service applications of several users. A *workstation* node has a single CPU but services one or more applications initiated by a single user. A node that is a multiprocessor system is called a *processor pool* node. It contains several CPUs, and the number of CPUs may exceed the number of users whose applications are serviced in parallel.

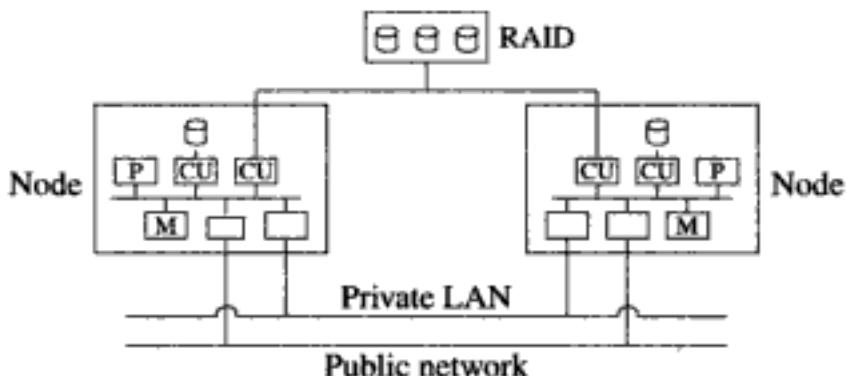


Fig. 15.1 Architecture of a cluster

A *cluster* is a group of hosts that work together in an integrated manner. It constitutes a single node of a distributed system; each individual host is a node *within* the cluster. *Cluster software* controls operation of all nodes in a cluster. It possesses several features of a distributed operating system such as scheduling to achieve computation speed-up, and reliability. Figure 15.1 is a schematic diagram of the architecture of a cluster. The cluster is shown to have two nodes; however, more nodes may be added to provide incremental growth. Each node is a computer system having its own memory and I/O devices. The nodes share disk storage, which could be a multi-host RAID, which offers both high transfer rate and high reliability (see Section 12.3.3), or a storage area network, which offers incremental growth. Each node is connected to two networks—a *private LAN* to which only the nodes in the cluster are connected, and a *public network* through which it can communicate with other nodes in the distributed system.

The cluster software can provide computation speed-up by scheduling sub-tasks in an application on different nodes within the cluster, and reliability by exploiting redundancy of CPUs and resources within the cluster. Section 15.4 describes how these features are implemented in the Windows cluster server and the Sun Cluster.

15.3 NETWORK OPERATING SYSTEMS

A *network operating system* is the earliest form of operating system for distributed architectures. Its goal is to provide resource sharing among two or more computer

a reply sent by the receiver process reaches the sender process. We discuss IPC protocols in Sections 15.5.2–15.5.3.

15.5.1 Naming of Processes

All entities in a distributed system, whether processes or resources, are assigned unique names as follows: Each host in a system is assigned a system-wide unique name, which can be either numeric or symbolic, and each process or resource in a host is assigned an id that is unique in the host. This way, the pair (*<host.name>*, *<process.id>*) is unique for each process and can be used as its name. A process that wishes to send a message to another process, uses a pair like (*human.resources*, P_j) as the name of the destination process, where *human.resources* is the name of a host. This name should be translated into a network address for sending the message.

To easily locate a host in the Internet, the Internet is partitioned into a set of *domains* that have unique names, each domain is partitioned into smaller domains that have unique names in the domain, and so on. A host has a unique name in the immediately containing domain, but its name may not be unique in the Internet, so a unique name for a host is formed by adding names of all the domains that contain it, separated by periods, starting with the smallest domain and ending with the largest domain. For example, the host name *Everest.cse.iitb.ac.in* refers to the server *Everest* in the CSE department of IIT Bombay, which is in the academic domain in India.

Each host connected to the Internet has a unique address known as the Internet protocol address (IP address). The *domain name system* (DNS) is a distributed Internet directory service that provides the IP address of a host with a given name. It has a *name server* in every domain, whose IP address is known to all hosts in the domain. The name server contains a directory giving the IP address of each host in the domain. When a process in a host wishes to send a message to another process with the name (*<host.name>*, *<process id>*), the host performs *name resolution* to determine the IP address of *<host.name>*. If *<host.name>* is not its own name, it sends the name to the name server in its immediate containing domain, which may send it to the name server in its immediate containing domain, and so on until the name reaches the name server of the largest domain contained in *<host.name>*. This name server removes its own name from *<host.name>* and checks whether the remaining string is the name of a single host. If so, it obtains the IP address of the host from the directory and passes it back along the same route along which it had received *<host.name>*; otherwise, the remaining name contains at least one domain name, so it passes the remaining name to the name server of that domain, and so on. Once the sending process receives the IP address of *<host.name>*, the pair (IP address, *<process id>*) is used to send the message to the destination process.

Name resolution using name servers can be slow, so each host can cache some name server data. This technique speeds up repeated name resolution the same way a directory cache speeds up repeated references to the directory entry of a file (see Section 7.15). The name server of a domain is often replicated or distributed to

enhance its availability and to avoid contention.

15.5.2 IPC Semantics

IPC semantics is the set of properties of an IPC protocol. IPC semantics depend on the arrangement of acknowledgments and retransmissions used in an IPC protocol. Table 15.3 summarizes three commonly used IPC semantics.

Table 15.3 IPC semantics

Semantics	Description
At-most-once semantics	A destination process either receives a message once, or does not receive it. These semantics are obtained when a process receiving a message does not send an acknowledgment and a sender process does not perform retransmission of messages.
At-least-once semantics	A destination process is guaranteed to receive a message; however, it may receive several copies of the message. These semantics are obtained when a process receiving a message sends an acknowledgment, and a sender process retransmits a message if it does not receive an acknowledgment before a time-out occurs.
Exactly-once semantics	A destination process receives a message exactly once. These semantics are obtained when sending of acknowledgments and retransmissions are performed as in at-least-once semantics, but the IPC protocol recognizes duplicate messages and discards them.

At-most-once semantics result when a protocol does not use acknowledgments or retransmission. These semantics are used if a lost message does not pose a serious threat to correctness of an application, or if the application knows how to recover from such situations. For example, an application that receives periodic reports from other processes knows when a message is not received as expected, so it may itself communicate with a sender whose message is lost and ask it to resend the message. These semantics are accompanied by high communication efficiency because acknowledgments and retransmissions are not used.

At-least-once semantics result when a protocol uses acknowledgments and retransmission, because a destination process may receive a message more than once if an acknowledgment is lost or delayed due to congestion in the network. A message received for the second or subsequent time is called *aduplicate message*. An application can use at-least-once semantics only if processing of duplicate messages does not pose any correctness problems such as multiple updates of data instead of a single update.

Exactly-once semantics result when a protocol uses acknowledgments and retransmission, but discards duplicate messages. These semantics hide transient faults from both sender and receiver processes; however, the IPC protocol incurs high communication overhead due to handling of faults and duplicate messages.

- sender process is blocked until a reply is received from the destination process.
2. *When a destination process receives a message:* The destination process analyses the request contained in the message and prepares a reply. The reply is copied in a buffer called the *reply buffer* in the destination site and also sent to the sender process.
 3. *When a timeout occurs in the sender process:* The copy of the request stored in the request buffer is retransmitted.
 4. *When the sender process receives a reply:* The sender process sends an acknowledgement to the destination process. It also releases the request buffer, if not already done.
 5. *When a timeout occurs in the destination process:* The copy of the reply stored in the reply buffer is retransmitted.
 6. *When the destination process receives an acknowledgment:* The destination process releases the reply buffer.

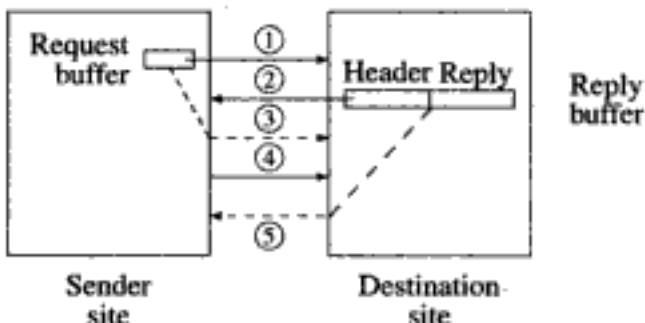


Fig. 15.3 Operation of the request-reply-acknowledgment (RRA) protocol

The sender process is blocked until it receives a reply, so a single request buffer in the sender site suffices irrespective of the number of messages a process sends out, or the number of processes it sends them to. The destination process is not blocked until it receives an acknowledgment, so it could handle requests from other processes while it waits for an acknowledgment. Consequently, the destination site needs one reply buffer for each sender process. The number of messages can be reduced through *piggybacking*, which is the technique of including the acknowledgment of a reply in the next request to the same destination process. Since a sender process is blocked until it receives a reply, an acknowledgment of a reply is even implicit in its next request. Hence only the reply to the last request would require an explicit acknowledgment message.

The RRA protocol has the at-least-once semantics because messages and replies cannot be lost, however, they might be delivered more than once. As mentioned in Table 15.3, duplicate requests would have to be discarded in the destination site to provide exactly-once semantics. It can be achieved as follows: A sender assigns ascending sequence numbers to its requests and includes them in its request messages. The sequence number of a message is copied into its reply and acknowledgement, and into the *header* field of the reply buffer in the destination site. The protocol also separately preserves the sequence number of the last request received from the sender process. A request is a duplicate request if its sequence number is not larger than the preserved sequence number. When a site receives a duplicate request, it

used with minor changes: A destination process preserves the sequence numbers and replies of *all* requests in a pool of buffers. When it recognizes a duplicate request through a comparison of sequence numbers, it searches for the reply of the request in the buffer pool using the sequence number, and retransmits the reply if found in a buffer; otherwise, it simply ignores the request as a reply would be sent after processing the request. Problem 5 of Exercise 15 addresses a refinement of this approach that is needed in practice.

The non-blocking RR protocol can be simplified for use in applications involving *idempotent computations*. An idempotent computation has the property that it produces the same result if executed repeatedly. For example, the computation $i := 5$ is idempotent, whereas the computation $i := i + 1$ is not. When an application involves only idempotent computations, data consistency would not be affected if a request is processed more than once, so it is possible to omit the arrangement for discarding duplicate requests. Read and write operations performed in a file are idempotent, so it is possible to employ the simplified RR protocol while using a remote file server. It has the additional advantage that the file server need not maintain information about which requests it has already processed, which helps to make it *stateless* and more reliable (see Section 19.4.3).

15.6 DISTRIBUTED COMPUTATION PARADIGMS

Data may be located in different sites of a distributed system due to the following considerations:

- *Data replication*: Several copies of a data D may be stored in different sites of a distributed system to provide availability and efficient access.
- *Data distribution*: Parts of a data D may be located in different sites of a system either because the data D is voluminous, or because its parts originate in different sites or are frequently used in different sites.

When data D is neither replicated nor distributed, the OS may position it such that the total network traffic generated by accesses to D by various applications is minimal.

Table 15.4 summarizes three modes of accessing data in a distributed system. In *remote data access*, the data is accessed *in situ*, i.e., where it is located. So it does not interfere with decisions concerning placement of the data. *Data migration* involves moving data to the site of the computation that uses it. It faces difficulties if data is used by many computations or if it is replicated to provide high availability. In the worst case, this approach may force the data to be used strictly by one computation at a time. *Computation migration* involves moving a computation to the site where its data is located. It does not interfere with replication and distribution of data. It also facilitates computation speed-up. Due to these advantages, a variety of features for computation migration have been devised and used in practice.

A *distributed computation* is one whose portions can be executed in different sites for reasons of data access efficiency, computation speed-up or reliability. A *dis-*

considerable manipulation of data located at some specific node S_i , the subcomputation can itself be executed at S_i . It would reduce the amount of network traffic involved in remote data access. Similarly, if a user wishes to send an email to a number of persons at S_i , the mail sending command can itself be executed at S_i .

15.6.4 Case Studies

SUN RPC Sun RPC was designed for client–server communication in NFS, the Sun network file system. NFS models file processing actions as idempotent actions, so Sun RPC provides the at-least-once semantics. This feature makes the RPC efficient; however, it requires applications using RPC to make their own arrangements for duplicate suppression if exactly-once semantics are desired.

Sun RPC provides an interface language called XDR and an interface compiler called `rpcgen`. To use a remote procedure, a user has to write an interface definition for it in XDR, which contains a specification of the remote procedure and its parameters. The interface definition is compiled using `rpcgen`, which produces the following: a client stub, the server procedure and a server stub, a header file for use in the client and server programs, and two parameter handling procedures that are invoked by the client and server stubs, respectively.

The client program is compiled with the header file and the client stub, while the server program is compiled with the header file and the server stub. The parameter handling procedure invoked by the client stub collects parameters and converts them into a machine independent format called the external data representation (XDR). The procedure invoked from the server stub converts parameters from the XDR format into the machine representation suitable for the called procedure.

The Sun RPC schematic has some limitations. The remote procedure can accept only one parameter. This limitation is overcome by defining a structure containing many data members and passing the structure as the parameter. The Sun RPC schematic also does not use the services of a name server. Instead, each site contains a port mapper that is like a local name server. It contains names of procedures and their port id's. A procedure that is to be invoked as a remote procedure is assigned a port and this information is registered with the port mapper. The client first makes a request to the port mapper of the remote site to find which port is used by the remote procedure. It then calls the procedure at that port. A weakness of this arrangement is that a caller must know the site where a remote procedure exists.

Java RMI Java provides a *remote method invocation* (RMI) facility that is language-specific in nature. A server application running on a host creates a special type of object called a *remote object*, whose methods may be invoked by clients operating in other hosts. The server selects a name for the service that is to be offered by a method of the remote object, and registers it with a name server called the `rmiregistry` which runs on the server's host. The `rmiregistry` typically listens on a standard port for registration and invocation requests. The prospec-

Table 15.6 Issues in networking

Issue	Description
Network type	The type of a network is determined by the geographical distribution of users and resources in the system. Two main types of networks are <i>wide area networks</i> (WAN) and <i>local area networks</i> (LAN).
Network topology	Network topology is the arrangement of nodes and communication links in a network. It influences the speed and reliability of communication, and the cost of network hardware.
Networking technology	Networking technology is concerned with transmission of data over a network. It influences network bandwidth and latency.
Naming of processes	Using the domain name service (DNS), the name of a destination process is translated into the pair (IP address, process id).
Connection strategy	A connection strategy decides how to set up data paths between communicating processes. It influences throughput of communication links and efficiency of communication between processes.
Routing strategy	A routing strategy decides the route along which a message would travel through the system. It influences communication delays suffered by a message.
Network protocols	A network protocol is a set of rules and conventions that ensure effective communication over a network. A hierarchy of network protocols is used to obtain a separation of various concerns involved in data transmission and reliability.
Network bandwidth and latency	The bandwidth of a network is the rate at which data is transferred over the network. Latency is the elapsed time before data is delivered at the destination site.

quality laser printers became critical resources, so *local area networks* (LANs) were set up to connect users and resources located within the same office or same building. Since all resources and users in a LAN belonged to the same organization, there was little motivation for sharing the data and resources with outsiders. Hence few LANs were connected to WANs, though the technology for making such connections existed. Advent of the Internet changed the scenario and most LANs and WANs are today connected to the Internet.

Figure 15.7 illustrates WANs and LANs. The LAN consists of PCs, printers and a file server. It is connected to a WAN through a *gateway*, which is a computer that is connected to two (or more) networks and transfers messages between them. Special purpose processors called *communication processors* (CPs) are used in the WAN to facilitate communication of messages between distant hosts. LANs use expensive

Since the basic Ethernet topology is that of a bus, only one conversation can be in progress at any time. The ‘carrier sense multiple access with collision detection’ (CSMA/CD) technology ensures it as follows: A station that wishes to send a message listens to the traffic on the cable to check whether a signal is being transmitted. This check is called *carrier sensing*. The station starts transmitting its frame if it does not detect a signal. However, if many stations find no signal on the cable and transmit at the same time, their frames would interfere with one another causing abnormal voltage on the cable. This situation is called a *collision*. A station that detects a collision emits a special 32 bit jam signal to notify other stations of a collision. All the transmitting stations now back-off by abandoning their transmissions and waiting for a random period of time, before retransmitting their frames. This procedure of recovering from a collision does not guarantee that the frames will not collide again; however, it helps in ensuring that eventually all frames will be transmitted and received without collisions. The frame size must exceed a minimum that facilitates collision detection. This size is 512 bits for the 10M bit and 100M bit Ethernets, and 4096 bits for the Gigabit Ethernet.

Token rings A token ring is a network with a ring topology that uses the notion of a *token* to decide which station may transmit a message at any time. The token is a special message circulating over the network. It has a status bit, which can be either *free* or *busy*. The status bit value *busy* indicates that a message is currently being transmitted over the network, whereas the value *free* indicates that the network is currently idle. Any station that wishes to transmit a message waits until it sees the token with the status bit *free*. It now changes the status to *busy* and starts transmitting its message. Thus a message follows a *busy* token, so only one message can be in transit at any time. A message can be of any length. It need not be split into frames of a standard size.

Every station that sees a message checks whether the message is intended for it; the destination station copies the message. When the transmitting station sees the *busy* token over the network, it resets its status bit to *free*. This action releases the network for another message transmission. When early token release is supported, the destination station resets the status bit of the token to *free*. Operation of the token ring comes to a halt if the token is lost due to communication errors. One of the stations is responsible for recovering from this situation—it listens continuously to the traffic on the network to check for the presence of a token, and creates a new token if it finds that the token has been lost.

Asynchronous transfer mode (ATM) technology ATM is a virtual circuit oriented packet-switching technology (see Sections 15.7.4 and 15.7.5). The virtual circuit is called a *virtual path* in ATM terminology, and a packet is called a *cell*. ATM implements a virtual path between sites by reserving specific bandwidth in physical links situated in a network path between the sites, that is, by reserving a specific portion of the capacity of each physical link for the virtual path. When a physical

is not the destination node, it saves the message in its memory and decides which of the neighbors to send it to, and so on until the message reaches the destination node. This method is called the *store-and-forward* method of transmitting a message. A packet is transmitted similarly. Connection-less transmission can adapt better to traffic densities in communication links than message or packet switching, because a node can make the choice of the link when it is ready to send out a message or packet. It is typically implemented by exchanging traffic information among nodes and maintaining a table in each node that indicates which neighbor to send to in order to reach a specific destination node. However, each node should have a large memory for buffering messages and packets when its outgoing links are congested.

15.7.5 Routing

The routing function is invoked whenever a connection is to be set up. It decides which network path would be used by the connection. Choice of the routing strategy influences ability to adapt to changing traffic patterns in the system. Figure 15.12 illustrates three routing strategies.

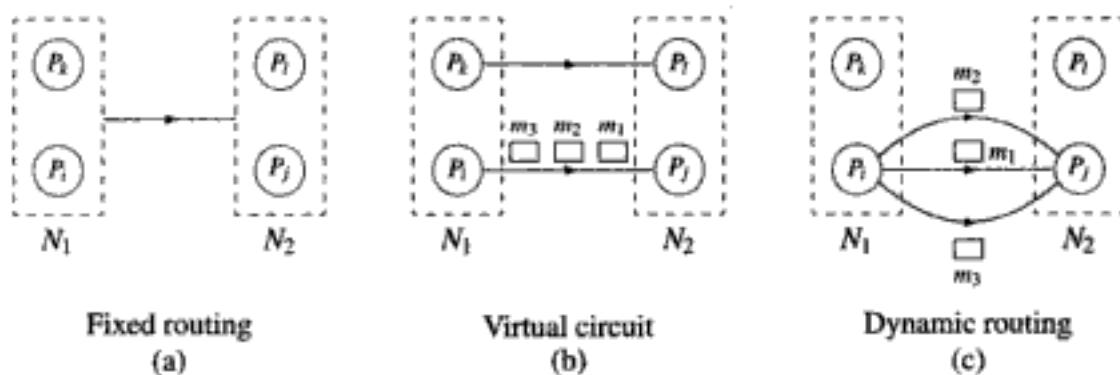


Fig. 15.12 Routing strategies: fixed routing, virtual circuit and dynamic routing

Fixed routing A path is permanently specified for communication between a pair of nodes (see Figure 15.12(a)). When processes located in these nodes wish to communicate, a connection is set up using this path. Fixed routing is simple and efficient to implement—each node merely contains a table showing paths to all other nodes in the system; however, it lacks flexibility to deal with fluctuations in traffic densities and node or link faults. Hence its use can result in delays or low throughputs.

Virtual circuit A path is selected at the start of a session between a pair of processes. It is used for all messages sent during the session (see Figure 15.12(b)). Information concerning traffic densities and communication delays along different links in the system is used to decide the best path for a session. Hence this strategy can adapt to changing traffic patterns and node or link faults, thus it ensures good network throughput and response times.

Dynamic routing A path is selected whenever a message or a packet is to be sent, so different messages between a pair of processes and different packets of a message may use different paths (see Figure 15.12(c)). This feature enables the routing strategy to respond more effectively to changes in traffic patterns and faults in nodes or links. It leads to better throughput and response times than when virtual circuits are used. In the Arpanet, which was the progenitor of the Internet, information concerning traffic density and communication delay along every link was constantly exchanged between nodes. This information was used to determine the current best path to a given destination node.

15.7.6 Network Protocols

Several concerns need to be addressed while implementing communication, such as naming of sites in the system, efficient name resolution, ensuring communication efficiency, and dealing with faults. A *network protocol* is a set of rules and conventions used to implement communication over a network.

A hierarchy of network protocols is used to provide a separation of concerns. Each protocol addresses one or more concerns in communication and provides an interface to the protocols above and below it in the hierarchy. Lower level protocols deal with data transmission related aspects while higher level protocols deal with semantic issues that concern applications. The protocol layers are like the layers of abstraction in a model (see Section 1.1). They provide the same advantages—an entity using a protocol in a higher layer need not be aware of details at a lower layer.

ISO protocol The International Standards Organization (ISO) developed an Open Systems Interconnection reference model (OSI model) for communication between entities in an open system. This model consists of seven protocol layers described in Table 15.7. It is variously called the *ISO protocol*, the *ISO protocol stack*, or the *OSI model*.

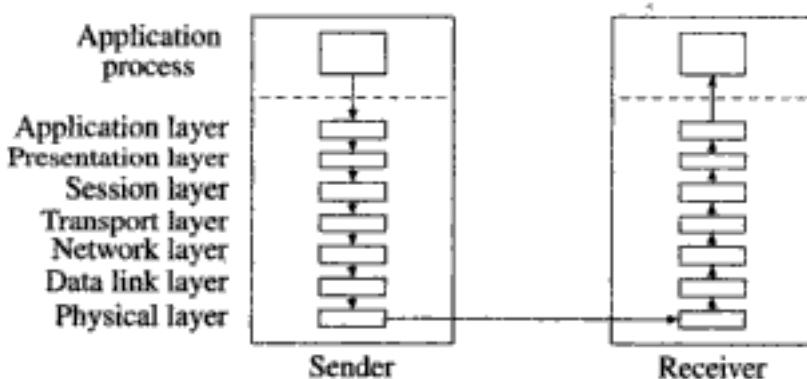


Fig. 15.13 Operation of the ISO protocol stack

Figure 15.13 illustrates operation of the OSI model when a message is exchanged

Table 15.7 Layers of the ISO protocol stack

Layer	Function
1. Physical Layer	Provides electrical mechanisms for bit transmission over a physical link.
2. Data Link Layer	Organizes received bits into frames. Performs error detection on frames. Performs flow control.
3. Network Layer	Performs routing and flow control.
4. Transport Layer	Forms outgoing packets. Assembles incoming packets. Performs error detection and retransmission.
5. Session Layer	Establishes and terminates sessions. Provides for restart and recovery.
6. Presentation Layer	Implements data semantics by performing change of representation, compression and encryption/decryption where necessary.
7. Application Layer	Provides network interface for applications.

by two application processes. The message originates in an application, which presents it to the application layer. The application layer adds some control information to it in the form of a header field. The message now passes through the presentation and session layers, which add their own headers. The presentation layer performs change of data representation and encryption/decryption. The session layer establishes a connection between the sender and receiver processes. The transport layer splits the message into packets and hands over the packets to the network layer. The network layer determines the link on which each packet is to be sent and hands over a link id and a packet to the data link layer. The data link layer views the packet as a string of bits, adds error detection and correction information to it, and hands it over to the physical layer for actual transmission. When the message is received, the data link layer performs error detection and forms frames, the transport layer forms messages and the presentation layer puts the data in the representation desired by the application. The protocol layers are discussed in the following.

The *physical layer* is responsible for the mechanical, electrical, functional and procedural aspects of transmitting bit streams over the network. It is implemented in the hardware of a networking device. RS-232C and EIA-232D are the common physical layer standards.

The *data-link layer* provides error detection, error correction and flow control facilities. It splits the bit stream to be sent into fixed sized blocks called *frames*, and adds a cyclic check sum (CRC) to each frame. It also provides flow control by sending frames at a rate that the receiver can handle. HDLC (high level data link control) is a common protocol of this layer.

The *network layer* is responsible for providing connections and routes between two sites in a system. Popular protocols of this layer are the X.25 protocol, which is

a connection-oriented protocol using virtual circuits, and the Internet protocol (IP), which is a connection-less protocol. Thus, routing is the primary function of this layer, and connection is an optional one. The network layer is mostly redundant in LANs and in systems with point-to-point connections.

The *transport layer* provides error-free transmission of messages between sites. It splits a message into packets, and hands them over to the network layer. It also handles communication errors like non-delivery of packets due to node or link faults. This feature resembles the reliability feature of IPC protocols, hence it is implemented analogously through timeouts and retransmissions (see Section 15.5). ISO has five classes of transport layer protocols named TP0 through TP4. Other common transport layer protocols are the Transport Control Protocol (TCP), which is a connection-oriented reliable protocol, and User Datagram Protocol (UDP), which is a connection-less unreliable protocol.

The *session layer* provides means to control the dialog between two entities that use a connection-oriented protocol. It provides authentication, different types of dialogs (one way, two way alternate or two way simultaneous) and checkpoint-recovery facilities. It provides dialog control to ensure that messages exchanged using non-blocking *send* primitives arrive in the correct order (see Section 15.5). It also provides a quarantine service whereby messages are buffered at a receiver site until explicitly released by a sender. This facility is useful in performing atomic actions in a file (see Section 7.10.2.2) and in implementing atomic transactions (see Section 18.4).

The *presentation layer* supports services that change the representation of a message to address hardware differences between senders and receivers, to preserve confidentiality of data through encryption, and to reduce data volumes through compression.

The *application layer* supports application-specific services like file transfer, e-mail and remote log in. Some popular protocols of this layer are FTP (file transfer protocol), X.400 (e-mail), and rlogin (remote log in).

TCP/IP Protocol The *transmission control protocol/Internet protocol* (TCP/IP) is a popular protocol for communication over the Internet. It has fewer layers than the ISO protocol, so it is both more efficient and more complex to implement. Figure 15.14 shows details of its layers. The lowest layer is occupied by a data link protocol. The *Internet protocol* (IP) is a network layer protocol in the ISO protocol stack; it can run on top of any data link protocol. The IP performs data transmission between two hosts in the Internet. The address of the destination host is provided in the 32 bit IP address format. The IP is a connectionless unreliable protocol; it does not guarantee that packets of a message will be delivered without error, only once, and in the correct order. These properties are provided by the protocols occupying higher levels in the hierarchy.

The Internet protocol provides host-to-host communication using an IP address,

ISO layers 5–7	File transfer protocol (FTP), e-mail, remote login or an application-specific protocol	
ISO layer 4	Transmission control protocol (TCP)	User datagram protocol (UDP)
ISO layer 3	Internet protocol (IP)	
ISO layer 2	Data link protocol	

Fig. 15.14 The transmission control protocol/internet protocol (TCP/IP) stack

which uniquely identifies a host on the Internet. Protocols in the next higher layers provide communication between processes—each host assigns unique 16 bit port numbers to processes, and a sender process uses a destination process address that is a pair (IP address, port number). Use of port numbers permits many processes within a host to use send and receive messages concurrently. Some well-known services such as FTP, telnet, SMTP and HTTP have been assigned standard port numbers by the Internet assigned numbers authority (IANA); other port numbers are assigned by the OS in a host.

As shown in Figure 15.14, two protocols can be used in the layer above the IP, which corresponds to the transport layer, i.e., layer 4, in the ISO protocol stack. The *transmission control protocol* (TCP) is a connection-oriented reliable protocol. It employs a virtual circuit between two processes and provides reliability by retransmitting a message that is not received in an expected time interval (see Section 15.5 for a discussion of acknowledgments and timeouts used to ensure reliable delivery of messages). The overhead of ensuring reliability is high if the speeds of a sender and a receiver mismatch, or if the network is overloaded; hence, the TCP performs *flow control* to ensure that a sender does not send packets faster than the rate at which a receiver can accept them, and *congestion control* to ensure that traffic is regulated so that a network is not overloaded.

The *user datagram protocol* (UDP) is a connectionless, unreliable protocol that neither guarantees delivery of a packet nor ensures that packets of a message will be delivered in the correct order. It incurs low overhead compared to the TCP because it does not have to set up and maintain a virtual circuit or ensure reliable delivery. The UDP is employed in multi-media applications and in video conferencing because the occasional loss of packets is not a correctness issue in these applications—it only leads to poor picture quality. These applications use their own flow and congestion control mechanisms such as reducing the resolution of pictures—and, consequently, lowering the picture quality—if a sender, a receiver or the network is overloaded.

The top layer in the TCP/IP stack is occupied by an application layer protocol like the file transfer protocol, an email protocol such as the SMTP, or a remote log in protocol. This layer corresponds to layers 5–7 in the ISO protocol. When the UDP

is used in the lower layer, the top layer can be occupied by an application-specific protocol implemented in an application process itself.

15.7.7 Network Bandwidth and Latency

When data is to be exchanged between two nodes, networking hardware and network protocols participate in data transfer over a link, and communication processors (CPs) store and forward the data until it reaches the destination node. Two aspects of network performance are how soon the data can reach the destination node, and at what rate the data can be delivered.

Network bandwidth is the rate at which data is transferred over a network. It is subject to various factors such as capacities of network links, error rates and delays at routers, bridges and gateways. Peak bandwidth is the theoretical maximum rate at which data can be transferred between two nodes. Effective bandwidth may be lower than the peak bandwidth due to data transmission errors, which lead to time-outs and retransmissions. *Latency* is the elapsed time between sending of a byte of data by a source node and its receipt at the destination node. It is typically computed for the first byte of data to be transferred. The processing time in the layers of a network protocol and delays due to network congestion contribute to latency.

15.8 MODEL OF A DISTRIBUTED SYSTEM

A system model is employed to reason useful properties of a distributed system, such as the impact of faults on its functioning, and the latency and cost of message communication. A distributed system is typically modeled as a graph

$$S = (N, E)$$

where N and E are sets of nodes and edges, respectively. Each node may represent a host, i.e., a computer system, and each edge may represent a communication link connecting two nodes; however, as discussed later, nodes and edges may also have other connotations. The *degree* of a node is the number of edges connected to it. Each node is assumed to have an *import list* describing non-local resources and services that the node can utilize, and an *export list* describing local resources of the node that are accessible to other nodes. For simplicity, we do not include the name server in the system model (see Section 15.5.1).

Two kinds of graph models of a distributed system are useful in practice. A *physical model* is used to represent the arrangement of physical entities in a distributed system. In this model, nodes and edges have the implications described earlier, i.e., a node is a computer system and an edge is a communication link. A *logical model* is an abstraction. Nodes in a logical model represent logical entities like processes and edges represent relationships between entities. A logical model may use undirected or directed edges. An undirected edge represents a symmetric relationship

like two-way interprocess communication. A directed edge represents an asymmetric relationship like parent-child relationship between processes or one-way interprocess communication. Note that nodes and edges in a logical model may not have a one-to-one correspondence with physical entities in a distributed system.

Table 15.8 System properties determined by analyzing a system model

Property	Description
Impact of faults	Faults can <i>partition</i> a system, i.e., split it into two or more isolated parts such that a node in one part cannot be reached from a node in another part. A system model can be analyzed to find the smallest number of faults which can partition the system.
Resiliency	A system is said to be <i>k-resilient</i> , where <i>k</i> is a constant, if <i>k</i> is the largest number of faults despite which a system can function without disruption.
Latency between two nodes	The minimum latency between two nodes depends on the number of communication links in any path between the nodes, and the minimum latency of each communication link.
Cost of sending information to every node	The cost of this operation depends on topology of the system. In a fully connected system containing <i>n</i> nodes, the cost can be as low as $n - 1$ messages. The cost may be more if the system is not fully connected.

A system model is analyzed to determine useful properties of a system. Table 15.8 describes some such properties. The smallest number of faults that can partition a system can be determined by analyzing the degree of all nodes in the system. If n' is the smallest degree of a node, n' faults may partition a system. Such a system is $(n' - 1)$ -resilient because it can tolerate up to $n - 1$ faults without suffering partitioning. As illustrated in Example 15.1, a similar analysis can be used as a network design technique.

Example 15.1 If it is expected that only one or two sites in a system may suffer faults simultaneously, and faults never occur in communication links, it is adequate to position three units of each resource in three different sites in the system. If communication links can also suffer faults but the total number of faults does not exceed two, three units of each resource must exist and each site must have at least three communication links connected to it. In such a system, a resource becomes unavailable only if three or more faults occur.

To send some information to all nodes in the system, the node where the information originates typically sends it to its neighbors and each node receiving it sends it to its neighbors, and so on. In general, this method requires e messages, where e is the number of edges in the system. However, if the system is fully connected, it is possible to use a simpler protocol in which only the originator node sends messages to its neighbors. This operation would require only $n - 1$ messages.

Computation migration is employed to ensure that nearly equal amounts of computational load are directed at all CPUs in the system. This technique is called *load balancing*.

A distributed system typically grows in size over time through addition of nodes and users. As the size of a system grows, process response times may degrade due to increased loading of resources and services of the OS, and increased overhead of OS control functions. Such degradation obstructs growth of a system, so the performance of a distributed system should be *scalable*, i.e., the performance measured as average response time should not vary much with size of the system. An important scalability technique is to make clusters of hosts discussed in Section 15.4 self-sufficient, so that the network traffic does not grow as more clusters are added to the system. In Chapter 19, we discuss how the technique of *file caching* used in distributed file systems helps satisfy this requirement.

Reliability Availability of resources is ensured using fault tolerance techniques to protect against likely faults. Link and node faults are tolerated by providing redundancy of resources and communication links. If a fault occurs in a network path to a resource or in the resource itself, an application can use another network path to the resource or use another resource. This way, a resource is unavailable only when unforeseen faults occur.

Consistency of data becomes an issue when data is distributed or replicated. When several parts of a distributed data are to be modified, a fault should not put the system in a state in which some parts of the data have been updated but others have not been. A conventional OS uses the technique of *atomic actions* to satisfy this requirement (see Section 7.10.2.2). A distributed OS employs a technique called *two-phase commit protocol* for this purpose (see Section 18.4.3).

Parts of a computation may be performed in different nodes of a system. If a node or link fault occurs during execution of such a computation, the system should assess the damage caused by the fault and judiciously restore some of the subcomputations to previous states recorded in back-ups. This approach is called *recovery*. The system must also deal with uncertainties concerning the cause of a fault. Example 15.2 illustrates these uncertainties.



Fig. 15.15 Resiliency issues in a remote request

Example 15.2 A distributed computation consists of two subcomputations represented by processes P_i and P_j , executing in nodes N_i and N_2 , respectively (see Figure 15.15). Process P_i sends a request to P_j and waits for a response. However, a timeout occurs before it receives a reply. The timeout could be caused by any one of the following possibilities:

Theoretical Issues in Distributed Systems

Time and *state* are two key notions used in an operating system—the OS needs to know the chronological order in which events such as resource requests occur, and it needs to know the states of resources and processes for performing resource allocation and scheduling. In a conventional computer system, presence of a single memory and a single CPU simplifies handling of time and state. Only one event can occur at any time, so the OS knows the chronological order of events implicitly, and it knows states of all processes and resources in the system.

A distributed system contains several computer systems, each with a clock, memory, and one or more CPUs; and communication between computer systems is through messages which incur unpredictable communication delays. Consequently, the distributed OS cannot know the chronological order in which events occur, or the states of resources and processes in all nodes of the system at the same instant of time. Therefore, the key theoretical issues in distributed systems are to evolve practical alternatives to the traditional notions of time and state, develop algorithms to implement these alternatives, and show correctness of these algorithms.

We present the notion of *event precedence* which can be used to discover the chronological order in which *some* events occur in a distributed system. We then discuss two alternatives to the traditional notion of time using the notions of *logical clocks* and *vector clocks*. We also present the notion of a *consistent recording of state* that can be used as an alternative to *global state* of a distributed system in several applications. These alternative notions of time and state are employed in the design of distributed control algorithms and recovery algorithms used in a distributed OS.

16.1 NOTIONS OF TIME AND STATE

Time is the fourth dimension; it indicates when an event occurred. The *state* of an entity is the condition or mode of its being. The state of an entity depends on its features, e.g., the state of a memory cell is the value contained in it. If an entity is composed of other entities, its state contains the states of its component entities. The *global state* of a system is comprised of the states of all entities in the system at a specific instant of time. An OS uses the notions of time and state for performing scheduling of resources and the CPU. It uses time to know when a request event occurred, or to find the *chronological order* in which request events occurred, and it uses the state of a resource to decide whether it can be allocated. A distributed OS also uses these notions in *recovery* to ensure that processes of a distributed computation would be in mutually consistent states when a node containing some of the processes crashes and recovers.

It is easy to handle time and state in a uniprocessor OS. The system has a clock and a single CPU, so the OS can find the times at which processes made their resource requests and use this information to determine their chronological order. However, a typical conventional operating system uses the notion of time only implicitly. It maintains information concerning events in a queue, so the queue shows the chronological order of events. The OS also knows states of processes, and physical and logical resources.

In a distributed system, each node is a computer system with its own clock and a local memory, and nodes are connected by communication links which have unpredictable communication delays. Consequently, a node cannot precisely determine the time at which an event occurred in another node; its perception of the state of a remote process or resource may also be stale. Thus, a distributed OS cannot use the notions of time and state in the same manner as a uniprocessor OS.

In this chapter, we discuss some theoretical concepts in distributed systems and use them to develop practical alternatives to the notions of time and state as used in a uniprocessor system. These alternative notions of time and state are used in Chapter 17 in the design of distributed control algorithms, and in Chapter 18 in the design of recovery algorithms.

16.2 STATES AND EVENTS IN A DISTRIBUTED SYSTEM

16.2.1 Local and Global States

Each entity in a system has its own state. The state of a memory cell is the value contained in it. The state of a CPU is the contents of its PSW and registers. The state of a process is its state tag, state of the memory allocated to it, the CPU state if it is currently scheduled on the CPU, or contents of PCB fields if it is not scheduled on the CPU; and the state of its interprocess communication. The state of an entity is called a *local state*. The *global state* of a system at time instant t is the collection of local states of all entities in it at time t .

We denote the local state of a process P_k at time t as s'_k , where the subscript is omitted if the identity of the process is implicit in the context. We denote the global state of a system at time t as S_t . If a system contains n processes $P_1 \dots P_n$, $S_t \equiv \{s'_1, s'_2, \dots, s'_n\}$.

16.2.2 Events

An *event* could be the sending or receiving of a message over a *channel*, which is an interprocess communication path, or some other happening that does not involve a message. The state of a process changes when an event occurs in it. We represent an event as follows:

$$(process\ id, old\ state, new\ state, event\ description, channel, message)$$

where *channel* and *message* are written as ‘ ’ if the event does not involve sending or receiving of a message. An event $e_i \equiv (P_k, s, s', send, c, m)$ can occur only when process P_k is in state s . The event is the sending of a message m over a channel c . When this event occurs, process P_k enters the new state s' (see Figure 16.1).

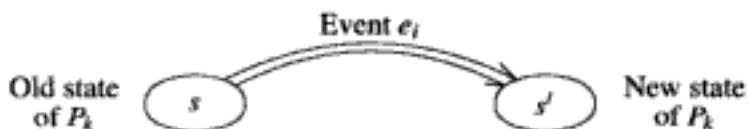


Fig. 16.1 Change of state in process P_k on occurrence of event $\langle P_k, s, s', send, c, m \rangle$

Certain events can occur only when some conditions are met, e.g., a receive event for message m on channel c can occur only if channel c contains message m . Thus, only some events may be feasible in the current state of a process.

16.3 TIME, CLOCKS AND EVENT PRECEDENCES

Let a *global clock* be an abstract clock that can be accessed from different sites of a distributed system with identical results. If processes in two nodes of a distributed system perform the system call ‘give current time’ at the same time instant, they would obtain identical time values. If they perform these system calls δ time units apart, they would obtain time values that differ by exactly δ time units. A global clock cannot be implemented in practice due to communication delays. Requests for current time made in two different nodes at the same time instant would face different communication delays to reach the site where the clock is maintained. Consequently, they would be given different time values. Similarly, requests that are made δ time units apart may get time values that do not differ by exactly δ time units.

Since a global clock cannot be implemented, we can explore an alternative arrangement that uses a clock in each process. Such a clock is called a *local clock*. The local clock of a process would be accessed whenever the process performs a

'give current time' system call. To implement a practical time-keeping service using this idea, local clocks should be reasonably well-synchronized. Section 16.3.2 discusses how it can be achieved using the notion of event precedence.

16.3.1 Event Precedence

The notation $e_1 \rightarrow e_2$ is used to indicate that event e_1 precedes event e_2 in time, i.e., event e_1 occurred before event e_2 . *Event ordering* implies arranging a set of events in a sequence such that each event in the sequence precedes the next one. In essence, it implies determining the order in which events have occurred in a system. A *total order* with respect to the precedes relation ' \rightarrow ' is said to exist if all events that can occur in a system can be ordered. A *partial order* implies that some events can be ordered but not all events can be ordered—to be precise, at least two events exist that cannot be ordered.

Table 16.1 Rules for ordering of events in a distributed system

Category	Description of rule
Events within a process	The OS performs event handling, so it knows the order in which events occur within a process.
Events in different processes	In a <i>causal relationship</i> , i.e., a cause-and-effect relationship, an event that corresponds to the cause precedes an event in another process that corresponds to the effect.
Transitive precedence	The precedence relation is transitive, i.e., $e_1 \rightarrow e_2$ and $e_2 \rightarrow e_3$ implies $e_1 \rightarrow e_3$.

Table 16.1 summarizes the fundamental rules used to perform event ordering. These rules can be explained as follows: The OS can readily determine precedence between events occurring within the same process. Events like execution of a 'send $P_3, <message\ m_i>$ ' event in a process P_2 and a receive event in P_3 that receives message m_i , have a *causal relationship*, i.e., a cause-and-effect relationship. Consequently, the send event in process P_2 precedes the receive event in process P_3 . The ' \rightarrow ' relation is transitive in nature, hence $e_1 \rightarrow e_3$ if $e_1 \rightarrow e_2$ and $e_2 \rightarrow e_3$. This property can be used to determine precedences between some events that are neither causally related nor occur within the same process. For example, an event e_i preceding the send event for message m_i in P_2 precedes an event e_j that follows the receive event for message m_i in P_3 , because e_i precedes the send event, the send event precedes the receive event, and the receive event precedes event e_j .

Using the rules of Table 16.1, precedence between any two events e_i and e_j can be classified as follows:

- e_i precedes e_j : If events e_k and e_l exist such that $e_k \rightarrow e_l$, $e_l \rightarrow e_k$ or $e_i \equiv e_k$, and $e_l \rightarrow e_j$ or $e_l \equiv e_j$.

- e_i follows e_j : If events e_g and e_h exist such that $e_g \rightarrow e_h$, $e_j \rightarrow e_g$ or $e_j \equiv e_g$, and $e_h \rightarrow e_i$ or $e_h \equiv e_i$.
- e_i is concurrent with e_j : This is the case if e_i neither precedes nor follows e_j .

A *timing diagram* is a plot of the activities of different processes against time—processes are marked along the vertical axis in the plot, and time is marked along the horizontal axis. Example 16.1 demonstrates use of a timing diagram in determining event precedences using transitivity of the precedes relation. It also illustrates why a total order over events does not exist in a distributed system.

Example 16.1 Figure 16.2 shows activities in processes P_1 and P_2 . Event e_{23} is a ‘send’ event while e_{12} is a ‘receive’ event for message m_1 . Hence $e_{23} \rightarrow e_{12}$. The transitive nature of ‘ \rightarrow ’ leads to the precedence relations $e_{22} \rightarrow e_{12}$ and $e_{21} \rightarrow e_{12}$. Transitivity also yields $e_{22} \rightarrow e_{13}$ and $e_{21} \rightarrow e_{13}$. Event e_{11} is concurrent with events e_{21} and e_{22} . It is also concurrent with events e_{23} and e_{24} !

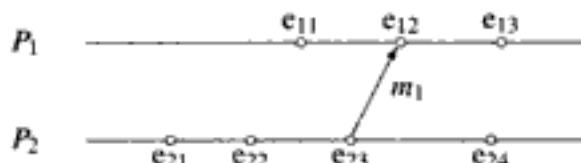


Fig. 16.2 Event precedence via timing diagram

16.3.2 Logical Clocks

An OS needs a practical method of event ordering for purposes related to scheduling and resource allocation. The method should be efficient, so it should perform event ordering directly, instead of working through causal relationships and transitivity. It should also provide a total order over events so that the OS can provide FCFS service to resource requests. Such an order can be obtained by

- Incorporating event precedences in the event order
- Arbitrarily ordering events that are concurrent, e.g., the events e_{11} and e_{21} in Figure 16.2.

Time-stamping of events provides a direct method of event ordering. Each process has a local clock that is accessible only to itself. The *time-stamp* of an event is its occurrence time according to the local clock of the process. Let $ts(e_i)$ represent the time-stamp of event e_i . Event ordering is performed in accordance with the time-stamps of events, i.e., $\forall e_i, e_j : e_i \rightarrow e_j$ if $ts(e_i) < ts(e_j)$ and $e_j \rightarrow e_i$ if $ts(e_i) > ts(e_j)$. However, local clocks in different processes may show different times due to clock drift, which would affect reliability of time-stamp based event ordering. For example, if event e_i occurred before event e_j , $ts(e_i)$ should be $< ts(e_j)$; however, if the clock at the process where event e_i occurred is running faster than the clock at

Time-stamping through vector clocks has two important properties: Every event has a unique time-stamp due to rules R3 and R4, and $vts(e_i) < vts(e_j)$ if and only if $e_i \rightarrow e_j$. The next example illustrates these properties.

Example 16.3 Figure 16.4 shows synchronization of vector clocks for the system of Figure 16.3. The vector time-stamp after the occurrence of an event is shown below it. When message m_1 is received, $C_2[2]$ is incremented by 1 and $C_2[1]$ is updated to 10. Analogously, when message m_2 is received by process P_3 , $C_3[3]$ is incremented by 1 and $C_3[1]$ and $C_3[2]$ are updated. Events e_{32} and e_{23} are concurrent events because $vts(e_{32}) \leq vts(e_{23})$ and $vts(e_{32}) \geq vts(e_{23})$.

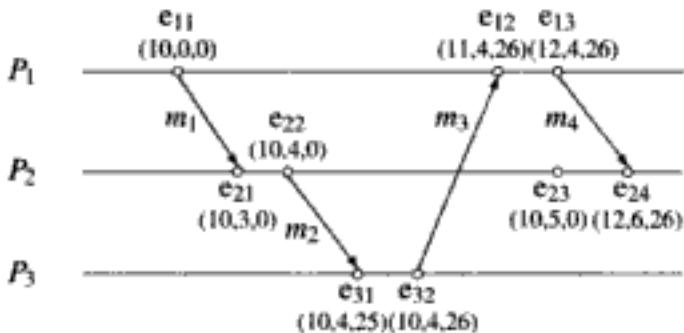


Fig. 16.4 Synchronization of vector clocks

The property that $vts(e_i) < vts(e_j)$ if and only if $e_i \rightarrow e_j$ implies that vector clocks do not provide a total order over events. Total order can be obtained using a pair $pvts(e_i) \equiv (\text{local time}, \text{process id})$ and the following relation:

$$\begin{aligned} e_i \text{ precedes } e_j \text{ iff } (i) \quad &pvts(e_i).\text{local time} < pvts(e_j).\text{local time}, \text{ or} \\ (ii) \quad &e_i, e_j \text{ are concurrent events and} \\ &pvts(e_i).\text{process id} < pvts(e_j).\text{process id} \end{aligned} \quad (16.2)$$

where $pvts(e_i).\text{local time}$ and $pvts(e_i).\text{process id}$ are the local vector time and process id in $pvts(e_i)$, respectively.

16.4 RECORDING THE STATE OF A DISTRIBUTED SYSTEM

As discussed in Section 16.2.1, the global state of a distributed system at a time instant t is the collection of local states of all entities in the system at timer. It is not possible to get all nodes to record their states at the same time instant because local clocks are not perfectly synchronized. Any other collection of local states may be inconsistent. Consider the distributed system shown in Figure 16.5. Let a banking application have a process P_1 in node N_1 that deducts 100 dollars from account A and sends a message to process P_2 in node N_2 to add this amount to account B. The recorded states of nodes N_1 and N_2 would be inconsistent if the balance in account A is recorded before the transfer and that in account B is recorded after the transfer.

Table 16.2 Local states of processes

Process	Description of recorded state
P_1	No messages have been sent. Message m_{21} has been received.
P_2	Messages m_{21} and m_{23} have been sent. No messages have been received.
P_3	No messages have been sent. Message m_{43} has been received.
P_4	No messages have been sent. No messages have been received.

The curve in Figure 16.7 is the cut of the distributed computation representing the recorded state shown in Table 16.2. The term 'a cut is taken' means that a collection of local states is recorded. An event that had occurred in a process before the state of the process was recorded is said to occur 'to the left of the cut' in the timing diagram. Such an event belongs in the *past of the cut*. An event that would occur in a process after the state of the process was recorded is said to occur 'to the right of the cut' in the timing diagram. Such an event belongs to the *future of the cut*. A cut represents a consistent state recording of a system if the states of each pair of processes satisfy Definition 16.1.

State of a channel The state of a channel Ch_{ij} is the set of messages contained in Ch_{ij} , i.e., the messages sent by process P_i that are not yet received by process P_j . We use the following notation to determine the state of a channel Ch_{ij} :

- Recorded_sent_{ij}*: the set of messages recorded as sent over channel Ch_{ij} in the state of P_i
- Recorded_recd_{ij}*: the set of messages recorded as received over channel Ch_{ij} in the state of P_j

$Recorded_sent_{ij} = Recorded_recd_{ij}$ implies that all messages sent by P_i have been received by P_j . Hence the channel is empty. $Recorded_sent_{ij} - Recorded_recd_{ij} \neq \emptyset$, where '-' represents the set difference operator, implies that some messages sent by P_i have not been received by P_j . These messages are still contained in channel Ch_{ij} . $Recorded_recd_{ij} - Recorded_sent_{ij} \neq \emptyset$ implies that process P_j has recorded as received at least one message that is not recorded as sent by process P_i . This situation indicates inconsistency of the recorded local states of P_i and P_j according to Definition 16.1.

A cut may intersect with a message m_k sent by process P_i to process P_j over channel Ch_{ij} . Three such possibilities are:

- *No intersection*: The message send and receive events are either both located to the left of the cut, or both located to the right of the cut. In either case, the message did not exist in channel Ch_{ij} when the cut was taken.

16.4.2 An Algorithm for Consistent State Recording

This section describes the state recording algorithm by Chandy and Lamport (1985). The algorithm makes the following assumptions:

1. Channels are unidirectional
2. Channels have unbounded capacities to hold messages
3. Channels are FIFO.

The assumption of FIFO channels implies that messages received by a destination process must be the first few messages sent by a sender process, and messages contained in a channel must be the last few messages sent by a process.

To initiate a state recording, a process records its own state and sends a state recording request called a *marker* on every outgoing channel. When a process receives a marker, it records the state of the channel over which it received the marker. If this is the first marker it received, it also records its own state and sends a marker on every outgoing channel. We use the following notation to discuss how the state of a channel is determined:

- $Received_{ij}$: the set of messages received by process P_j on channel Ch_{ij} before it received the marker on channel Ch_{ij} .
 $Recorded_recd_{ij}$: the set of messages recorded as received over channel Ch_{ij} in the state of process P_j .

Algorithm 16.2 (Chandy–Lamport algorithm for consistent state recording)

1. When a process P_i initiates the state recording: P_i records its own state. It then sends a marker on each outgoing channel connected to it.
2. When process P_j receives a marker over an incoming channel Ch_{ij} : Process P_j performs the following actions:
 - (a) If this is the first marker P_j received, then
 - (i) Record its own state.
 - (ii) Record the state of channel Ch_{ij} as empty.
 - (iii) Send a marker on each outgoing channel connected to it.
 - (b) Record the state of channel Ch_{ij} to contain the messages ($Received_{ij} - Recorded_recd_{ij}$).

Let a process P_i send messages m_1, m_2, \dots, m_n on channel Ch_{ij} before recording its own state and sending a marker on Ch_{ij} . Let process P_j have two incoming channels Ch_{ij} and Ch_{kj} . If the marker on channel Ch_{ij} is the first marker P_j received, it would record its own state, which would contain $Recorded_recd_{ij}$ and $Recorded_recd_{kj}$. P_j would also record the state of Ch_{ij} . Because channels are FIFO, process P_j would have received the marker after receiving messages m_1, m_2, \dots, m_n on Ch_{ij} , so it is correct to record the state of channel Ch_{ij} as *empty*.

Let process P_j not receive any more messages on channel Ch_{kj} before it received a marker on Ch_{kj} . The state of channel Ch_{kj} would be recorded as $Received_{kj}$.

Recorded_recd_{kj}, i.e., as empty. If P_j had received two more messages m_{k_1} and m_{k_2} on Ch_{kj} before it received the marker, $Received_{kj} = Recorded_recd_{kj} \cup \{m_{k_1}, m_{k_2}\}$. The state of channel Ch_{kj} would now be recorded as containing the set of messages $Received_{kj} - Recorded_recd_{kj}$ i.e., $\{m_{k_1}, m_{k_2}\}$.

Example 16.5 illustrates operation of the Chandy–Lamport algorithm. Note that rules of the algorithm are executed atomically, i.e., as indivisible operations.

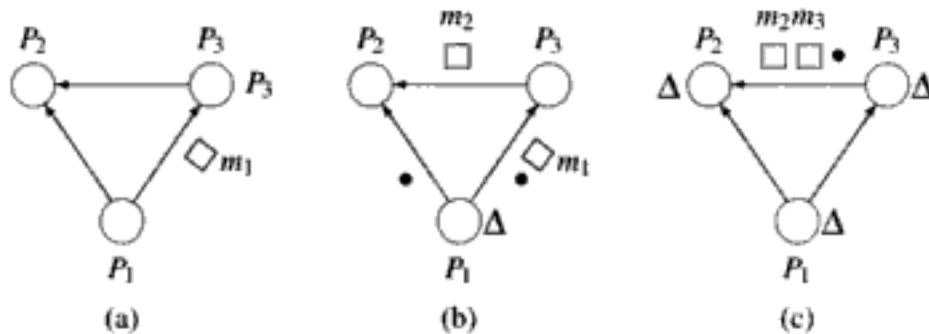


Fig. 16.9 Example of the Chandy–Lamport algorithm: System at times 0, 2^+ and 5^+

Example 16.5 Figure 16.9(a) shows a distributed system at time 0. Process P_1 has sent message m_1 to P_3 . The message currently exists in Ch_{13} . At time 1, process P_3 sends message m_2 to process P_2 . At time 2, P_1 decides to record the state of the system, so it records its own state and sends markers on its outgoing channels. Figure 16.9(b) shows the situation at time 2^+ . Message m_1 is still in channel Ch_{13} and m_2 is in Ch_{32} . The bullets indicate markers. The symbol Δ indicates that the state of a process has been recorded.

Process P_2 receives the marker on Ch_{12} at time 3, records its own state and records the state of Ch_{12} as empty. Process P_3 sends message m_3 to process P_2 at time 4 and receives the marker on Ch_{13} at time 5. It now records its own state, records the state of Ch_{13} as empty, and sends a marker on Ch_{32} . Figure 16.9(c) shows the situation at time 5^+ . States of all processes have been recorded. States of channels Ch_{12} and Ch_{13} have also been recorded; however, the state of Ch_{32} is yet to be recorded.

When the marker on Ch_{32} reaches process P_2 , P_2 will record the state of Ch_{32} according to step 2(b) of Algorithm 16.2. It is recorded as messages $\{m_2, m_3\}$ as these messages are in $Received_{32}$ but not in $Recorded_recd_{32}$. Table 16.3 shows the state recording of the system.

Properties of the recorded state Let t_b and t_e be the time instants when the state recording of system S begins and ends. Let RS be the recorded state of the system. One would expect that system S would have been in the state RS at some time instant t_l such that $t_b < t_l < t_e$. However, this may not be so! That is, the recorded state RS may not match any global state of the system. Example 16.6 illustrates this fact.

Example 16.6 Figure 16.7 shows the timing diagram of the distributed system of Figure 16.6. Let P_4 initiate state recording at time instant t_1 . The timing diagram of Figure 16.10 shows how the markers are sent and received by processes during state recording. The markers are shown as dotted arrows.

Table 16.3 Recorded states of processes and channels in Fig. 16.9

Entity	Description of recorded state
P_1	Message m_1 has been sent. No messages have been received.
P_2	No message have been sent or received.
P_3	Messages m_2 and m_3 have been sent. Message m_1 has been received.
Ch_{12}	Empty
Ch_{13}	Empty
Ch_{23}	Contains messages $\{m_2, m_3\}$.

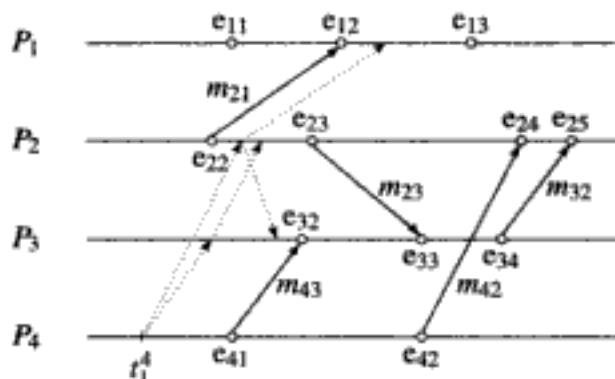
**Fig. 16.10** State recording of the system of Figs. 16.6, 16.7

Table 16.4 shows channel and process states recorded by the Chandy-Lamport algorithm. Only message m_{21} is recorded as sent by P_2 and received by P_1 . No other messages are recorded as sent or received. However, if the timing diagram of Figure 16.7 is to scale, it is clear that the system never existed in a state in which message m_{21} had been sent and received but no other messages had been sent. This is because the message-send and message-receive events e_{23} , e_{32} and e_{41} occurred before event e_{12} , the message-receive event for message m_{21} . Thus any global state that recorded message m_{21} as received, would have also recorded message m_{43} as sent and received, and message m_{23} as sent.

Table 16.4 A recorded state that does not match any global state

Entity*	Description of recorded state
P_1	No messages have been sent. Message m_{21} has been received.
P_2	Message m_{21} has been sent. No messages have been received.
P_3	No messages have been sent or received.
P_4	No messages have been sent or received.

*: States of all channels are recorded as empty

- (a) $RP(i,k) \cap RP(j,k) \neq \emptyset$
 (b) $\exists P_g \in SP(i,k) \ni P_j \in SP(g,k)$
 (c) $P_j \in SP(i,k)$
 (d) $P_j \in RP(i,k)$ but $P_j \notin RP(i,k-1)$
 (e) $P_j \in SP(i,k)$ and $P_i \in SP(j,k)$
 (f) $P_j \in RP(i,k)$ but $P_j \notin RP(i,k-1)$ and P_i has not received any message from any process after the time it sent a message to P_j .
4. Relation (16.1) imposes a total order even if events can be only partially ordered using causal relationships. Give an example of a system showing such events. Comment on the advantages and drawbacks of using relation (16.1).
 5. Instead of using relation (16.2) to obtain a total order using vector time-stamps, it is proposed to use the following relation:
- $$e_i \text{ precedes } e_j \text{ iff } \begin{array}{l} (i) \text{ } pvt(e_i).local\ time < pvt(e_j).local\ time, \text{ or} \\ (ii) \text{ } pvt(e_i).local\ time = pvt(e_j).local\ time \text{ and} \\ \quad pvt(e_i).process\ id < pvt(e_j).process\ id \end{array}$$
- Comment on correctness of this proposal.
6. A and B are users of a distributed operating system using logical clocks. User A causes event e_A and then phones user B. After the phone conversation, user B causes event e_B . However the time-stamp of e_B is smaller than the time-stamp of e_A . Explain why this may be so (Adapted from Lamport [1978]).
 7. t_i and t_j are time-stamps of events e_i and e_j .
 - (a) Give an example of a system in which $t_i < t_j$ when logical clocks are used but $t_i \not< t_j$ when vector clocks are used.
 - (b) If $t_i < t_j$ when vector clocks are used, show that t_i must be $< t_j$ when logical clocks are used.
 - (c) If $t_i < t_j$ when logical clocks are used, show that t_i cannot be $> t_j$ when vector clocks are used.
 8. Vector time-stamps of concurrent events e_i and e_j are such that $vts(e_i)[k] < vts(e_j)[k]$. Show that events e_i and e_j are concurrent if $vts(e_i)[g] = vts(e_j)[g]$ for all $g \neq k$ and $vts(e_i)[k] > vts(e_j)[k]$.
 9. Explain, with the help of an example, why the Chandy-Lamport algorithm requires channels to be FIFO.
 10. A transit-less state of a system is a state in which no messages are in transit. (See Table 16.4 for an example.) Give an example of a system in which all states recorded by the Chandy-Lamport algorithm are necessarily transit-less.
 11. A system consists of processes P_i, P_j and channels Ch_{ij} and Ch_{ji} . Each process sends a message to the other process every δ seconds, the first message being at time $t = 0$. Every message requires σ seconds to reach P_j . Prove that if $\delta < \sigma$, the state recording initiated by P_i using the Chandy-Lamport algorithm cannot be transit-less.
 12. Give an example of a system in which the state recorded by the Chandy-Lamport algorithm is one of the states in which the system existed sometime during the execution of the algorithm.
 13. What will be the state recording in Example 16.6, if the state recording request

11. Tel, G. (2000): *Introduction to Distributed Algorithms, Second edition*, Cambridge University Press, Cambridge.

Distributed Control Algorithms

A distributed operating system performs several control functions. Of these control functions, the *mutual exclusion* and *deadlock handling* functions are similar to those performed in a conventional OS. The *scheduling* function performs *load balancing* to ensure that computational loads in all nodes of the system are comparable. The *election* function elects one among a group of processes as the coordinator. The *termination detection* function checks whether processes of a distributed computation, which may be operating in different nodes of the system, have all completed their tasks.

To respond speedily and reliably to events occurring in the system, a distributed operating system performs a control function using a *distributed control algorithm*, whose actions are performed in several nodes of the distributed system. Distributed control algorithms avoid using the global state of a system. Instead, they depend on local states of different nodes, and use interprocess messages to query the states and make decisions. Their correctness depends on the arrangement of local states and interprocess messages used for arriving at correct decisions, and for avoiding wrong decisions. These two aspects of correctness are called *liveness* and *safety*, respectively.

We present distributed control algorithms for the different control functions and discuss their properties such as overhead and impact on system performance.

17.1 OPERATION OF DISTRIBUTED CONTROL ALGORITHMS

A distributed operating system implements a control function through a *distributed control algorithm*, whose actions are performed in several nodes of the system and whose data is also spread across several nodes. This approach has the following advantages over a centralized implementation of control functions:

- The delays and overhead involved in collecting the global state of a system would be avoided.

- The control function would be able to respond speedily to events in different nodes of the system.
- Failure of a single node would not cripple the control function.

Table 17.1 Overview of control functions in a distributed OS

Function	Description
Mutual exclusion	Implement a critical section (CS) for a data item d_s for use by processes in a distributed system. It involves synchronization of processes operating in different nodes of the system so that at most one process is in a CS for d_s at any time.
Deadlock handling	Prevent or detect deadlocks that arise due to resource sharing within and across nodes of a distributed system.
Scheduling	Perform <i>load balancing</i> to ensure that computational loads in different nodes of a distributed system are comparable. It involves transferring processes from heavily loaded nodes to lightly loaded nodes.
Termination detection	Processes of a distributed computation may operate in several nodes of a distributed system. Termination detection is the act of determining whether such a computation has completed its operation. It involves checking whether any process of the computation is active or whether any interprocess message is in transit between processes of the computation.
Election	A <i>coordinator</i> (also called a <i>leader</i> process) is the one that performs some privileged function like resource allocation. An election is performed when a coordinator fails or is terminated. It selects one process to become the new coordinator and informs the identity of the new coordinator to all other processes.

A distributed control algorithm provides a service whose clients include both user applications and the kernel. Table 17.1 describes control functions in a distributed OS. *Mutual exclusion* and *election* are services provided to user processes, *deadlock handling* and *scheduling* are services offered to the kernel, while the *termination detection* service may be used by both user processes and the kernel. In OS literature, names of these functions are generally prefixed with the word 'distributed' to indicate that the functions are performed in a distributed manner. Note that fault tolerance and recovery issues are not discussed here; they are discussed in Chapter 18.

A distributed control algorithm operates in parallel with its clients, so that it can respond readily to events related to its service. The following terminology is used to distinguish between actions of a client and those of a control algorithm.

- *Basic computation:* Operation of a client constitutes a *basic computation*. A basic computation may involve processes in one or more nodes of the system. The messages exchanged by these processes are called *basic messages*.
- *Control computation:* Operation of a control algorithm constitutes a *control computation*. Messages exchanged by processes of a control computation are

called *control messages*.

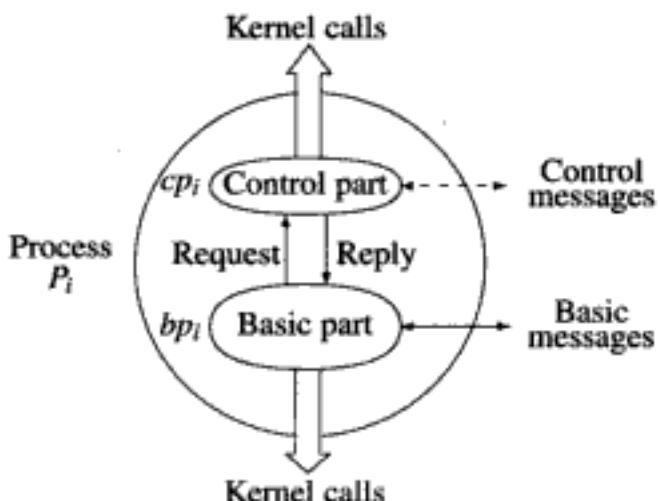


Fig. 17.1 Basic and control parts of a process P_i

To understand operation of a distributed control algorithm, we visualize each process to consist of two parts that operate in parallel—a basic part and a control part. Figure 17.1 illustrates the two parts of a process P_i . The *basic part* of a process participates in a basic computation. It exchanges basic messages with basic parts of other processes. When it requires a service offered by a control algorithm, it makes a request to the control part of the process. All other requests are made directly to the kernel. The *control part* of a process participates in a control computation. It exchanges control messages with control parts of other processes, and may interact with the kernel to implement its part in the control function. The basic part of a process may be blocked due to a resource request; however, the control part of a process is never blocked—this feature enables it to respond to events related to its service in a timely manner.

Example 17.1 A distributed application consists of four processes P_1-P_4 . Let process P_2 be currently in a CS for shared data d_s . When process P_1 wishes to enter a CS for d_s , bp_1 makes a request to cp_1 , which is a part of some distributed mutual exclusion algorithm discussed later in Section 17.3. To decide whether P_1 can be allowed to enter a CS for d_s , cp_1 exchanges messages with cp_2 , cp_3 and cp_4 . From their replies, it realizes that some other process is currently in a CS for d_s , so it makes a kernel call to block bp_1 . Note that cp_2 participates in this decision even while bp_2 was executing in a CS. When process P_2 wishes to exit the CS, bp_2 makes a request to cp_2 , which interacts with control parts of other processes and decides that process P_1 may enter a CS for d_s . Accordingly, cp_1 makes a kernel call to activate bp_1 .

17.2 CORRECTNESS OF DISTRIBUTED CONTROL ALGORITHMS

Processes of a distributed control algorithm exchange control data and coordinate their actions through control messages. However, message communication incurs

delays, so the data used by the algorithm may become stale and inconsistent, and the algorithm may either miss performing correct actions or perform wrong actions. Accordingly, correctness of a distributed control algorithm has two facets:

- *Liveness*: Liveness implies that an algorithm will *eventually* perform correct actions, i.e., perform them without indefinite delays.
- *Safety*: Safety implies that the algorithm does not perform wrong actions.

Lack of liveness implies that an algorithm would fail to perform correct actions. For example, a distributed mutual exclusion algorithm might fail to satisfy the *progress* and *bounded wait* properties of Section 9.2.1, or a deadlock detection algorithm might not be able to detect a deadlock that exists in the system. Note that the amount of time needed to perform a correct action is immaterial for the liveness property; the action must *eventually* be performed. Lack of safety implies that an algorithm may perform wrong actions like permitting more than one process to be in CS at the same time. Table 17.2 summarizes the liveness and safety properties of some distributed control algorithms.

Table 17.2 Liveness and safety of distributed control algorithms

Algorithm	Liveness	Safety
Mutual exclusion	(1) If a CS is free and some processes have requested entry to it, one of them will enter it in finite time. (2) A process requesting entry to a CS will enter it in finite time.	At most one process will be in a CS at any time.
Deadlock handling	If a deadlock arises, it will be detected in finite time.	A deadlock will not be declared unless it has occurred.
Termination detection	Termination of a distributed computation will be detected within a finite time.	Termination will not be declared unless it has occurred.
Election	A new coordinator will be elected in a finite time.	Exactly one process will be elected coordinator.

Assuming a set of distinct actions in a distributed control algorithm and a set of distinct conditions in it, we can specify a distributed control algorithm as a set of rules of the form $\langle \text{condition} \rangle : \langle \text{action} \rangle$ where $\langle \text{action} \rangle$ is an action the algorithm should perform if and only if $\langle \text{condition} \rangle$ is true. Using the notation \mapsto for the words 'eventually leads to', the implication of defining liveness and safety in this manner is that

- *Liveness*: For all rules, $\langle \text{condition} \rangle \mapsto \langle \text{action} \rangle$, i.e., $\langle \text{action} \rangle$ will be eventually performed if $\langle \text{condition} \rangle$ holds.

Example 17.2 Figure 17.4(a) shows the situation in the system of Figure 17.3 after the requests made by P_4 and P_1 have reached P_5 , which is P_{holder} (see Steps 1 and 2 of Algorithm 17.3). When process P_5 exits its CS, it removes P_3 from its local queue, passes the token to P_3 and reverses the edge (P_3, P_5) . P_5 now sends a request to P_3 since its local queue is not empty (see Step 3(d)). P_3 performs similar actions (see Step 4), which result in sending the token to process P_4 , reversal of the edge (P_4, P_3) and sending of a request by P_3 to P_4 .

Figure 17.4(b) shows the resulting abstract inverted tree. P_4 now enters its CS. After P_4 completes the CS, the token is transferred to process P_1 via P_3 and P_5 in an analogous manner, which enables P_1 to enter its CS. Note that this would not have been possible if Step 3(d) did not exist in the algorithm.

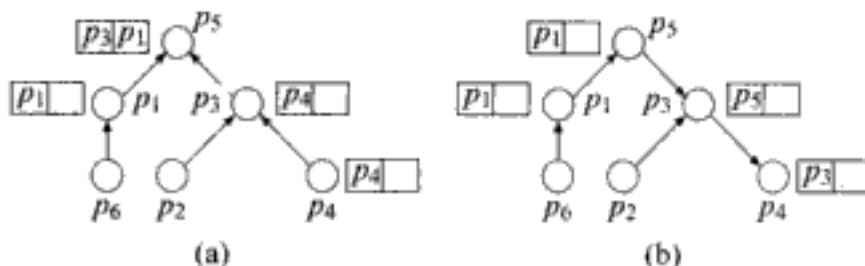


Fig. 17.4 An example of Raymond's algorithm

17.4 DISTRIBUTED DEADLOCK HANDLING

The deadlock detection, prevention and avoidance approaches discussed in Section 11.3 make use of state information. This section illustrates problems in extending these approaches to a distributed system, and then describes distributed deadlock detection and distributed deadlock prevention approaches. No special techniques for distributed deadlock avoidance have been discussed in OS literature. For simplicity, the discussion in this Section is restricted to the single-instance-single-request (SISR) model of resource allocation (see Section 11.3). Thus, each resource class contains only one resource instance and each request is only for an instance of one resource class.

17.4.1 Problems in Centralized Deadlock Detection

Distributed applications may use resources located in several nodes of the system. Deadlocks involving such applications could be detected by collecting the wait-for graphs (WFG's) of all nodes at a central node, superimposing them to form a merged WFG, and employing a conventional deadlock detection algorithm to check for deadlocks. However, this scheme has a weakness. It may obtain WFGs of individual nodes at different instants of time, so the merged WFG may represent a misleading view of wait-for relationships in the system. This could lead to detection of *phantom deadlocks*, which is a violation of the safety property in deadlock detection. Example 17.3 illustrates such a situation.

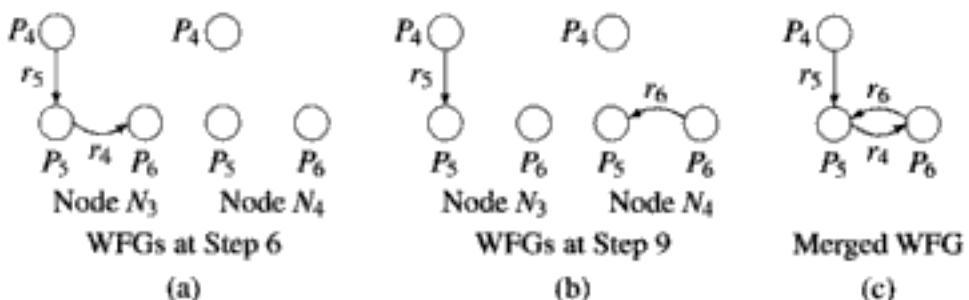


Fig. 17.5 Phantom deadlock in Example 17.3; Node WFG's at steps 6, 9 and the merged WFG

Example 17.3 The sequence of events in a system containing three processes P_4 , P_5 and P_6 is as follows:

1. Process P_5 requests and obtains resource r_5 in node N_3 .
 2. Process P_6 requests and obtains resource r_4 in node N_3 .
 3. Process P_5 requests and obtains resource r_6 in node N_4 .
 4. Process P_4 requests resource r_5 in node N_3 .
 5. Process P_5 requests resource r_4 in node N_3 .
 6. Node N_3 sends its local WFG to the coordinator node.
 7. Process P_6 releases resource r_4 in node N_3 .
 8. Process P_6 requests resource r_6 in node N_4 .
 9. Node N_4 sends its local WFG to the coordinator node.

Figures 17.5(a) and (b) show WFG's of the nodes at Steps 6 and 9, respectively. It can be seen that no deadlock exists in the system at any of these times. However, the merged WFG is constructed by superimposing the WFG of node N_3 taken at Step 6 and WFG of node N_4 taken at Step 9 (see Figure 17.5(c)), so it contains a cycle $\{P_5, P_6\}$ and the coordinator detects a phantom deadlock.

17.4.2 Distributed Deadlock Detection

In the distributed deadlock detection approach, every node in the system has the ability to detect a deadlock. We discuss two such algorithms.

Diffusion computation-based deadlock detection The *diffusion computation* is a distributed control computation which has the liveness and safety properties. (Dijkstra and Scholten, who proposed it in 1980, called it the diffusing computation.) The diffusion computation contains two phases—a diffusion phase and an information collection phase. In the diffusion phase, the computation originates in one node and spreads to other nodes through control messages called *queries* that are sent along all edges in the system. A node may receive more than one query if it has many in-edges. The first query received by a node is called an *engaging query*, while queries received later are called *non-engaging queries*. When a node receives an engaging query, it sends queries along all its out-edges. If it receives a non-engaging query subsequently, it does not send out any queries because it would have already sent queries when it received the engaging query. In the information collection phase, each node in the system sends a reply to every query received by it. The reply to an

time it received the engaging query are not needed in this system, which is an SISR system; these conditions are necessary to detect deadlocks in MISR systems.

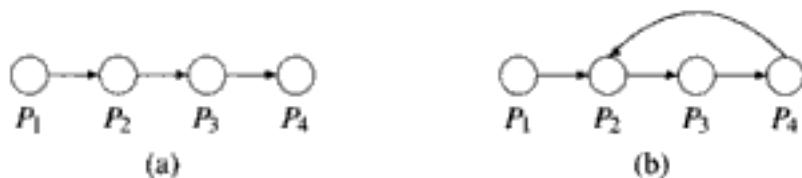


Fig. 17.6 System for illustrating diffusion computation based distributed deadlock detection

Mitchell–Merritt algorithm This algorithm belongs to the class of algorithms called *edge chasing* algorithms—a control message is sent over an edge in the WFG to facilitate detection of cycles in the WFG. Each process is assigned two numerical labels called a *public label* and a *private label*. The public and private labels of a process are identical when the process is created. These labels change when a process gets blocked on a resource. The public label of a process also changes when it waits for a process having a larger public label. A wait-for edge that has a specific relation between the public and private labels of its start and end processes signals a deadlock.

Name of rule	Pre-condition	After applying the rule
Block	$\frac{u}{v} \rightarrow x$	$\frac{z}{z} \rightarrow x$
Activate	$\frac{}{} \rightarrow \frac{}{}$	$\frac{}{} \rightarrow \frac{}{}$
Transmit	$\frac{u}{v} \xrightarrow{u < w} \frac{w}{v}$	$\frac{w}{v} \rightarrow \frac{w}{v}$
Detect	$\frac{u}{u} \rightarrow \frac{u}{u}$	$\frac{u}{u} \wedge \frac{u}{u}$

Fig. 17.7 Rules of Mitchell–Merritt algorithm

Figure 17.7 illustrates rules of the Mitchell–Merritt algorithm. A process is represented as $\frac{u}{v}$ where u and v are its public and private labels, respectively. Figure 17.7 illustrates rules of the Mitchell–Merritt algorithm. A rule is applied when the public and private labels of processes at the start and end of a wait-for edge satisfy the pre-condition. It changes the labels of the processes as shown to the right of ' \Rightarrow '. Details of the four rules are as follows:

1. *Block*: The public and private labels of a process are changed to a value z when it becomes blocked due to a resource request. The value z is generated through the statement $z := inc(u, x)$, where u is the public label of the process, x is the public label of the process for which it waits, and function *inc* generates a

may wait for a younger process, but a younger process cannot wait for an older process.

- **Wound-or-wait:** If P_{req} is younger than P_{holder} , it is allowed to wait for the resource held by P_{holder} ; otherwise, P_{holder} is killed and the requested resource is allocated to P_{req} .

In both approaches, the younger process is aborted and has to be re-initiated sometime in future. To avoid starvation due to repeated aborts, a process may be permitted to retain its old time-stamp when it is re-initiated. The wait-or-die scheme may be preferred in practice because it does not involve preemption of a resource, whereas the wound-or-wait scheme does.

17.5 DISTRIBUTED SCHEDULING ALGORITHMS

Both system performance and computation speed-ups in applications would be adversely affected if computational loads in the nodes of a distributed system are unequal. A distributed scheduling algorithm balances computational loads in the nodes by transferring some processes from a heavily loaded node to a lightly loaded one. Figure 17.8 illustrates this technique, which is called *process migration*. Process P_i is created in node N_1 at time $t = 0$. At time t_i , the scheduling function decides to transfer the process to node N_2 , so operation of the process is halted in node N_1 and the kernel starts transferring its state to node N_2 . At time t_j , the transfer of state is complete and the process starts operating in node N_2 .

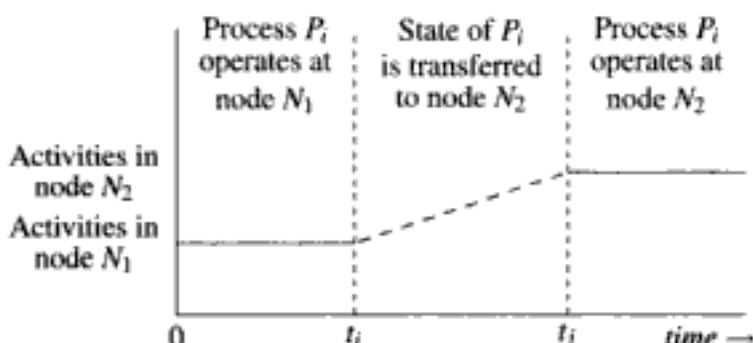


Fig. 17.8 Migration of process P_i from node N_1 to node N_2

To perform load balancing through process migration, a scheduling algorithm needs to measure the computational load in a node, and apply a *threshold* to decide whether it is heavily loaded or lightly loaded. At appropriate times, it transfers processes from heavily loaded nodes to lightly loaded nodes. These nodes are called *sender nodes* and *receiver nodes*, respectively. CPU utilization is a direct indicator of the computational load serviced in a node; however, monitoring of CPU utilization incurs high execution overhead. Hence operating systems prefer to use the number of processes in a node or the length of the *ready queue* of processes, as measures of

computational loads. These measures possess a good correlation with the average response time in a node, and their use incurs a low overhead.

Actual migration of a process can be performed in two ways. *Preemptive migration* involves suspending a process, recording its state, transferring it to another node (see Figure 17.8), and resuming operation of the process in the new node; it requires extensive kernel support. In *non-preemptive migration*, the load balancing decision is taken while creating a new process. If the node in which a 'create process' call is performed is heavily loaded, the process is simply created in a remote node. Non-preemptive migration does not require any special support in the kernel.

Stability is an important issue in the design of a distributed scheduling algorithm. An algorithm is unstable if, under some load conditions, its overhead is not bounded. Consider a distributed scheduling algorithm that transfers a process from a heavily loaded node to a randomly selected node. If the node to which the process is sent is itself heavily loaded, the process would have to be migrated once again. Under heavy load conditions, this algorithm would lead to a situation that resembles thrashing—the scheduling overhead would be high because process migration is frequent, but processes being transferred would not make much progress.

A *sender initiated algorithm* typically transfers a process non-preemptively, i.e., if length of the CPU queue would exceed a specified threshold on creation of a new process. A sender node polls other nodes in the system to find a node whose CPU queue length is smaller than a threshold value, or to find the lightest loaded node that satisfies this condition. A sender initiated algorithm may be unstable at high system loads because a sender that cannot find a lightly loaded node would poll continuously and waste a considerable fraction of its CPU's time. Instability can be prevented by limiting the number of attempts a sender is allowed to make to find a receiver. If this number is exceeded, the sender would abandon the attempt the process migration attempt, and create the new process locally. Instability may also result if several processes are sent to the same receiver node, which now becomes a sender node and has to migrate some of the processes it received. This situation can be avoided by using a protocol whereby a node accepts a process only if it is still a receiver node (see Problem 14 in Exercise 17).

A *receiver initiated algorithm* checks whether a node is a receiver node every time a process in the node completes. It now polls other nodes in the system to find a node that would not become a receiver node even if a process is transferred out of it, and transfers a process from such a node to the receiver node. Thus, process migration is necessarily preemptive. At high system loads, the polling overhead would be bounded because the receiver would be able to find a sender. At low system loads, continuous polling by a receiver would not be harmful because idle CPU times would exist in the system. Unbounded load balancing overhead can be prevented by abandoning a load balancing attempt if a sender cannot be found in a fixed number of polls; however, a receiver must repeat load balancing attempts at fixed intervals of time to provide the liveness property.

system and few, if any, receivers exist. Consequently, the response time increases sharply. A receiver initiated algorithm incurs a higher overhead at low system loads than a sender initiated algorithm because a large number of receivers exists at low system loads. Hence the response time is not as good as when a sender initiated algorithm is used. At high system loads, few receivers exist in the system, so a receiver initiated algorithm performs distinctly better than a sender initiated algorithm. The performance of a symmetrically initiated algorithm would resemble that of a sender initiated algorithm at low system loads that of receiver initiated algorithms at high system loads.

17.6 DISTRIBUTED TERMINATION DETECTION

A process ties up system resources such as kernel data structures and memory. The kernel releases these resources either when the process makes a 'terminate me' system call at the end of its operation, or when it is killed by another process. This method is not adequate for processes of a distributed computation because they may not be able to decide when they should terminate themselves or kill other processes. For example, consider a distributed computation whose processes have a client-server relationship. A server would not know whether any more requests would be made to it, because it would not know who its clients are and whether all of them have completed their operation. In such cases, the kernel employs methods of *distributed termination detection* to check whether the entire distributed computation has terminated. If it is so, it releases the resources allocated to all of its processes.

Two process states are defined for facilitating termination. A process is in the *passive state* when it has no work to perform; it is dormant and waits for some other process to send it some work through an interprocess message. A process is in the *active state* when it is engaged in performing some work. It can be performing I/O, waiting for a resource, waiting for the CPU to be allocated to it, or executing instructions. The state of a process changes several times during its execution. A passive process becomes active immediately on receiving a message, sends an acknowledgment to the sender of the message, and starts processing the message. An active process acknowledges a message immediately, though it may delay its processing until a convenient time. An active process becomes passive when it finishes its current work and does not have other work to perform. It is assumed that both control and basic messages travel along the same interprocess channels.

A distributed computation is said to have terminated if it satisfies the *distributed termination condition* (DTC). The DTC is comprised of two parts:

1. All processes of a distributed computation are passive
 2. No basic messages are in transit.
- (17.1)

The second part is needed because a message in transit will make its destination process active when it is delivered. We discuss two approaches to determine whether DTC holds for a distributed computation.

Credit distribution-based termination detection In this approach by Mattern[1989], every activity or potential activity in a distributed computation is assigned a numerical weightage called *credit*. A distributed computation is initiated with a known finite amount of credit C . This credit is distributed among its processes. The manner of its distribution is immaterial so long as each process P_i receives a non-zero credit c_i . When a process sends a basic message to another process, it puts a part of its credit into the message—again, it is immaterial how much credit it puts into a message, so long as it is neither zero nor its entire credit. A process receiving a message extracts the credit from the message and adds it to its own

credit before processing the message. When a process becomes passive, it sends its entire credit to a special system process called the *collector process*, which accumulates all credit it receives. The distributed computation is known to have terminated when the credit accumulated by the collector process equals C . This algorithm is simple and elegant; however, credit may be distributed indefinitely, so a convenient representation of credit should be used in its implementation.

Diffusion computation-based termination detection Each process that becomes passive initiates a diffusion computation to determine whether the DTC holds. Thus, every process has the capability to detect termination. We discuss detection of the DTC in a simplified setting where the following three rules hold:

1. Processes are not created or destroyed dynamically during execution of the computation, i.e., all processes are created when the distributed computation is initiated, and remain in existence until the computation terminates.
2. Interprocess communication channels are FIFO.
3. Processes communicate with one another through synchronous communication, i.e., the sender of a message becomes blocked until it receives an acknowledgment for the message.

Rule 3 simplifies checking for the DTC as follows: The sender of a basic message becomes blocked; it resumes its operation after it receives the acknowledgment. It may enter the passive state only after finishing its work. Thus, the basic message sent by a process cannot be in transit when it becomes passive and the system cannot have any basic messages in transit when all processes are passive. Hence it is sufficient to check only the first part of the DTC condition, i.e., whether all processes are passive. Algorithm 17.6 performs this check through a diffusion computation whose queries travel over the edges that represent interprocess communication. Example 17.4 illustrates operation of Algorithm 17.6.

Algorithm 17.6 (Distributed Termination Detection)

1. *When a process becomes passive:* The process initiates a diffusion computation through the following actions:
 - (a) Send “Shall I declare distributed termination?” queries along all edges connected to it.
 - (b) Remember the number of queries sent out, and await replies.
 - (c) After replies are received for all of its queries, declare distributed termination if all replies are yes.
2. *When a process receives an engaging query:* If the process is in the *active* state, it sends a *no* reply; otherwise, it performs the following actions:
 - (a) Send queries along all edges connected to it excepting the edge on which it received the engaging query.
 - (b) Remember the number of queries sent out, and await replies.
 - (c) After replies are received for all of its queries: If all replies are yes, send a *yes* reply to the process from which it received the engaging query; otherwise, send a *no* reply.
3. *When a process receives a non-engaging query:* The process immediately sends a *yes* reply to the process from which it received the query.

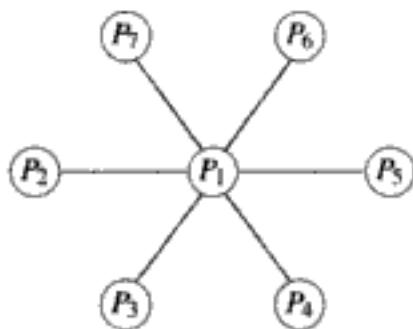


Fig. 17.10 Illustration of DTC

Example 17.4 Figure 17.10 shows a distributed computation. Only processes P_1 and P_2 are active; all other processes are passive. Now the following events occur:

1. Process P_2 becomes passive, initiates termination detection and sends a query to process P_1 .
2. Process P_1 sends a basic message to process P_5 along the edge (P_1, P_5) and becomes passive at the earliest opportunity.

The receive event in P_5 for the basic message of P_1 , and events concerning sending/receipt of queries and their replies by the processes could occur in several different sequences. Two sequences of interest are as follows: If process P_1 received the query from P_2 before it became passive, it would send a *no* reply to P_2 , so P_2 would not declare termination. If process P_1 received the query from P_2 after it became passive, according to Rule 3, it would have already received an acknowledgment to the basic message it had sent to process P_5 in Step 2, so process P_5 must have become active after receiving P_1 's message before P_1 became passive. Now, when P_1 receives the query from P_2 , it would send a query to each of P_3-P_7 . P_5 would send a *no* reply to P_1 , which would send a *no* reply to P_2 , so P_2 would not declare termination. If Rules 2 and 3 are removed, the algorithm would suffer from safety problems in some situations.

Distributed termination detection algorithms become complex when they try to remove Rules 1–3. Papers cited in the Bibliography discuss details of such algorithms.

17.7 ELECTION ALGORITHMS

A critical function like replacing the lost token in a token-based algorithm is assigned to a single process called the *coordinator* for the function. Typically, priorities are associated with processes and the highest priority process among a group of processes is chosen as the coordinator for a function. If the coordinator fails, an *election algorithm* is employed to choose the highest priority non-failed process as the new coordinator, and announce its id to all non-failed processes.

Election algorithms for ring topologies A unidirectional ring is formed by connecting adjoining processes through FIFO channels. It is assumed that the control part of a failed

Example 17.5 A system contains 10 processes P_1, P_2, \dots, P_{10} , with the priorities 1...10, 10 being the highest priority. Process P_{10} is the coordinator process. Its failure is detected by process P_2 . P_2 sends ("elect me", P_2) messages to P_3-P_{10} . Each of P_3-P_9 respond by sending a "don't you dare" message to P_2 , and start their own elections by sending "elect me" messages to higher priority processes. Eventually processes P_2-P_8 receive "don't you dare" messages from all higher priority processes excepting P_{10} , which has failed. Process P_9 does not receive any "don't you dare" message, so it elects itself as the coordinator and sends a ("new coordinator", P_9) message to P_1-P_8 . During the election, 36 "elect me" messages, 28 "don't you dare" messages and 8 "new coordinator" messages are sent. The total number of messages for this election is thus 72.

If the same system had been organized as a unidirectional ring with edges $(P_i, P_{i+1}) \forall i < 10$ and edge (P_{10}, P_1) , a total of 27 messages would have been needed to complete the election.

17.8 PRACTICAL ISSUES IN USING DISTRIBUTED CONTROL ALGORITHMS

17.8.1 Resource Management

When a process requests access to a resource, the resource allocator must find the location of matching resource(s) in the system, determine their availability, and allocate one of the resources. Figure 17.11 contains a schematic of resource allocation. A *resource manager* exists in each node of the system. It consists of a name server and a resource allocator. The numbered arcs in the schematic correspond to steps in the following resource allocation procedure:

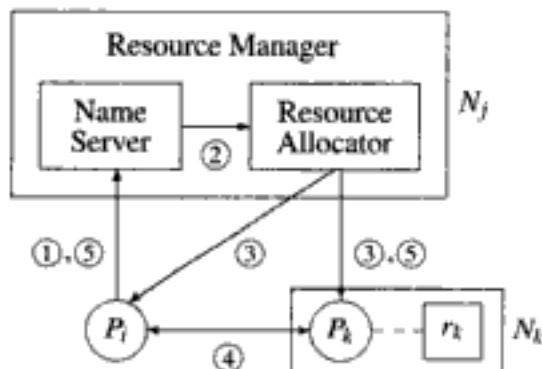


Fig. 17.11 Resource allocation in a distributed system

1. When process P_i wishes to use a resource named res_j , it constructs a pair (res_j, P_i) and forwards it to the resource manager. The resource manager receives the request and hands it over to the name server.
2. The name server locates res_j using its name and attributes, and constructs the triple (r_k, N_k, P_i) where res_j is resource r_k at node N_k . The triple is handed over to the resource allocator.
3. The resource allocator finds whether resource r_k of node N_k is available. If so, it passes P_k , the id of the resource controller process for the resource, to P_i . It also sends

an allocation message containing the id of P_i to P_k . If the resource is not available, it stores the request in a queue of pending requests. The request would be honored sometime in future when the resource becomes available.

4. Process P_k interacts with process P_l to fulfill P_l 's service requests.
5. After completing its use of the resource, process P_l makes a release request. The resource manager sends a release message to P_k and allocates the resource to a pending request, if any.

The important issue in Step 3 is ensuring non-interference of resource allocators of different nodes. It could be achieved either through a distributed mutual exclusion algorithm or through an election algorithm to elect a coordinator that would perform all allocations in the system. Both these approaches would incur high overhead. Use of a Mutual exclusion algorithm would incur overhead at every allocation. Use of an election algorithm would avoid this overhead. However, it would require a protocol to ensure that resource status information would be accessible to a new coordinator if the present coordinator failed. A simpler arrangement would be to entrust allocation of resources in a node to the resource allocator of that node. This scheme would avoid the overhead of mutual exclusion, election, and fault tolerance. It would also be more robust because a resource could be allocated to a process so long as the nodes containing the process and the resource, and a network path between the two are functional. The name server in each node would have to be updated when resources are added. This problem can be solved through an arrangement of name servers as in the domain name service (DNS) (see Section 15.5.1).

17.8.2 Process Migration

The process migration mechanism is used to transfer a process between nodes in a distributed system. It is used to achieve load balancing, or to reduce network traffic involved in utilizing a remote resource. It may also be used to provide availability of services when a node has to be shut down for maintenance. The schematic Figure 17.8 made process migration look deceptively simple; however, in reality, it is quite complex due to several reasons. The state of a process is comprised of the following:

- Process identifier and id's of its child processes
- Pending signals and messages.
- Current working directory, file descriptors, and file buffers

Two kinds of problems are faced in transferring process state: Process state is often spread across many data structures in the kernel, so it is difficult to identify and extract it from kernel data structures. Process id's and file descriptors have to be unique in the node where a process operates; such information may have to be changed when a process is migrated. This requirement creates difficulties in process synchronization and in I/O. Providing globally unique process ids as in the Sun Cluster (see Section 15.4) and transparency of resources and services (see Section 15.9) are important in this context.

When a message is sent to a process, the DNS converts the process name ($<\text{host name}>$, $<\text{process id}>$) into the pair (IP address, $<\text{process id}>$). Such a message may be in transit when its destination process is migrated, so arrangements have to be made to deliver the message to the process at its new location. Each node could maintain the *residual state* of a process which was migrated out of it. This state would contain the id of the node to which it was migrated. If a message intended for such a process reaches this node, it would simply redirect the message to its new location. If the process had been migrated out of

this node in the meanwhile, the node would similarly redirect the message using the residual state maintained by it. In this manner a message would reach the process irrespective of its migration. However, the residual state causes poor reliability because a message may not be delivered if the residual state of its destination process has been lost or has become inaccessible due to a node fault. This situation would justify the sarcastic comment that 'a distributed system is one that I cannot use because some machine that I do not know about has failed.' An alternatively scheme would be to inform the changed location of a process (as also a change in the process id, if any) to all other processes that communicate with it. This way, a message could be sent to the process directly at its new location. If a message that was in transit when a process was migrated, reached the old node where the process once existed, the node would return a 'no longer here' reply to the sender. The sender would then resend the message to the process at its new location.

EXERCISE 17

1. State and compare the liveness properties of (a) a distributed mutual exclusion algorithm, and (b) an election algorithm.
2. Prove that the Ricart–Agrawala algorithm is FCFS and deadlock-free.
3. Step 2(b) of the Ricart–Agrawala algorithm is modified such that a process wishing to enter a CS does not send a 'go ahead' reply to any other process until it has used its CS. Prove that this modified algorithm is not deadlock free.
4. Prove the safety property of Maekawa's algorithm, which uses request sets of size \sqrt{n} .
5. Construct an example where Raymond's algorithm does not exhibit FCFS behavior for entry to a CS. (*Hint:* Consider the following situation in Example 17.2: Process P_2 makes a request for CS entry while P_3 is still in CS.)
6. Show actions of the basic and control parts of a process to implement Raymond's algorithm.
7. Identify the engaging and non-engaging queries in the Chandy–Lamport algorithm for consistent state recording (Algorithm 16.2). Extend the algorithm to collect the recorded state information at the site of the node that initiated a state recording.
8. Discuss influence of the wait-or-die and wound-or-wait schemes on response times and deadlines for processes.
9. Prove that a resource allocator using the wait-or-die and wound-or-wait scheme for deadlock detection does not possess the liveness property if a killed process is given a new time-stamp when it is re-initiated.
10. It is proposed to use an edge chasing deadlock detection algorithm for deadlocks arising in interprocess communication. When a process gets blocked on a 'receive message' request, a query is sent to the process from which it expects the message. If that process is blocked on a 'receive message' request, it forwards the query to the process for which it is waiting, and so on. A process declares a deadlock if it receives its own query. Comment on the suitability of this algorithm for
 - (a) Symmetric communication.
 - (b) Asymmetric communication.
11. Prove that a distributed deadlock detection algorithm will detect phantom deadlocks if processes are permitted to withdraw resource requests when a time-out occurs.
12. If use of the *inc* function in the *block* rule is omitted from the Mitchell–Merritt algo-

- rithm, show that the modified algorithm violates the liveness requirement.
13. Prove correctness of the credit distribution-based distributed termination detection algorithm.
 14. A sender initiated distributed scheduling algorithm uses the following protocol to transfer a process from one node to another:
 - (a) A sender polls all other nodes in the system in search of a receiver node.
 - (b) It selects a node as the prospective receiver, and sends it a 'lock yourself for a process transfer' message.
 - (c) The recipient of the message sends a *no* reply if it is no longer a receiver. Else it increases the length of its CPU queue by 1 and sends a *yes* reply.
 - (d) The sender transfers a process when it receives a *yes* reply.
 - (e) If it receives a *no* reply, it selects another node and repeats Steps 14(b)–14(e).
- Does this protocol avoid instability at high system loads?
15. Define the liveness and safety properties of a distributed scheduling algorithm. (*Hint:* Will imbalances of computational load arise in a system if its scheduling algorithm possesses liveness and safety properties?)

BIBLIOGRAPHY

Dijkstra and Scholten (1980) and Chang (1982) discuss the diffusion computation model of distributed algorithms. Andrews (1991) discusses broadcast and token passing algorithms.

Raymond (1989), Ricart and Agrawala (1981) discuss distributed mutual exclusion algorithms. Dhamdhere and Kulkarni (1994) discuss a fault tolerant mutual exclusion algorithm. The diffusion computation-based distributed deadlock detection algorithm (Algorithm 17.4) is adapted from Chandy *et al* (1983). Knapp (1987) discusses several distributed deadlock detection algorithms. Sinha and Natarajan (1984) discuss an edge chasing algorithm for distributed deadlock detection. Wu *et al* (2002) describe a distributed deadlock detection algorithms for the AND model.

Distributed termination detection is discussed in Dijkstra and Scholten (1980) and Dhamdhere *et al* (1997). The bully algorithm for distributed elections is discussed in Garcia-Molina (1982). Smith (1988) discusses process migration techniques.

Singhal and Shivaratri (1994) and Lynch (1996) describe many distributed control algorithms in detail. Tel (2000) and Garg (2002) discuss election and termination detection algorithms. Attiya and Welch (2004) discuss algorithms for the election problem.

1. Attiya, H. and J. Welch (2004): *Distributed Computing: Fundamentals, Simulations and Advanced Topics*, John Wiley, New Jersey.
2. Andrews, G. R. (1991) : "Paradigms for process interaction in distributed programs," *Computing Surveys*, **23**, 1, 49–40.
3. Chandy, K. M., J. Misra, and L. M. Haas (1983) : "Distributed deadlock detection," *ACM Transactions on Computer Systems*, **1** (2), 144–152.
4. Chang, E. (1982) : "Echo algorithms: depth parallel operations on general graphs," *IEEE Transactions on Software Engineering*, **8** (4), 391–401.
5. Dhamdhere, D. M., and S. S. Kulkarni (1994) : "A token based k -resilient mutual exclusion algorithm for distributed systems," *Information Processing Letters*, **50** (1994), 151–157.

chapter 18

Recovery and Fault Tolerance

A fault may damage the state of some data or processes. Several things could go wrong if a fault occurs during operation of a system—data consistency could be lost, a server could malfunction, resources and services could become unavailable, or the system could cease operation. To provide reliable operation, an OS avoids such consequences of faults using three approaches called *recovery*, *fault tolerance*, and *resiliency*.

Recovery in a distributed system is analogous to recovery in a file system discussed earlier in Chapter 7. When a fault occurs, some data or processes would be rolled back to states that were recorded before the fault occurred, so that the resulting states of computations and the OS would be consistent. Due to the distributed nature of computations, a roll-back of one process may require roll-backs of a few others. This requirement is called the *domino effect*. Normal operation of a computation would be resumed after recovery is completed; however, the computation may have to re-execute some actions it had performed before the fault occurred.

Fault tolerance provides un-interrupted operation of a system by repairing the states of data or processes affected by a fault, rather than by rolling them back to states that were recorded before the fault occurred. The *resiliency* approach tries to minimize the cost of re-execution when faults occur. Resiliency is achieved through special techniques for (1) remembering useful results computed in a subcomputation and using them directly, i.e., without re-execution, after a fault, and (2) re-executing a subcomputation, rather than a complete computation, when a fault occurs.

We begin this chapter with an overview of different classes of faults and various ways of dealing with them. Subsequent Sections discuss recovery, fault tolerance and resiliency.

18.1 FAULTS, FAILURES AND RECOVERY

A fault like a power outage or a memory read error may damage the state of a system. For reliable operation, the system should be restored to a consistent state, and its operation should be resumed. *Recovery* is the generic name for all approaches used for this purpose.

A fault like a power outage is noticed readily, whereas a fault like a memory read error becomes noticeable only when the damage suffered by the system state causes an unexpected behavior of the system or an unusual situation in it. Such unexpected behavior or situation is called a *failure*. Figure 18.1 illustrates how a failure arises. A fault causes an *error*, which is a part of the system state that is erroneous. An error causes unexpected behavior of the system, which is a failure. Example 18.1 discusses a fault, an error and a failure in a banking system.

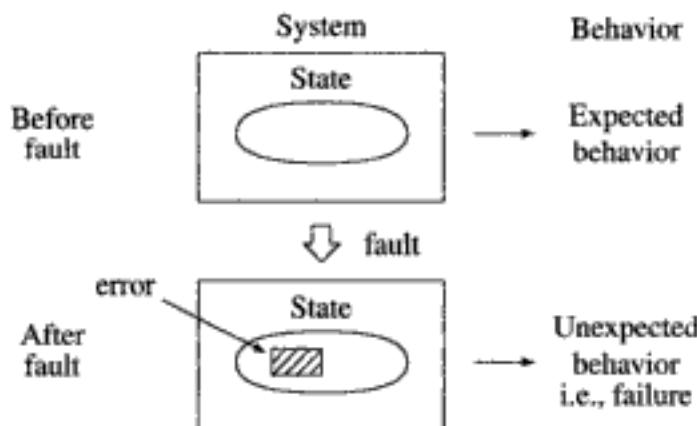


Fig. 18.1 Fault, error and failure in a system

Example 18.1 Bank accounts A and B contain 1000 dollars and 250 dollars, respectively. A banking application transfers 100 dollars from account A to account B. A power outage occurs after it deducts 100 dollars from the balance in account A, but before it adds 100 dollars to the balance in account B. The power outage is a fault. The error is that 100 dollars has been deducted from account A but has not been added to account B. The failure is that 100 dollars have vanished!

A recovery is performed when a failure is noticed. Figure 18.2 illustrates the state of a system during normal operation, after a fault, and after recovery. All times are logical times in the system. The system is initiated in a consistent state S_0 at time 0. A fault occurs at time t_1 . The fault, or the consequent failure, is detected at t_i . The system would have been in state S_i at time t_i if the fault had not occurred; however, it is actually in state S'_i . The recovery procedure applies a correction ΔS to the state and makes the system ready to resume its operation. Let the resulting state be called S_{new} . The effort spent in applying ΔS determines the cost of recovery. It would be ideal if $S_{new} = S_i$; however, the nature of a fault and the failure caused by it determine whether it could be so.

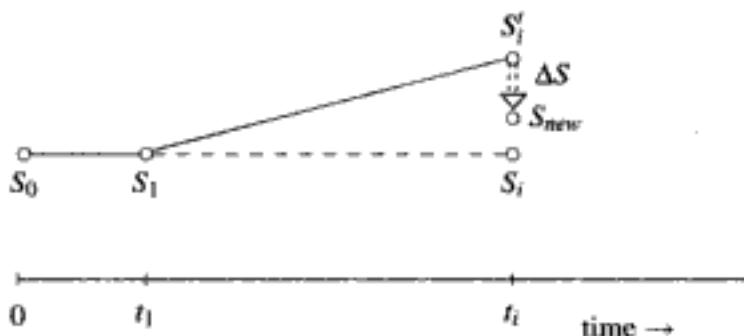


Fig. 18.2 Recovery after a fault

18.1.1 Classes of Faults

A fault may affect a computer system, affect only a process in the system, or affect hardware components such as memory and the communications hardware. Accordingly, faults are classified into system, process, storage, and communication faults. Within a class of faults, a *fault model* describes the properties of a fault which determine the kinds of errors and failures that might result from a fault.

A *system fault* is a system crash caused by a power outage or by component faults. System faults are classified into *amnesia* and *semi-amnesia* faults depending on how much state information is lost when a fault occurs. In an *amnesia fault*, the system completely ‘forgets’ the state it was in when the fault occurred. In a *semi-amnesia* fault, some components of the system state at the time of the fault survive. A *fail-stop* system fault is one that brings a system to a halt. File servers typically suffer semi-amnesia faults because they lose a part of their state that corresponds to the data that was stored in memory or on a failed disk. The *fail-stop* characteristic of a system fault is convenient because a system halts its operation whenever a fault occurs. It permits an external observer, whether a person or a computer system, to check whether a fault has occurred by observing whether the system’s operation has halted. It also provides an opportunity to recover from a fault by repairing the system state.

A process that suffers a *Byzantine* fault may perform malicious or arbitrary actions. It is not possible to undo such actions when a failure is noticed. Hence Byzantine faults are handled using redundant processes and agreement protocols. In this approach, a set of processes perform the same computation. If their results do not match, the system uses an agreement protocol to decide which of them is the correct result. Processes producing incorrect results are identified and aborted before they perform any data updates; others are permitted to perform updates and continue their operation.

A typical *storage fault* occurs due to a bad block on a storage medium. It makes some data unreadable. The occurrence of a storage fault may be detected using error checking techniques (see Section 14.3), or it may be noticed when data is accessed.

Storage faults are basically amnesia faults, however they could be made non-amnesia faults using software techniques. Communication faults are caused by link or transmission faults. These faults are non-amnesia faults because the networking software includes sufficient buffering and error handling capability to ensure that messages are not lost.

In the following, we discuss how Byzantine faults are handled in practice. The rest of this Chapter assumes faults to be non-Byzantine.

18.1.2 Overview of Recovery Techniques

For non-Byzantine faults, recovery involves restoring a system or an application to a consistent state; it does not involve undoing of wrong actions. Recovery techniques can be classified into data recovery, process recovery, fault tolerance and resiliency. These techniques have different implications for reliability, response times to computations and the cost of recovery. Table 18.1 summarizes their features.

Table 18.1 Recovery techniques

Technique	Description
Data recovery	A <i>back-up</i> is a recording of the state of a file. When a fault occurs, the state of the file is set to that found in its latest back-up (see Section 13.11).
Process recovery	A <i>checkpoint</i> is a recording of the state of a process and its file processing activities. A process is recovered by setting its state to that found in a checkpoint that was taken before a fault occurred. This action is called a <i>roll-back</i> .
Fault tolerance	The error in state caused by a fault is corrected without interrupting the system's operation.
Resiliency	Fault tolerance is performed and some results that were produced in a computation before a fault occurred are used in the computation after the fault.

Data recovery techniques guard against loss of data in a file through *back-ups*. Back-ups are taken periodically. When a fault occurs, a file is restored to the state found in its latest back-up (see Section 13.11). Hence data recovery techniques would incur substantial re-execution costs if back-ups are produced at large intervals. However, taking back-ups more frequently would incur higher overhead during normal operation. So deciding the frequency of back-ups involves a trade-off. Response times of applications would degrade considerably if re-execution costs are high.

Process recovery techniques employ *checkpoints* to record the state of a process and its file processing activities. This operation is called *checkpointing*. When a fault occurs, the recovery procedure sets the state of a process to that found in a checkpoint taken before the fault. This operation is called a *roll-back* of the process. It incurs the cost of re-execution of computations that were processed after the last

checkpoint was taken. The tradeoff between the cost of a rollback and the overhead of checkpointing during normal operation of the system is analogous to that in data recovery techniques.

Fault tolerance techniques enable a system or an application to continue its operation despite the occurrence of a fault. A fault tolerance technique recovers the system or the application to a consistent state that differs only marginally, if at all, from its state at the time when the fault occurred. Results of some computations that were in progress at the time when a fault occurred may be lost. These computations have to be re-executed.

Resiliency techniques ensure that some of the results that were produced by a computation that was in operation when a fault occurred would be used in the computation after the fault. It reduces re-execution costs and degradation of response times due to a fault.

Backward and forward recovery Recovery approaches are classified into two broad classes. *Backward recovery* implies *resetting* the state of an entity or an application affected by a fault to some prior state and resuming its operation from this state. It involves re-execution of some actions that were performed before a fault. *Forward recovery* is based on *repairing* the erroneous state of a system to restore normalcy. The repair cost depends on nature of the computation and may involve a certain amount of re-execution.

Backward recovery is simpler to implement than forward recovery. However, it assumes that state recording is both feasible and practical. This aspect poses obvious difficulties in a distributed system. Another weakness of the backward recovery technique is that an application may not make any progress if faults occur frequently. A major advantage of forward recovery is that the operation of a system or an application continues from the repaired state rather than from some previous state as in backward recovery. This feature guarantees forward progress of a computation with time for certain classes of faults.

18.2 BYZANTINE FAULTS AND AGREEMENT PROTOCOLS

Due to the difficulty in undoing wrong actions, recovery from Byzantine faults has been studied only in the restricted context of agreement between processes. The agreement problem is motivated by the *Byzantine generals* problem where a group of generals have to decide whether to attack the enemy. The generals and their armies are located in different geographical positions, hence generals have to depend on exchange of messages to arrive at a decision. Possible faults are that messages may get lost, or some generals may be traitors who deliberately send out confusing messages. An agreement protocol is designed to arrive at an agreement in spite of such faults.

Three agreement problems have been defined in the fault tolerance literature. In the *Byzantine agreement problem* one process starts the agreement protocol by

broadcasting a single value to all other processes. A process that receives the value broadcasts it to other processes. A non-faulty process broadcasts the same value which it receives. A faulty process may broadcast an arbitrary value; it may even send different values to different processes. Processes may have to perform many rounds of broadcasts before an agreement is reached. The problem requires all non-faulty processes to agree on the same value. This value should be the same as the value broadcast by the initiator if the initiator is a non-faulty process; otherwise, it could be any value. In the *consensus problem*, each process has its own initial value and all non-faulty processes have to agree on a common value. In the *interactive consistency problem*, non-faulty processes have to agree on a set of values. We discuss only the Byzantine agreement problem.

Lamport *et al* (1982) developed an agreement protocol for use when processes may fail but messages are delivered without fail. It involves $m + 1$ rounds of information exchange, where the number of faulty processes is $\leq m$. However, there are some restrictions on the value of m . Agreement is possible only if the total number of processes exceeds three times the number of faulty processes. An impossibility result states that a group of three processes containing one faulty process cannot reach agreement.

The impossibility result is easy to prove if the initiator is a faulty process. Let process P_1 , the initiator, send values 0 and 1 to processes P_2 and P_3 . Process P_2 will send 0 to process P_3 . Now, process P_3 has received two different values from two processes. It cannot decide which of the two is the correct value. A similar situation arises if P_1 is a non-faulty initiator and sends 1 to P_2 and P_3 , but process P_2 is faulty and sends 0 to process P_3 . Agreement would have been possible if the system contained n processes, $n \geq 4$, and the following algorithm was used:

1. The initiator sends its value to every other process.
2. A process receiving the value from the initiator sends it to all processes other than itself and the initiator.
3. Each process forms a collection of $n - 1$ values containing one value received from the initiator in Step 1 and $n - 2$ values received from other processes in Step 2. If it did not receive a value from the initiator or from some other process, it would assume an arbitrary value 0. It uses the value appearing the majority of times in this collection.

This is the algorithm followed for a single Byzantine fault, i.e., for $m = 1$. The algorithm for $m > 1$ is quite complex, hence we do not discuss it here.

18.3 RECOVERY

A recovery scheme consists of two components. The *checkpointing algorithm* decides when and how to create checkpoints of processes. We will use the notation C_{ij} to denote the j^{th} checkpoint taken by process P_i . The *recovery algorithm* uses the

checkpoints to roll back processes such that new process states are mutually consistent. Example 18.2 illustrates the fundamental issue in the design of checkpointing and recovery algorithms.

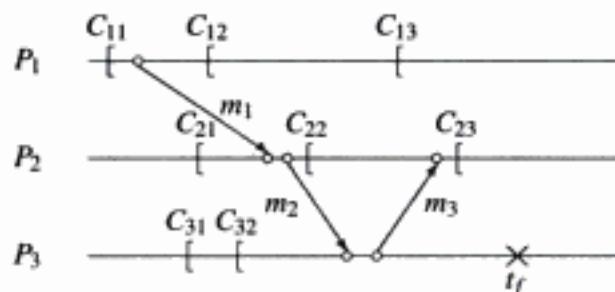


Fig. 18.3 Checkpoints of processes in a distributed system

Example 18.2 Figure 18.3 shows the timing diagram of a distributed computation. C_{11} , C_{12} and C_{13} are the checkpoints taken by process P_1 . Similarly C_{21} , C_{22} , C_{23} and C_{31} , C_{32} are the checkpoints taken by processes P_2 and P_3 , respectively. We denote the state recorded in $\langle \text{checkpoint} \rangle$ as $\text{state}(\langle \text{checkpoint} \rangle)$. Let processes P_1 , P_2 and P_3 be in the states s_1 , s_2 and s_3 , respectively, at time instant t_f . Hence the distributed computation is in the state $S \equiv \{s_1, s_2, s_3\}$. Let a failure occur in node N_3 at time instant t_f . A naive recovery algorithm simply rolls back process P_3 to its latest checkpoint, i.e., C_{32} . However, the new state of the computation, viz. $\{s_1, s_2, \text{state}(C_{32})\}$, is not a consistent state because P_2 has received message m_3 in state s_2 but P_3 has not sent m_3 in $\text{state}(C_{32})$ (see Def. 17.1).

From Example 18.2 it is clear that the state of a process cannot be recovered in isolation. Hence a recovery algorithm should restore the state of the computation to a state S' in which states of all pairs of processes are mutually consistent according to Def. 17.1. Hence the goal of a recovery algorithm is to make the following decisions for each process P_i in a distributed computation:

1. Decide whether process P_i should be rolled back.
2. If so, identify a checkpoint C_{ij} to which P_i should be rolled back.

In Example 18.2, the distributed computation could be recovered to the state $\{s_1, \text{state}(C_{22}), \text{state}(C_{32})\}$. We discuss a basis for such recovery in the following.

Definition 18.1 (Orphan message) A message m_k sent by process P_i to process P_j is an orphan message in the new state $S \equiv \{s_1, \dots, s'_j, \dots, s_k\}$ of a system during recovery if s'_j , the new state of process P_j , records m_k as received but s'_j , the new state of process P_i , does not record it as sent.

An orphan message is a message that has been received by the destination process, but it is disowned by its sender due to recovery. Hence the states of its sender and destination processes are inconsistent. This inconsistency is removed by rolling back its destination process to some state in which it had not received the orphan

message. This effect is called the *domino effect*. In Example 18.2, m_3 becomes an orphan message when P_3 is rolled back to the state in C_{32} . Hence P_2 should be rolled back to some checkpoint that was taken before P_2 received message m_3 , for example, to checkpoint C_{22} . If process P_2 had sent a message m_4 to P_1 after C_{22} and process P_1 had received this message in state s_3 , the domino effect would force a roll-back of process P_1 as well.

Using these ideas, checkpointing and recovery can be performed in one of two ways. The checkpointing algorithm would permit processes to take checkpoints at will. This method is called *asynchronous checkpointing*. At a fault, the recovery algorithm would roll back processes one-by-one in accordance with the domino effect. Alternatively, the checkpointing algorithm would coordinate the checkpointing actions of processes to ensure that the process states in the checkpoints are mutually consistent. This method is called *synchronous checkpointing*, and the collection of process checkpoints produced by it is called a *synchronous checkpoint*. When applied to the system of Figure 18.3, a synchronous checkpointing algorithm would produce either the synchronous checkpoint $\{C_{11}, C_{21}, C_{31}\}$ or the synchronous checkpoint $\{C_{12}, C_{22}, C_{32}\}$. The recovery algorithm would simply roll back each process to its individual checkpoint in the latest synchronous checkpoint.

18.4 FAULT TOLERANCE TECHNIQUES

The basic principle in fault tolerance is to ensure that a fault either does not cause an error, or the error can be removed easily. In some earlier Chapters and Sections, we saw how fault tolerance techniques ensure that no error in state would arise due to process, storage and communication faults: Section 18.2 described how process faults of a Byzantine nature can be tolerated, Section 13.11.2.2 discussed how the stable storage technique tolerates storage faults, and Section 16.5 discussed an arrangement involving acknowledgment and retransmission of messages to tolerate communication faults.

In this section, we discuss two facets of the tolerance of system faults which follow the fail-stop model.

- *Fault tolerance for replicated data:* Despite a fault, data should be available and applications should see values resulting from the latest update operation.
- *Fault tolerance for distributed data:* Despite a fault, mutual consistency of different parts of the data should not be affected.

18.4.1 Logs, Forward Recovery and Backward Recovery

A *log* is a record of actions or activities in a process. Two kinds of logs are used in practice:

- *Do logs:* A *do log* records those actions that should be performed to ensure correctness of state of an entity or a system. A do log is also called a *redo*

D may be modified, it is essential to use rules that would ensure correctness of data access and updates. We use the following rules:

1. Many processes can concurrently read D .
2. Only one process can write new values(s) into D at any time.
3. Reading and writing cannot be performed concurrently.
4. A process reading D must see its latest value.

Rules 1–3 are analogous to rules of the readers and writers problem of Section 6.7.2. Rule 4 addresses a special issue in data replication.

Quorum algorithms A quorum is the number of copies of D that must be accessed to perform a specific operation on D . Quorum algorithms ensure adherence to rules 1–4 by specifying a *read quorum* Q_r , and a *write quorum* Q_w . Two kinds of locks are used on D . A *read lock* is a shared lock, and a *write lock* is an exclusive lock. A process requesting a read lock is granted the lock if D is presently unlocked or if it is already under a read lock. Request for a write lock is granted only if D is presently unlocked. Processes use read and write quorums while accessing D , so a process can read D after putting a read lock on Q_r copies of D , and can write D after putting a write lock on Q_w copies of D .

Since a read lock is a shared lock, any value of Q_r would satisfy Rule 1. For implementing Rules 2 and 3, we choose Q_r and Q_w such that

$$2 \times Q_w > n \quad (18.1)$$

$$Q_r + Q_w > n \quad (18.2)$$

Equation (18.2) also ensures that a reader will always lock at least one copy that participated in the last write operation. This copy contains the latest value of D , so Eq.(18.2) also satisfy Rule 4.

A choice of values that satisfies Eqs.(18.1)–(18.2) is $Q_r = 1$ and $Q_w = n$. With these quorums, a read operation is much faster than a write operation. This would be appropriate if read operations are more frequent than write operations. Many other quorum values are also possible. If write operations are more frequent, we could choose values of Q_r and Q_w such that Eqs.(18.1)–(18.2) are satisfied and Q_w is as small as possible. If $Q_w \neq n$, a writer would not update all copies of D , so a reader would access some copies of D that contain its latest value, and some copies that contain its old values. To be able to identify the latest value, we could associate a time-stamp with each copy of D to indicate when it was last modified.

The choice of $Q_r = 1$ and $Q_w = n$ is not fault tolerant. $Q_w = n$ implies that a process would have to put locks on all n copies of D in order to perform a write operation. Hence a writer would be unable to write if even one node containing a copy of D failed or became inaccessible to it. If a system is required to tolerate faults in up to k nodes, we could choose

coordinator. According to the message from the coordinator, either perform commit processing or abort processing and release locks on the data.

The 2PC protocol handles failure of a participating node as follows: If a participating node fails before the 2PC protocol was initiated by the coordinator, it would not have sent a *prepared* reply to the coordinator in the first phase of the protocol. Consequently, the coordinator would abort the transaction. When the participating node recovers, it would not find a *prepared* or *abandoned* record for the transaction in its log. It would assume that the first phase of the 2PC protocol would have timed out and the coordinator would have aborted the transaction. Hence it would abandon the transaction. This action is safe because a participating node can unilaterally withdraw from a transaction any time before sending a *prepared* reply in the first phase. If the participating node fails after sending a *prepared* or *abandoned* reply to the coordinator, it would find a *prepared* or *abandoned* record in its log when it recovers. Now, it would have to query the coordinator to find whether the transaction had been committed or aborted, and accordingly perform commit or abort processing. If the node fails while it was performing commit processing, it would find a *commit* record in its log when it recovered. So it would repeat commit processing. Recall from Section 13.11.2.2 that repeated commit processing would not cause data consistency problems if the data update operations performed during commit processing are idempotent.

If the coordinator fails after writing the *commit* record in its log, but before writing the *complete* record in the log, it would see the *commit* record in its log when it recovers. It would now resend *commit* T_i messages to all participating nodes, because it would not know whether it had sent such messages before it crashed. However, this requirement constitutes a weakness in the 2PC protocol. If the coordinator had failed before sending *commit* T_i messages, participating nodes would not know whether the coordinator decided to commit or abort the transaction. Any participating node that had sent an *abandoned* T_i reply in the first phase would know that the decision could not be to commit the transaction; however, a node that had sent a *prepared* T_i reply would be blocked until the coordinator recovered and sent it a *commit* T_i or *abort* T_i message. A three phase commit protocol may be used to avoid this blocking situation.

18.5 RESILIENCY

Resiliency techniques focus on minimizing the cost of re-execution when faults occur. The basis for resiliency is the property that failures in a distributed system are partial, rather than total, so some parts of a distributed computation, or the results computed by them, may survive a failure. Use of such results after recovery would reduce, and may even avoid, re-execution. Consider a distributed transaction that is initiated in node N_i and involves computations in nodes N_i and N_j . The transaction would be aborted if the transaction manager in node N_j does not respond to the *prepare* message from the coordinator in node N_i because of the failure of node N_j .

Distributed File Systems

Users of a distributed file system expect the convenience, reliability and performance provided by conventional file systems. The convenience of using a distributed file system depends on two key issues. *Transparency* of a distributed file system makes users oblivious to the location of their files in the nodes and disks in the system. *File sharing semantics* specify the rules of file sharing—whether and how the effect of file modifications made by one process are visible to other processes using the file concurrently.

A process and a file accessed by it may exist in different nodes of a distributed system, so a fault in either node or in a communication link between the two can affect a file processing activity. Distributed file systems ensure high reliability through *file replication*, and through use of a *stateless file server* design to minimize the impact of file server crashes on ongoing file processing activities.

Response time to file system commands is influenced by network latencies in data transfer caused by processing of remote files, so the technique of *file caching* is used to reduce network traffic in file processing. Another aspect of performance is *scalability*—response times should not degrade when new nodes are added to the distributed system. It is addressed through techniques that localize data transfer to sections of a distributed system, called *clusters*, which have a high speed LAN.

This chapter discusses DFS techniques for user convenience, reliability and high performance. Case studies of distributed file systems illustrate their operation in practice.

19.1 DESIGN ISSUES IN DISTRIBUTED FILE SYSTEMS

A distributed file system (DFS) stores user files in several nodes of a distributed system, so a process and a file being accessed by it often exist in different nodes of the distributed system. This situation has three likely consequences:

- A user may have to know the topology of the distributed system to open and access files located in various nodes of the system.
- A file processing activity in a process might be disrupted if a fault occurs in the node containing the process, the node containing the file being accessed, or a communication link connecting the two.
- Performance of the file system may be poor due to the network traffic involved in accessing a file.

The need to avoid these consequences motivates the three design issues summarized in Table 19.1 and discussed in the following.

Table 19.1 Design issues in distributed file systems

Design issue	Description
Transparency of file system	High transparency of a file system implies that a user need not know much about location of files in a system. Transparency has two aspects. <i>Location transparency</i> implies that the name of a file should not reveal its location in the file system. <i>Location independence</i> implies that it should be possible to change the location of a file without having to change its name.
Fault tolerance	A fault in a computer system or a communication link may disrupt ongoing file processing activities. It affects availability of the file system and also impairs consistency of file data and <i>meta data</i> , i.e., control data of the file system. A DFS should employ special techniques to avoid these consequences of faults.
Performance	Network latency is a dominant factor of file access times in a DFS; it affects both efficiency and scalability of a DFS. Hence a DFS uses techniques to reduce network traffic generated by file accesses.

Transparency of a file system A file system finds the location of a file during path name resolution (see Section 7.8.1). Two relevant issues in a distributed file system are: How much information about the location of a file should be reflected in its path name, and can a DFS change the location of a file to optimize file access performance? The notion of transparency has two facets which address these issues.

- *Location transparency*: The name of a file should not reveal its location.
- *Location independence*: The file system should be able to change the location of a file without having to change its name.

Location transparency provides user convenience as a user or a computation need not know the location of a file. Location independence enables a file system to optimize its own performance. For example, if accesses to files stored at a node cause network congestion and result in poor performance, the DFS may move some files to other nodes. This operation is called *file migration*. Location independence can also be

used to improve utilization of storage media in the system. We discuss these two facets of transparency in Section 19.2.

Fault tolerance A fault disrupts an ongoing file processing activity, thereby threatening consistency of file data and *meta data*, i.e., control data, of the file system. A DFS may employ a journaling technique as in a conventional file system to protect consistency of meta data, or it may use a *stateless file server* design which makes it unnecessary to protect consistency of meta data when a fault occurs. To protect file data, it may provide transaction semantics, which are useful in implementing *atomic transactions* (see Section 18.4.1), so that an application may itself perform fault tolerance if it so desires. We discuss fault tolerance issues in Section 19.4.

Performance Performance of a DFS has two facets—efficiency and scalability. In a distributed system, network latency is the dominant factor influencing efficiency of a file processing activity. Network latency typically exceeds the processing time for a file record so, unlike I/O device latency, it cannot be masked by blocking and buffering of records. A DFS employs the technique of *file caching*, which keeps a copy of a remote file in the node of a process that accesses the file. This way accesses to the file do not cause network traffic, though staleness of data in a file cache has to be prevented through cache coherence techniques. *Scalability* of DFS performance requires that response times should not degrade when system size increases due to addition of nodes or users. A distributed system is composed of clusters of computer systems that are connected by high speed LANs (see Section 15.2), so caching a single copy of a file in a cluster suffices to reduce inter-cluster network traffic and provide scalability of DFS performance. When several processes access the same file in parallel, *distributed locking* techniques are employed to ensure that synchronization of the file processing activities scales well with an increase in system size. We discuss DFS performance enhancement techniques in Section 19.5.

19.1.1 Overview of DFS Operation

Figure 19.1 shows the basics of file processing in a DFS; the schematic does not show any special DFS techniques because they are yet to be discussed. A process in node N_1 opens a file with path name ...alpha. We call this process a *client process* of this file, or simply a *client* of this file, and call node N_1 the *client node*. Through path name resolution, the DFS finds that this file exists in node N_2 , so it sets up the arrangement shown in Figure 19.1. The file system component in node N_2 is called a *file server*, and node N_2 is called the *server node*. Other nodes that were involved in path name resolution or that would be involved in transferring file data between nodes N_1 and N_2 are called *intermediate nodes*.

We refer to this model as the *remote file processing* model. An arrangement analogous to RPC is used to implement file accesses through stub processes called *file server agent* and *client agent*. When the client opens the file, the request is handed

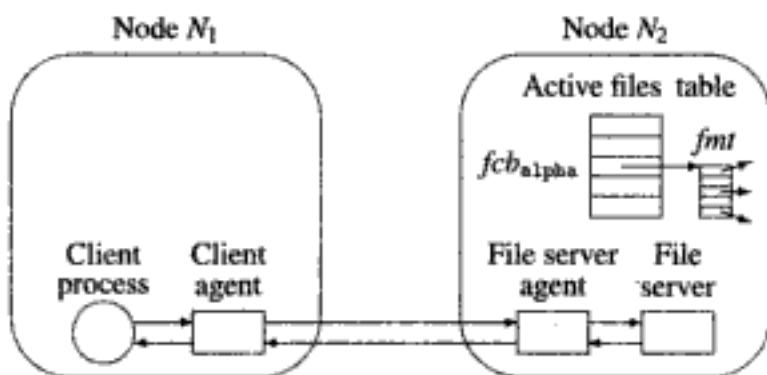


Fig. 19.1 Basics of file processing in a DFS

over to the client agent. The client agent communicates the request to the file server agent in node N_2 , which hands over the request to the file server. The file server opens alpha and builds fcb_{α} . Whenever the client performs a read or write operation on alpha, the operation is implemented through a message between the client agent and the file server agent. I/O buffers for the file exist at node N_2 , and only one record at a time gets passed to the client.

19.2 TRANSPARENCY

In a conventional file system, a user identifies a file through a path name. He is aware that the file belongs in a specific directory; however, he is not aware of its location in the system. The *location info* field of the file's directory entry indicates the file's location on disk. This arrangement would be adequate to provide *location transparency* in a DFS as well—a user would use a path name to access a file, and the DFS would obtain the location of the file from its directory entry. The DFS may choose to keep all files of a directory in the same node of the distributed system, or disperse them to different nodes. In the former case, its meta data would be identical with that of a conventional file system. In the latter case, the *location info* field of the directory entry of a file would contain a pair (node id, location).

Providing *location independence* would require the information in the *location info* field of a directory entry to vary dynamically. Now, the DFS could change the location of a file at will, so long as it puts information about the new location in the *location info* field of the directory entry. It should similarly change information in all links to the file (see Section 7.4.1). To simplify these changes, a DFS may use the following arrangement: Each file is assigned a globally unique file id. The directory entry of the file contains this file id. DFS maintains a separate data structure to hold file id–location pairs. This way, the DFS needs to change only one pair in this data structure when the location of a file is changed.

Most distributed file systems provide location transparency, but not location independence. Hence files cannot be migrated to other nodes. This restriction

Table 19.3 Fault tolerance techniques of distributed file systems

Technique	Description
Cached directories	A cached directory is a copy of a directory that exists at a remote site. It helps the DFS to tolerate faults in intermediate nodes involved in path name resolution.
File replication	Several copies of a file are maintained in the interest of availability. Special techniques are used to avoid inconsistencies between the copies. The <i>primary copy</i> technique permits client programs to read-access any copy of a file but restricts file updates only to a special copy called the primary copy. The changes are propagated to other copies. This method simplifies concurrency control.
Stateless file server	A conventional file server maintains information concerning state of a file processing activity in the meta data, for example, in file control blocks and file buffers. A stateless file server does not maintain such information, so it is immune to faults which lead to loss of state information.

approach, the DFS would perform resolution of all path components in the client node itself. When it finds that a path name component is the name of a directory in a remote node, it would copy the directory from the remote node and continue path name resolution using it. This way, all directories would be copied into the client node during path name resolution. As we shall see later, these approaches have different implications for availability. In either approach, an access to file data does not involve the intermediate nodes involved in path name resolution. File processing would not be affected if any of these nodes failed after the file was opened.

Cached directories An anomalous situation may arise when path names span many nodes. In the previous example, let node c fail after file d was opened using path name a/b/c/d and its processing was underway. If another client in node A performs the operation open a/b/c/z, where file z also exists in node D, it would fail because node c has failed. So file z cannot be processed even though its processing involves the same client and server nodes as file d.

The only way to avoid this anomaly is to cache remote directories accessed during path name resolution at the client node. For the path name a/b/c/d, it implies that the DFS would cache the directories a/b and a/b/c at node A. While resolving path names involving the prefixes a/b and a/b/c, the DFS would directly use the cached directories. Thus, it would be able to resolve the path name a/b/c/z without having to access nodes B or C. However, information in cached directories may be outdated due to creation or deletion of files in some of the intermediate nodes, so a cache updating protocol would have to be used. We discuss a related issue in the next Section.

server crashes and recovers during a file processing activity.

Use of a stateless file server provides fault tolerance, but it also incurs a substantial performance penalty due to two reasons. First, the file server opens a file at every file operation, and passes back state information to the client. Second, when a client performs a write operation, reliability considerations require that data should be written into the disk copy of a file immediately. Consequently, the file server cannot employ buffering, file caching (see Section 19.5.2), or disk caching (see Section 12.9) to speed up its own operation. In Section 19.5.1, we discuss a hybrid design of file servers that avoids repeated file open operations.

A stateless file server is oblivious of client failures because it does not possess any state information for a client or its file processing activity. If a client fails, recovers and re-sends some requests to the file server, the file server would simply reprocess them. Even though an individual read or write operation is idempotent, a sequence of operations including a read and write may not be. For example, a sequence of operations involving reading a record from a file, searching for a string xyz in the record, inserting a string S before string xyz, and writing the modified record back into the file, is not idempotent. If a failed client has performed such a non-idempotent sequence, it must restore the file to a previous state before re-issuing the sequence of operations.

19.5 DFS PERFORMANCE

Inherent efficiency of file access mechanisms determines peak performance of a DFS measured as either average response time to client requests or throughput of client requests. The DFS can achieve peak performance when all data accesses are local to client nodes, i.e., when clients and file servers are located in the same node. The actual performance of a DFS depends on the volume of network traffic generated by accesses to remote files. In fact, network latencies can completely overshadow the efficiency of access mechanisms even when only a small fraction of data accesses are non-local. This fact motivates measures to reduce network traffic by reducing the number of data transfers over the network during a file processing activity.

A DFS design is *scalable* if DFS performance does not degrade with an increase in the size of a distributed system. Scalability is important for avoiding a situation in which a DFS that used to perform well in a client's organization becomes a bottleneck when the organization becomes large. Scalability is achieved through special techniques which ensure that network traffic does not grow with size of the distributed system.

Table 19.4 summarizes techniques used to achieve high DFS performance. These techniques are discussed in the following sections.

Table 19.4 Performance techniques of distributed file systems

Technique	Description
Multi-threaded file server design	Each thread in the file server handles one client request. File processing is an I/O-bound activity, hence several threads can make progress in parallel, thereby contributing to higher throughput.
Hint-based file server design	A <i>hint</i> is some information related to an ongoing file processing activity that <i>may</i> be maintained by a file server. When a suitable hint is available, the file server behaves like a stateful file server so that it can perform a file operation efficiently; otherwise, it behaves like a stateless file server.
File caching	Some part of a file located in a remote node is copied into the <i>file cache</i> in the client node. File caching reduces network traffic during file processing by converting data transfers over the network into data transfers that are local to a client node.
Semi-independent clusters of nodes	A <i>cluster of nodes</i> is a section of the distributed system that contains sufficient hardware and software resources such that processes operating in a cluster rarely need resources located elsewhere in the system.

19.5.1 Efficient File Access

Inherent efficiency of file access depends on how the operation of a file server is structured. We discuss two server structures that provide efficient file access.

Multi-threaded file server The file server has several threads; each thread is capable of servicing one client request. Operation of several of these threads can be overlapped because file processing is an I/O-bound activity. This arrangement provides fast response to client requests and a high throughput. The number of threads can be varied in accordance with the number of client requests that are active at any time, and the availability of OS resources such as thread control blocks.

Hint-based file server A hint-based file server is a hybrid design in that it has features of both a stateful and a stateless file server. In the interest of efficiency, it operates in a stateful manner whenever possible. At other times, it operates in a stateless manner. A *hint* is some information concerning an ongoing file processing activity, e.g., id of the next record in a sequential file that would be accessed by a file processing activity (see Section 7.6). The file server maintains a collection of hints in its volatile storage. When a client requests a file operation, the file server checks for presence of a hint that would help in its processing. If a hint is available, the file server uses it to speed up the file operation; otherwise, the file server operates

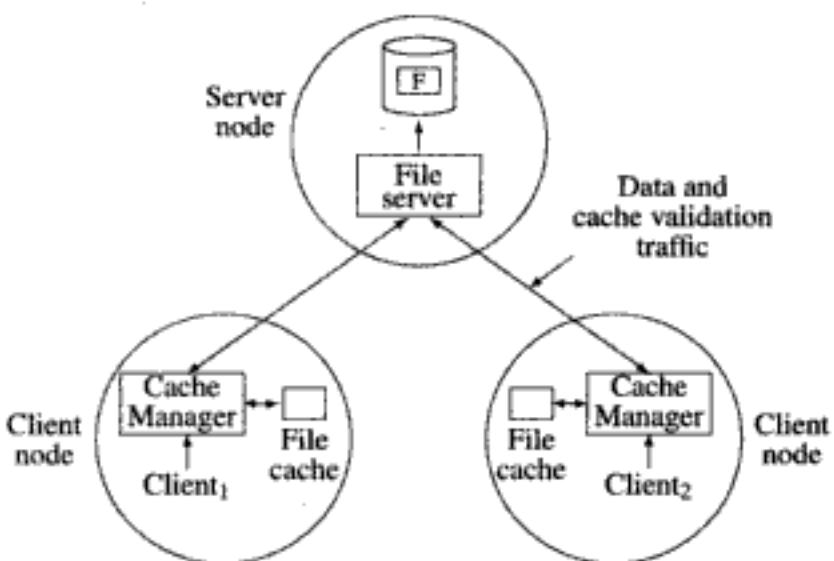


Fig. 19.3 A schematic of file caching

to be written to the file copy in the server. Locating the cache on the disk would slow down access to file data, but would provide reliability as the file cache and the modified data contained in it would survive client node crashes. Redundancy-based techniques like disk mirroring could be used to further enhance reliability of the file cache organized on a disk.

When a client performs a write operation on a disk, the modified file data would have to be written into the file copy in the server. The decision of whether to update the file copy immediately or at a later time, involves a tradeoff between delay in the client process and reliability of the DFS. It is simplest to use the *write-through policy*, which updates the file cache in the client node and the file copy in the server node at the same time. This method is reliable, because the write-through could be implemented as a transaction to ensure that it completes; however, it delays the client that performed the write operation. To avoid delaying the client, the update of the file copy could be performed at a later time provided arrangements are made to ensure that the modified data would not be lost if the client node failed in the meanwhile. This policy is called the *delayed write policy*. Its variations—the write at different times—when the modified chunk is deleted from the file cache due to replacement, or when the client closes the file.

When a file is processed by many clients in parallel, copies of its data would exist in several file caches at the same time. If one client performs a write operation, copies in other clients' caches become *invalid*, i.e., stale. The *cache validation* function identifies invalid data and deals with it in accordance with the file sharing semantics of the DFS. For example, when Unix semantics are used, file updates made by a client should be immediately visible to other clients of the file, so the cache validation function either refreshes invalid data or prevents its use by a client.

Chunk size in the file cache should be large so that spatial locality of file data contributes to a high hit ratio in the file cache. However, use of a large chunk size implies a higher probability of data invalidation due to modifications performed by other clients, hence more delays and more cache validation overhead than when a small chunk size is used. So the chunk size used in a DFS is a compromise between these two considerations. A fixed size of chunk size may not suit all clients of a DFS, so some distributed file systems, notably AFS, adapt the chunk size to each individual client.

Cache validation A simple method to identify invalid data is through time-stamps. A time-stamp is associated with a file and each of its cached chunks. The time-stamp of a file indicates when the file was last modified. When a chunk of the file is copied into a cache, the file's time-stamp is also copied as the time-stamp of the chunk. At any time, the cached chunk is invalid if its time-stamp is smaller than the file's time-stamp. This way a write operation in some part x of a file by one client invalidates all copies of x in other clients' caches. Such data is refreshed, i.e., reloaded, at its next reference.

Two basic approaches to cache validation are *client initiated validation* and *server initiated validation*. Client initiated validation is performed by the cache manager at a client node. At every file access by a client, it checks whether the required data is already in the cache. If so, it checks whether the data is valid. If the check succeeds, the cache manager provides the data from the cache to the client; otherwise, it refreshes the data in the cache before providing it to the client. This approach leads to cache validation traffic over the network at every access to the file. This traffic can be reduced by performing validation periodically rather than at every file access, provided such validation is not inconsistent with the file sharing semantics of the DFS. Sun NFS uses this approach (see Section 19.6.1).

In the server initiated approach, the file server keeps track of which client nodes contain what file data in their caches and uses this information as follows: When a client updates data in some part x of a file, the file server finds the client nodes that have x in their file cache, and informs their cache managers that their copies of x have become invalid. Each cache manager now has an option of deleting the copy of x from its cache, or of caching it afresh either immediately, or at the first reference to it.

Cache validation is an expensive operation hence some file sharing semantics like the session semantics do not require that updates made by one client should be visible to clients in other nodes. This feature avoids the need for validation altogether. Another way to avoid the cache validation overhead is to disable file caching if some client opens a file in the update mode. All accesses to such a file are directly implemented in the server node. Now, all clients wishing to use the file have to access it as in remote file processing.

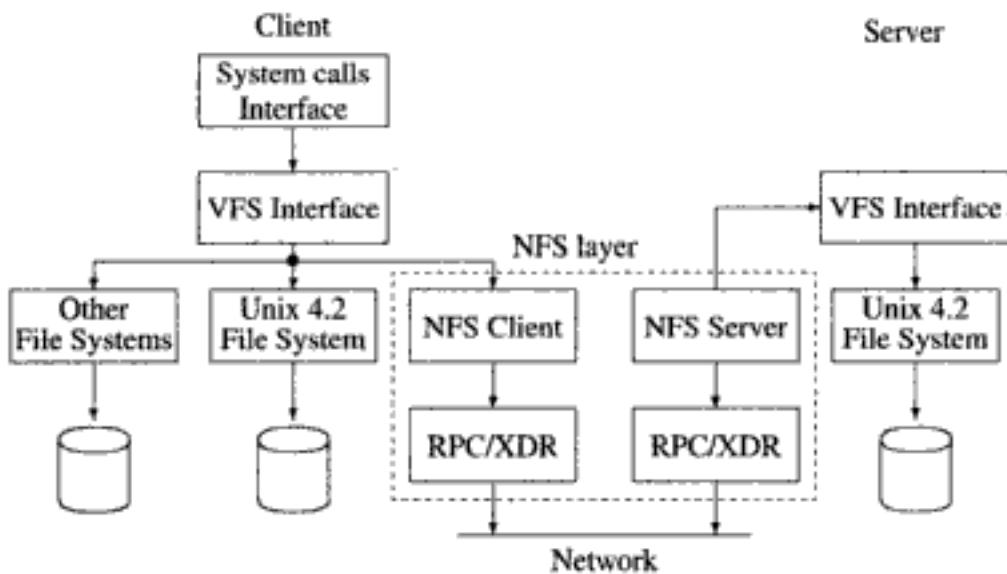


Fig. 19.4 Architecture of the Sun network file system (NFS)

tivity of the mount mechanism. For example, consider the following situation:

1. The superuser in node N_1 of the system mounts the file system C of node N_3 at mount point y in the local file system B.
2. The superuser in node N_2 mounts the file system B of node N_1 at mount point x in the local file system A.

The NFS does not permit users in node N_2 to access the file system C which was mounted over some part of file system B. This way, each host's view of the directory hierarchy is the result of the mounts performed by its own superuser only, which enables the file server to operate in a stateless manner. If this restriction was not imposed, each file server would have to know about all mounts performed by all clients over its file system, which would require the file server to be stateful.

NFS protocol The NFS protocol uses the remote service paradigm (i.e., remote file processing—see Section 19.1.1) through a client–server model employing remote procedure calls (RPC). A file server is stateless, so each RPC has parameters that identify the file, the directory containing the file, and the data to be read or written. The NFS provides calls for looking up a file within a directory, reading directory entries, manipulating links and directories, accessing file attributes, i.e., inode information; and performing a file read/write operation.

Since a file server is stateless, it performs an implicit open and close for every file operation, and does not use the Unix buffer cache (see Section 12.10.2 for a description of the Unix buffer cache). An NFS server does not provide locking of files or records; users must use their own means for concurrency control.

require human intervention in some cases.

Having a single file in cache may not be adequate for disconnected operation, so Coda provides *hoarding* of files. A user can provide a *hoarding data base*, which contains path names of important files, to Coda. During a session initiated by the user, Coda uses a prioritized cache management policy to hold some recently accessed files and files named in the hoarding data base in the cache of the user's node. This set of files is refined by recomputing their priorities periodically. This way, the cache in the node may contain an adequate set of files when the node becomes disconnected, which would enable meaningful disconnected operation.

19.6.3 GPFS

The general parallel file system is a high performance shared-disk file system for large computing clusters operating under Linux. GPFS uses data striping (see Section 12.3.3) across all disks available in a cluster. Thus, data of a file is written on several disks, which can be read from or written to in parallel. A large sized block, i.e., strip, is used to minimize seek overhead during a file read/write; however, a large disk block may not provide high data transfer rate for small files that would fit into a small number of strips, so a smaller sub-block which could be as small as 1/32 of a block is used for small files.

Locking is used to maintain consistency of file data when processes in several nodes of the cluster access a common file. High parallelism in accessing a common file requires a fine granularity of locking, whereas low locking overhead requires a coarse granularity. So GPFS uses a composite approach that works as follows: The first process that performs a write operation on a file is given a lock whose byte range covers the entire file. If no other process accesses the same file, this process does not have to set and reset locks while processing the file. If another process wishes to write into the same file, that process is given a lock with a byte range that covers the bytes it wishes to write, and the byte range in the lock already held by the first process is reduced to exclude these bytes. This way the lock granularity is as coarse as possible, but as fine as necessary, subject to the restriction that the byte range in a lock cannot be smaller than a data block on a disk. Whenever the byte range in a lock is narrowed, updates made on the bytes that are not covered by the new byte range in the lock are flushed to the file. This way, a process acquiring a lock for these bytes would see their latest values.

The locking schematic of GPFS involves a centralized lock manager and a few distributed lock managers, and employs the notion of *lock tokens* to reduce the latency and overhead of locking. The first time some process in a node accesses a file, the centralized lock manager issues a lock token to that node. This token authorizes the node to locally issue locks on the file to other processes in that node, until the lock token is taken away from it. This arrangement avoids repeated traffic between a node and the centralized lock manager for acquiring locks on a file. When a process in some other node wishes to access the same file, the centralized lock manager takes

away the lock token from the first node and gives it to the second node. Now, this node can issue locks on that file locally. The data bytes covered by byte ranges in the locks issued by a node can be cached locally at that node; no cache coherence traffic would be generated when these bytes are accessed or updated because no process in any other node would be permitted to access these bytes.

The meta data of a file, such as the direct and indirect pointers in the FMT, may become inconsistent when several nodes update the meta data in parallel. For example, when two nodes add a pointer each to the same indirect block in the FMT, one client's update would be overwritten by the other client's. Thus, one of the pointers would be lost. To prevent such inconsistencies, one of the nodes is designated as the meta node for the file, and all accesses and updates to the file's meta data are made only by the meta node. Other nodes that update the file send their meta data to the meta node and the meta node commits them to the disk. No conflicts can arise while updating the FMT entries because the byte ranges in locks held at different nodes do not overlap.

The list of free disk space can become a performance bottleneck due to allocation of disk space during file processing activities in different nodes. To avoid this, the free space map is partitioned and a central allocation manager gives one partition of the map to each node. The node makes all disk space allocations using this partition of the map. When the free space in that partition is exhausted, it requests the allocation manager for access to another partition.

Each node writes a separate journal for recovery. This journal is located in the file system to which the file being processed, belongs. When a node fails, other nodes can access its journal and carry out the pending updates. Consistency of the data bytes being updated is implicit because the failed node would have locked the data bytes; these locks are released only after the journal of the failed node is processed.

Communication failures may partition the system. However, file processing activities in individual nodes may not be affected because a node may be able to access some of the disks. This can lead to inconsistencies in the meta data. To prevent such inconsistencies, only nodes in one partition should continue file processing, while all other nodes cease file processing. GPFS achieves this as follows: Only nodes in the majority partition, i.e., the partition that contains a majority of the nodes, are allowed to perform file processing at any time. GPFS contains a group services layer that uses heartbeat messages to detect node failures; it notifies a node when the node has fallen out of the majority partition or has become a part of the majority partition once again. However, this notification may itself be delayed indefinitely due to communication failures, so GPFS uses features in the I/O subsystem to prevent those nodes that are not included in the majority partition from accessing any disks. GPFS uses a replication policy to protect against disk failures.

19.6.4 Windows

The file system of the Windows Server 2003 provides two features for data replication and data distribution:

- *Remote differential compression* (RDC) is a protocol for file replication that reduces the file replication and file coherence traffic between servers.
- *DFS namespaces* is a method of forming a virtual tree of folders located on different servers, so that a client located in any node can access these folders.

Replication is organized using the notion of a *replication group*, which is a group of servers that replicates a group of folders. This replication arrangement is convenient as several folders from the group of folders would be accessed off the same server even when failures occur. The RDC protocol is used to synchronize copies of a replicated folder across servers in its replication group. This protocol transmits only changes made to a file, or only the differences between copies of a file, between different members of a replication group, thereby conserving bandwidth between servers. Copies of a file are synchronized periodically. When a new file is created, cross-file RDC identifies existing files that are similar to the new file, and transmits only differences of the new file from one of these files to members of the replication group. This protocol reduces the bandwidth consumed by the replication operation.

The DFS namespace is created by a system administrator. For every folder in the namespace, the administrator specifies a list of servers that contain a copy of the folder. When a client refers to a shared folder that appears in the namespace, the namespace server is contacted to resolve the name in the virtual tree. It sends back a *referral* to the client, which contains the list of servers that contain a copy of the folder. The client contacts the first server in this list to access the folder. If this server does not respond and client fallback is enabled, the client is notified of this failure and goes on to contact the next server in the list. Thus, if the list of servers contains two servers, the second server acts as a *hot standby* for the first server.

EXERCISE 19

1. Write a short note on the influence of stateful and stateless file server design on tolerance of faults in the client and server nodes.
2. Discuss how session semantics can be implemented.
3. Should a DFS maintain file buffers at a server node or at a client node? What is the influence of this decision on Unix file sharing semantics (see Section 7.9) and session semantics?
4. Discuss how a client should protect itself against failures in a distributed file system using (a) a stateful file server design, (b) a stateless file server design.
5. A file server spawns multiple threads in order to provide speedy response to client requests. Describe the synchronization requirements of these threads.
6. Discuss important issues to be handled during recovery of a failed node in a system that uses file replication to provide availability.

7. Comment on validity and implementation aspects of the following statement: "In a stateless file server using caching, cache validation must be client initiated".
8. Justify the following statement: "File caching integrates well with session semantics, but not so with Unix semantics".

BIBLIOGRAPHY

Svobodova (1986) and Levy and Silberschatz (1990) are survey papers on distributed file systems. Comer and Peterson (1986) discuss concepts in naming and discuss name resolution mechanisms in many systems.

Lampson (1983) and Terry (1987) discuss use of hints to improve performance of a distributed file system. Makaroff and Eager (1990) discuss effect of cache sizes on file system performance.

Brownbridge *et al* (1982) discuss the Unix United system, which is an early network file system. Sandberg (1987), and Callaghan (2000) discuss the Sun NFS. Satyanarayanan (1990) discusses the Andrew distributed file system, while Kistler and Satyanarayanan (1992) describes the Coda file system. Braam and Nelson (1999) discuss the performance bottlenecks in Coda and Intermezzo, which is a sequel to Coda that incorporates Journaling. Russinovich and Solomon (2005) discuss replication and distributed file server in Windows.

Application processes running in different nodes of a cluster of computer systems may make parallel accesses to files. Thekkath *et al* (1997) discuss a scalable distributed file system for clusters of computer systems. Preslan *et al* (2000) describe fault tolerance in a cluster file system through journaling. Carns *et al* (2000) discuss a parallel file system that provides high bandwidth for parallel file accesses to data in common files. Schmuck and Haskin (2002) discuss use of shared disks in a parallel file system and describe distributed synchronization and fault tolerance techniques.

1. Brownbridge, D. R., L. F. Marshall, and B. Randell (1982) : "The Newcastle Connection or UNIXes of the World Unite!," *Software—Practice and Experience*, **12** (12), 1147–1162.
2. Braam, P. J., and P. A. Nelson (1999): "Removing bottlenecks in distributed file systems: Coda and InterMezzo as examples," *Proceedings of Linux Expo*, 1999.
3. Callaghan, B. (2000): *NFS Illustrated*, Addison Wesley.
4. Carns, P. H., W. B. Ligon III, R. B. Ross, and R. thakur (2000): "PVFS: A parallel file system for Linux Clusters," *2000 Extreme Linux Workshop*.
5. Comer, D., and L. L. Peterson (1986) : "A model of name resolution in distributed mechanisms," *Proceedings of the 6th International Conference on Distributed Computing Systems*, 509–514.
6. Ghemawat, S., H. Gobioff, and S. T. Leung (2003): "The Google file system," *Proceedings of the 19th ACM Symposium on Operating System Principles*, 29–43.
7. Gray, C. G., and D. R. Cheriton (1989): "Leases: an efficient fault-tolerant mechanism for distributed file cache consistency," *Proceedings of the 12th ACM Symposium on Operating Systems Principles*, 202–210.
8. Kistler, J. J. and M. Satyanarayanan (1992): "Disconnected operation in the Coda file system," *ACM Transactions on Computer Systems*, **10**, 1, 3–25.
9. Lampson, B. W. (1983) : "Hints for computer system designers," *Proceedings of the 9th Symposium of Operating Systems Principles*, 33–48.

10. Levy, E., and A. Silberschatz (1990): "Distributed File Systems: Concepts and Examples," *Computing Surveys*, **22** (4), 321–374.
11. Melamed, A. S. (1987) : "Performance analysis of Unix-based network file systems," *IEEE Micro*, 25–38.
12. Makaroff, D. J., and D. L. Eager (1990) : "Disk cache performance for distributed systems," *Proceedings of the 10th International Conference on Distributed Computing Systems*, 212–219.
13. Preslan, K. W., A. P. Barry, J. Brassow, R. Cattelan, A. Manthei, E. Nygaard, S. V. Oort, D. Teigland, M Tilstra, M. O'Keefe, G. Erickson, and M. Agarwal (2000): "Implementing journaling in a Linux shared disk file system," *Proceedings of the Seventh IEEE Symposium on Mass Storage Systems*, 351–378.
14. Russinovich, M. E., and D. A. Solomon (2005): *Microsoft Windows Internals. Fourth edition*, Microsoft Press, Redmond.
15. Sandberg, R. (1987) : *The Sun Network File System: Design, Implementation, and experience*, Sun Microsystems, Mountain View.
16. Schmuck, F., and R. Haskin (2002): "GPFS: A shared-disk file system for large computing clusters," *Proceedings of the First USENIX Conference on File and Storage Technologies*, 231-244.
17. Satyanarayanan, M. (1990) : "Scalable, secure, and highly available distributed file access," *Computer*, **23** (5), 9–21.
18. Svobodova, L. (1986) : "File servers for network-based distributed systems," *Computing Surveys*, **16** (4), 353–398.
19. Thekkath, C. A., T. Mann, and E. K. Lee (1997): "Frangipani: A scalable DFS," *Proceedings of the 16th ACM symposium on Operating System Principles*, 224-237.
20. Terry, D. B. (1987) : "Caching hints in distributed systems," *IEEE Transactions on Software Engineering*, **13** (1), 48–54.

Distributed System Security

A distributed OS faces security threats because processes in it use the network for accessing distant resources and for communicating with other processes. The network may include public communication channels or communication processors that are not under control of the distributed OS. Hence an intruder located in a communication processor may be able to corrupt interprocess messages, or use a masquerading attack to trick distant nodes and use their resources without authorization.

A distributed OS employs *message security* techniques to prevent intruders from tampering with interprocess messages. These techniques perform *encryption* with *public keys* or *session keys*, which are securely distributed to communicating processes by *key distribution centers*. To prevent masquerading, communicating processes authenticate each other through trusted third party authentication means provided in the distributed OS.

In this chapter, we discuss message security and authentication techniques of distributed systems. We also discuss how integrity and authenticity of data is ensured through *message authentication codes* and *digital signatures*, respectively.

20.1 ISSUES IN DISTRIBUTED SYSTEM SECURITY

Resources and services in a distributed OS are located in *secure nodes*, i.e., nodes that are directly under control of the distributed OS. As shown in Figure 20.1, a user process accesses a remote resource through a message sent to the resource coordinator process. Such a message may travel over public networks and pass through communication processors that operate under local operating systems.

Communication processors employ the store-and-forward technique to route a message to its destination. Thus, messages are exposed to observation and interference by external entities, thereby raising new security threats that do not arise in a conventional system. Such threats can be classified as follows:

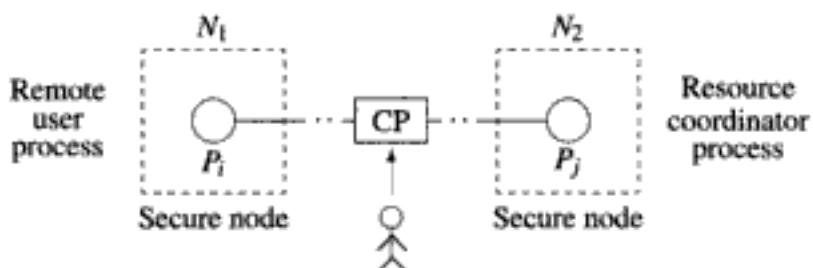


Fig. 20.1 Security threats in a network

1. *Leakage*: Release of message contents to unauthorized user(s).
2. *Tampering*: Modification of message contents.
3. *Stealing*: Use of system resources without authorization.
4. *Denial of service to authorized users*: This threat can be in the form of willful destruction of system resources not amounting to stealing, i.e., destruction without any gain to the perpetrator, or disruption of access to resources.

Leakage and tampering are termed threats to *message security*. Tampering may be employed to modify the text of a message, which is a threat to its *integrity*, or modify the identity of its sender, which is a threat to its *authenticity*. In addition to these threats, an intruder may also be able to masquerade as an authorized user by fabricating and inserting messages. Similarly, a user may be able to fool a resource coordinator located in a remote node and masquerade as another user. These security threats are addressed through two means:

- *Message security techniques*: Special techniques are employed to thwart attacks on messages.
- *Authentication of remote users*: Trusted means are provided to authenticate remote users.

Attacks on integrity and authenticity are addressed through a combination of these two means.

20.1.1 Security Mechanisms and Policies

Figure 20.2 shows an arrangement of security mechanisms and policies. *Authentication* in conventional systems has been described earlier in Chapter 8. Authentication in a distributed system has two new facets: The authentication service must be trustworthy and available to all nodes in a system. *Encryption* is used to ensure secrecy and integrity of the authentication and authorization data bases. It is also used to implement message security by encoding the text of each message. Such encryption uses a lower-level mechanism called *key distribution* to generate and distribute encryption keys for use by communicating processes. We discuss distribution of encryption keys in Section 20.2.1.

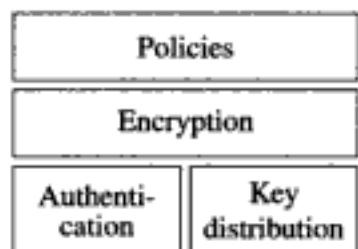


Fig. 20.2 Mechanisms and policies for distributed system security

20.1.2 Security Attacks in Distributed Systems

Security attacks in distributed systems, which are typically launched through messages, can be classified into the four categories summarized in Table 20.1. *Eavesdropping* can take various forms like obtaining the content of a message, obtaining statistical information about messages passing over a link and about messages exchanged by specific nodes in the system. In a police or military information system, the latter analyses can be used to reveal or guess identities of communicating entities. *Message tampering* can be used to mislead the recipient of a message. This attack is feasible in a store-and-forward network.

Table 20.1 Classes of security attacks in distributed systems

Attack	Description
Eavesdropping	An intruder listens to interprocess messages over the network to obtain information concerning message content or statistical features of messages.
Message tampering	An intruder intercepts messages, alters their contents, and reinserts them into the communication stream.
Message replay	An intruder makes copies of messages exchanged by communicating processes and inserts the copies into the communication stream at a later time as if they were genuine messages being sent at that time.
Masquerading	An intruder is able to impersonate an authorized user of the system while consuming resources, and while sending and receiving messages.

Message replay can be used to achieve a variety of nefarious ends. The recipient of a replayed message may be misled into thinking that messages are being exchanged in real time. If the recipient is a user process, it might be fooled into taking actions that are unnecessary, absurd or wasteful in terms of resources. It may also be misled into revealing confidential information. If the recipient is a server process, a replayed message may lead to wrong authentication, leading to opportunities for impersonation or stealing of resources.

In *masquerading*, an intruder is able to impersonate an authorized user of the

system. The intruder could corrupt or destroy information belonging to the impersonated user, or communicate with other processes and trick them into believing that they are communicating with the impersonated user.

Passive and active attacks Security attacks can be classified into *passive* attacks and *active* attacks. A passive attack does not interfere with the system's functioning in any manner. It neither fabricates messages nor destroys genuine messages. Eavesdropping is an example of a passive attack. An active attack interferes with a system's functioning. Replay, fabrication, modification and destruction of messages are examples of active attacks. Passive attacks are harder to detect or thwart than active attacks.

20.2 MESSAGE SECURITY

Approaches to message security can be classified into *link oriented approaches* and *end-to-end approaches*. In a link oriented approach, security measures are applied at every link of a communication path. This approach tends to be expensive since its cost depends on the number of links over which a message travels. For example, if a message between process P_i located at node N_1 and process P_j located at node N_3 passes along the path $N_1-N_2-N_3$, it has to incur security overhead for links N_1-N_2 and N_2-N_3 . In the end-to-end approach, security measures can be employed selectively by nodes or processes in the system. This feature permits users to employ security measures with varying degrees of cost and sophistication. In the following discussion, we will assume that end-to-end measures are used.

We describe three approaches to message security. They involve encryption using public keys, private keys and session keys, respectively. Table 20.2 summarizes their features.

Private key encryption Private key encryption (also called *secret key encryption*) is the classical approach based on symmetric keys. Each process P_i has a *private key* V_i known to itself and to a few other processes in the system. A process sending a message to P_i must encrypt it using V_i . On receiving a message, P_i decrypts it using V_i . The main advantage of private key encryption is that the number of encryption keys in the system is limited to n , where n is the number of communicating entities in the system. Since all messages intended for process P_i are encrypted with the same key, P_i need not know the identity of the sender of a message in order to read the message.

Private key encryption suffers from a few drawbacks. Each sender process needs to know the private key of P_i . Thus, many processes know the private key of a process, and an intruder may discover the key due to somebody's negligence. The private key is exposed to intruder attacks over a long period of time, so chances of a successful attack on the private key increase with time; however, it is not possible to change the private key of a process because it is known to many other processes in

Table 20.2 Encryption techniques used for message security

Technique	Description
Private key encryption	Employs symmetric encryption. A process P_i in the system has a unique encryption key V_i called the <i>private key</i> . All messages sent to P_i must be encrypted using V_i . Process P_i decrypts them using V_i . The private key of a process is exposed to intruder attacks over the entire lifetime of a process.
Public key encryption	Employs asymmetric encryption. A process P_i in the system has a pair of unique keys (U_i, V_i) . U_i is the <i>public key</i> which is made known to all processes in the system, whereas V_i is the <i>private key</i> , which is kept secret. Messages to P_i are encrypted using U_i , but P_i decrypts them using V_i . The Rivest–Shamir–Adelman (RSA) algorithm is widely used to generate the pair of keys for a process. The private key of a process is not exposed to intruder attacks.
Session key encryption	A pair of communicating processes (P_i, P_j) is assigned a <i>session key</i> SK_{ij} when they begin a communication session. The session key is used for symmetric encryption of all messages exchanged during the session. The session key has a smaller lifetime than a private or public key, so it suffers less exposure to intruder attacks.

the system.

User processes do not know each other's private keys, so private key encryption is not useful for security of interprocess messages in general. OS processes know private keys of user processes, so they use private key encryption while communicating with user processes. As discussed in Section 20.2.1, this feature is used in the implementation of key distribution centers. User processes need to use some other encryption scheme while communicating with one another.

Public key encryption Each process P_i has a pair of keys (U_i, V_i) . U_i is the *public key* of P_i , which can be made known to all processes in the system. V_i is the *private key* known only to process P_i . U_i and V_i are chosen such that

- V_i cannot be guessed from U_i , and
- for any message m

$$D_{V_i}(E_{U_i}(P_m)) = P_m \quad \forall i \quad (20.1)$$

where P_m is the plaintext form of message m (see Section 8.5), and E, D are the encryption and decryption functions, respectively.

Three steps are involved in transmitting a message m from process P_j to process P_i :

1. Process P_j encrypts the message with the public key of the destination process P_i , i.e., with U_i .
2. The encrypted message, i.e., $E_{U_i}(P_m)$, is transmitted over the network and is received by process P_i .
3. Process P_i decrypts the received message with its own private key, i.e., with V_i . Thus, it performs $D_{V_i}(E_{U_i}(P_m))$, which yields P_m .

When public key encryption is used, the OS can publish directories of public keys for all processes in the system. A process can consult the directory to obtain the public key of a process with which it wishes to communicate.

The Rivest–Shamir–Adelman (RSA) encryption algorithm is used to generate pairs of keys (U_i, V_i) which satisfy Eq. (20.1). Let (u, v) be such a pair of keys generated by the RSA algorithm. Given two numbers x and y , both smaller than a chosen integer number n , encryption and decryption using u and v , respectively, is performed as follows:

$$\begin{aligned} E_u(x) &= x^u \bmod n \\ D_v(y) &= y^v \bmod n \end{aligned}$$

To encrypt and decrypt a message m , the RSA algorithm is used as a block cipher with a block size s , which is chosen such that $n > 2^s$. x is now the number formed by the bit-string found in a block of P_m , the plaintext form of message m , and y is the number formed by the bit-string in the corresponding block of C_m , the ciphertext form of message m . This way, $x < 2^s$ and $y < 2^s$, so each of them is smaller than n , as required.

The RSA algorithm chooses n as the product of two large prime numbers p and q . Typically, p and q are 100 digits each, which makes n a 200 digit number. Assuming u and v to be the public and private keys, to satisfy relation (20.1) v should be relatively prime to $(p - 1) \times (q - 1)$ (i.e. v and $(p - 1) \times (q - 1)$ should not have any common factors except 1), and u should satisfy the relation

$$u \times v \bmod [(p - 1) \times (q - 1)] = 1.$$

Choice of u and v as the public and private keys implies that a standard value of n is used in the system. Alternatively, the pair (u, n) can be used as the public key and the pair (v, n) can be used as the private key of a process. This will permit different values of n to be used for different pairs of processes.

An attack on the RSA cipher can succeed if n can be factored into p and q . However, it is estimated that factorization of a 200 digit number, which would be needed to break the cipher, would need four billion years on a computer which can perform one million operations per second.

Public key encryption suffers from some drawbacks when compared with private key encryption. Keys used in public key encryption are approximately an order of magnitude larger in size than private keys. This is unavoidable since public keys

1. $P_i \rightarrow \text{KDC} : P_i, P_j$
 2. $\text{KDC} \rightarrow P_i : E_{V_i}(P_j, SK_{i,j}), E_{V_j}(P_i, SK_{i,j})$
 3. $P_i \rightarrow P_j : E_{V_j}(P_i, SK_{i,j}), E_{SK_{i,j}}(<\text{message}>)$
- (20.3)

In the second step, the KDC sends a reply to P_i , which is encrypted with P_i 's private key. The reply contains the session key $SK_{i,j}$ and $E_{V_j}(P_i, SK_{i,j})$, which is the session key encrypted with P_j 's private key. P_i decrypts the KDC's message with its own private key to obtain the session key $SK_{i,j}$. Decryption also yields $E_{V_j}(P_i, SK_{i,j})$. P_i copies this unit in the first message it sends to P_j . When P_j decrypts this unit, it obtains $SK_{i,j}$ which it uses to decrypt all subsequent messages from P_i .

In a public key system, session keys need not be distributed by the KDC—a sender process can itself choose a session key. This approach is possible because processes can communicate with each other using public keys. Thus a process P_i can employ the following protocol to communicate a session key to process P_j :

1. $P_i \rightarrow \text{KDC} : E_{U_{\text{kdc}}}(P_i, P_j)$
 2. $\text{KDC} \rightarrow P_i : E_{U_i}(P_j, U_j)$
 3. $P_i \rightarrow P_j : E_{U_j}(P_i, SK_{i,j}), E_{SK_{i,j}}(<\text{message}>)$
- (20.4)

The first two steps of this protocol are identical with the first two steps of protocol (20.2); they provide P_i with the public key of P_j . Now, P_i itself generates a session key $SK_{i,j}$, and passes the session key and its first message to P_j using Step 3.

20.2.2 Preventing Message Replay Attacks

In a message replay attack, an intruder simply copies messages passing over the network and ‘plays them back’ in future. A replayed message may mislead its recipient into taking wrong or duplicate actions which may affect data consistency or reveal confidential information. For example, in a system using session keys, an intruder could replay the message of Step 3 in protocol (20.3) or protocol (20.4). When P_j receives the replayed message, it would be tricked into thinking that P_i is communicating with it using the session key $SK_{i,j}$. When process P_j responds to this message, the intruder would replay the next copied message. In this manner, it could replay an entire session.

The recipient of a message can employ the *challenge-response protocol* to check whether the message exchange is taking place in real time. Steps of the challenge-response protocol are as follows:

- *Challenge*: When a process P_j receives a message originated by a process P_i , it throws a challenge to P_i to prove that it is engaged in a message exchange with it in real time. The challenge is in the form of a message containing a *challenge string*, which is encrypted in such a manner that only process P_i can decrypt it.

5. $P_i \rightarrow P_j : E_{U_j}(n+1)$
6. $P_i \rightarrow P_j : E_{SK_{i,j}}(<\text{message}>)$

In Step 4, P_j sends a nonce n encrypted with the public key of P_i . The identity of P_i is verified by its ability to decrypt this message, extract the nonce, and transform it in the expected manner. Note that in Step 4, P_j must not encrypt its message using the session key $SK_{i,j}$ as the intruder would be able to decrypt such a message if he had fabricated the message in Step 3!

20.3 AUTHENTICATION OF DATA AND MESSAGES

Authenticity of data requires that a process should be capable of verifying that data was originated or sent by a claimed person or process and that it has not been tampered with by an intruder. The latter aspect implies *integrity* of data.

Integrity of data is ensured as follows: When data d is originated or is to be transmitted, a special one-way hash function h is used to compute a hash value v . This hash value, also called a *message digest*, has a fixed length irrespective of the size of data. Apart from the properties of one-way functions described earlier in Section 8.5.1, this special one-way hash function has the property that a *birthday attack* is infeasible, i.e., given the hash value v of data d , it is impractical to construct another data d' whose hash value would also be v . The data and the hash value are stored and transmitted as a pair $\langle d, v \rangle$. To check the authenticity of d , its hash value is computed afresh using h , and it is compared with v . Following from the special property of h mentioned above, data d is considered to be in its original form if the two match; otherwise, d has been tampered with. For this scheme to work, the value v should itself be protected against tampering or substitution by an intruder; otherwise, an intruder could substitute a pair $\langle d, v \rangle$ by another pair $\langle d', v' \rangle$ and mislead other processes into thinking that data d' is genuine. Accordingly, the person or process originating or transmitting d encrypts v or the pair $\langle d, v \rangle$ so that tampering or substitution of v can be detected. Note that it is cheaper to encrypt v rather than $\langle d, v \rangle$.

Authenticity requires one more check. The encryption of v or $\langle d, v \rangle$ should be performed in such a manner that its successful decryption would establish the identity of the person or process that originated or transmitted d . This requirement is met by setting up *certification authorities*, which provide a secure method of distributing information concerning encryption keys used by persons or processes.

20.3.1 Certification Authorities and Digital Certificates

A certification authority (CA) assigns public and private keys to an entity, whether a person or a process, after ascertaining its identity using some means of physical verification. The keys are valid for a specific period of time. The certification authority also acts like a key distribution center discussed in Section 20.2.1: It keeps a record of all keys allocated by it, and when a process requests it for the public

key of some person or process, it issues a *public key certificate* which includes the following information:

- Serial number of the certificate
- Owner's distinguished name (DN)
- Identifying information of owner, such as address
- Owner's public key
- Date of issue and date of expiry, and the issuer's distinguished name
- Digital signature on the above information by the certification authority.

A number of certification authorities could operate in parallel. A server would obtain a certificate from one of these. If a client knows which certification authority a server is registered with, it can request the certification authority for the server's public key certificate. Alternatively, if it knows the IP address of the server, it can request the server to forward its own public key certificate.

The primary item of information in the certificate of an entity is the public key of the entity. However, before the receiver of the certificate uses the key to communicate with the entity, it has to ensure that the certificate is genuine and belongs to the entity with which it wishes to communicate, i.e., it is not subject to a security attack called the *man-in-the-middle* attack. In this attack, an intruder masquerades as a server. When a client requests the server for a digital certificate, the intruder intercepts the message and sends a forged certificate containing the intruder's public key to the client. Now, if it can intercept subsequent messages from the client to the server, it can read those messages. If it so desires, it can initiate a conversation with the genuine server, this time masquerading as the client, and pass on the client's messages to the server after reading them. Neither the client nor the server would be able to discover that they are subject to a successful man-in-the-middle attack.

The public key certificate contains many items of information which are used to prevent such attacks. The certificate is digitally signed by the certification authority. The client can use this digital signature to ensure that the certificate has not been faked. For this, it requires the public key of the certification authority that issued the certificate. If it does not already know this, it can request a higher order certification authority for a certificate of this certification authority. Once genuineness of the certificate has been established, it can check whether the certificate is valid by checking whether the current date falls within the validity period of the certificate. If it knows the IP address of the server, it can check that against the IP address information mentioned in the certificate. It begins exchanging messages with the server only if all these checks succeed.

20.3.2 Message Authentication Codes and Digital Signature

A *message authentication code* (MAC) is used to check the integrity of data. A process that originates or transmits data d obtains MAC_d , the message authentication

code of d , as follows: It generates a message digest v for d through a one-way hashing function. It encrypts v using an encryption key that is known only to itself and to the intended recipient of d . It now stores or transmits the pair $\langle d, MAC_d \rangle$. Only the intended recipient of d can check and verify the integrity of d .

A *digital signature* is used to ensure authenticity of data. A person or process that originates or transmits data d obtains v from d as mentioned above. It now obtains DS_d , the digital signature of d , by encrypting v and, optionally, a time-stamp, using its own private key. The pair $\langle d, DS_d \rangle$ is now stored or transmitted. Any process that wishes to check the authenticity of d decrypts DS_d using the public key of the originator of d . Successful decryption validates the integrity of d and also identifies its originator or sender in a non-repudiable manner, i.e., the identified originator or sender cannot deny creating or sending the data. Non-repudiability arises from the fact that the data was encrypted using the private key of the originator or sender, which is known only to itself. The digital signature can also be used to detect any modifications of data after the data was created or sent by a process.

Figure 20.5 illustrates steps in the use of a digital signature. The sender applies a one-way hash function to the text of a message to obtain a message digest. He signs the message digest by encrypting it with his private key. This digital signature is added at the end of the message text before sending the message. The recipient applies the same one-way hash function to the message text received by it to obtain its message digest. It now obtains a public key certificate of the sender of the message, and uses the public key contained in it to decrypt the digital signature. This step yields the message digest that was computed by the sender. The recipient compares this message digest with its own message digest. The message is authentic only if the two match and the time-stamp in the digital signature is within the validity period of the public key certificate.

20.4 THIRD-PARTY AUTHENTICATION

An open system uses standard, well-specified interfaces with other systems. A process in any node with matching interfaces can request access to resources and services of an open system. This fact gives rise to an obvious problem in authentication—how does a server know whether a process wishing to act as its client was created by an authorized user? One solution is to require each server to authenticate every user through a password. This approach is inconvenient since each server would have to possess a system-wide authentication database and each user would be authenticated several times while using the system. An alternative is to use a third-party authenticator and a secure arrangement by which the authenticator can introduce an authorized user to a server. This way each server does not have to authenticate each user.

We discuss two protocols for third-party authentication in a distributed system. The Kerberos protocol employs an authentication database, whereas the *secure sockets layer* (SSL) protocol performs authentication in a decentralized manner.

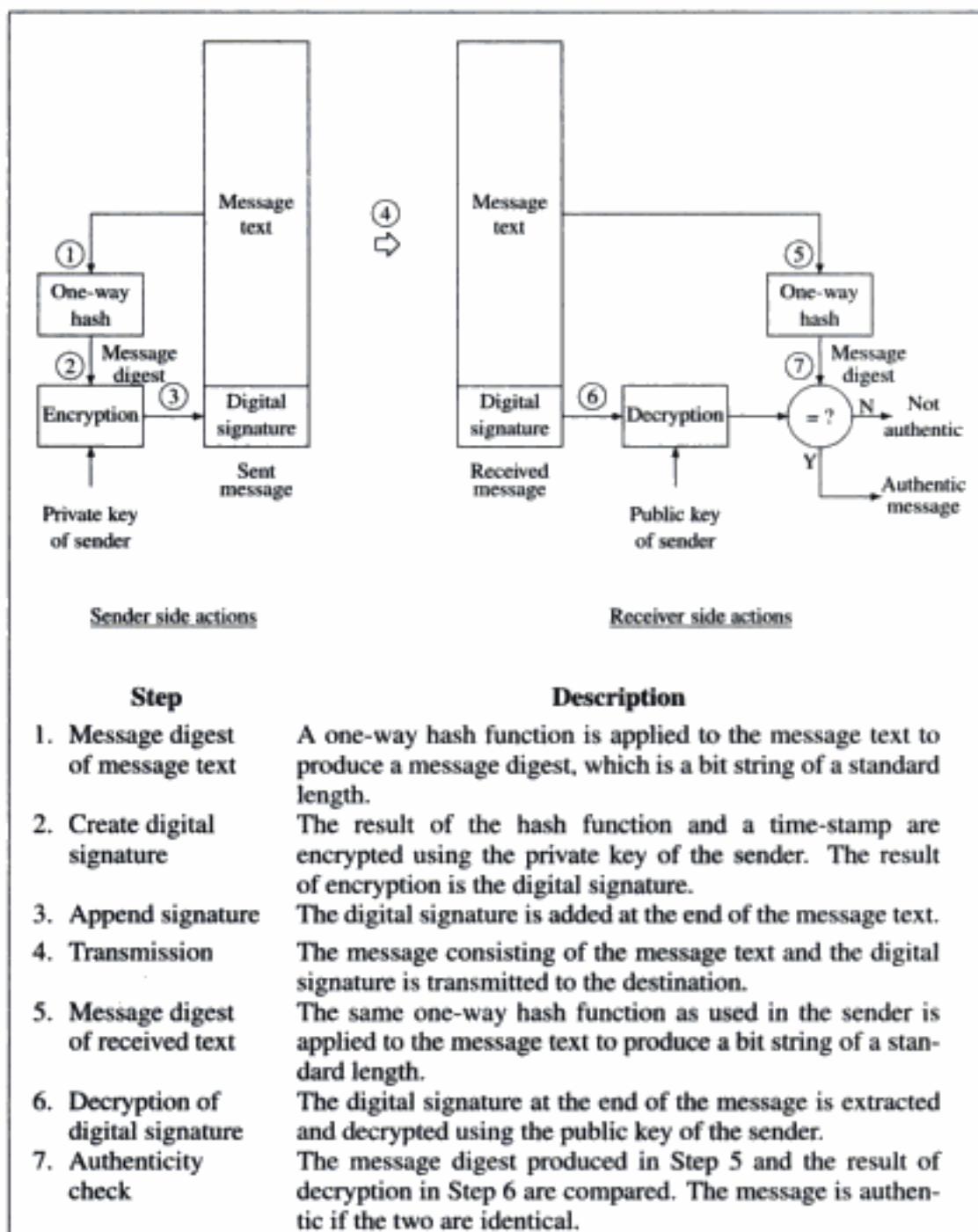


Fig. 20.5 Message authenticity through digital signature

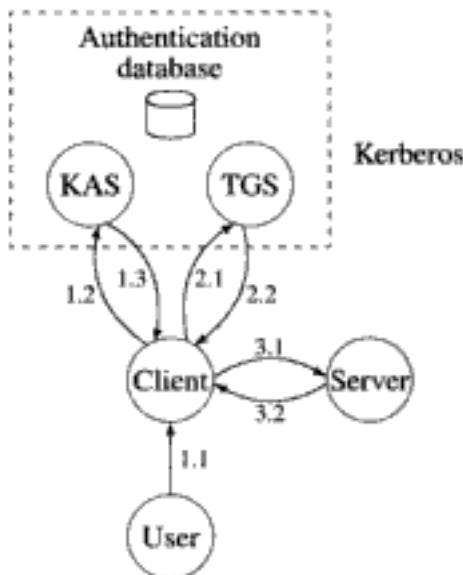


Fig. 20.6 Kerberos

The user submits his id and password to the client in Step 1.1. In Step 1.2, the client forwards the user id to KAS. It also encloses a nonce n_1 to authenticate KAS. This message is a request for a ticket to TGS. Note that the user's password is not passed to KAS. This fact avoids its exposure over the network. It also implies that authentication is not performed by KAS; it is actually performed by C in an interesting manner described later. In Step 1.3, KAS uses the user id U to retrieve V_U , the private key of U , from the authentication database and uses it to encrypt its reply to C. $SK_{U,TGS}$ is a session key for the session between the user and TGS, and T_{TGS} is a ticket for TGS encrypted with the key of TGS.

C has to decrypt the reply from KAS using the key V_U to obtain $SK_{U,TGS}$ and T_{TGS} . This step authenticates the user as follows: V_U , the private key of the user, satisfies the relation $V_U = f(\text{password})$, where f is a one-way function known to C. C obtains V_U by applying f to *password*. It now decrypts the reply received from KAS using this key. Decryption would be unsuccessful if the password supplied by the user is invalid. In this case C cannot extract T_{TGS} from the reply sent by KAS, so the user cannot use any services or resources in the system.

2. Obtaining a ticket for a server: When a user wishes to use a server, C obtains a ticket for the server using the following protocol:

- 2.1 $C \rightarrow TGS : \langle \text{Server_id} \rangle, T_{TGS}, AU, n_2$
- 2.2 $TGS \rightarrow C : E_{SK_{U,TGS}}(n_2, T_{\langle \text{Server_id} \rangle}, SK_{U,\langle \text{Server_id} \rangle}, \langle \text{Server_id} \rangle)$

where $\langle \text{Server_id} \rangle$ is the name of the server which C wishes to use, AU is an au-

thenticator, $SK_{U,Server_id}$ is a session key for the session between the client and the desired server, and T_{Server_id} is the ticket for the desired server encrypted using the key of the server. Before replying to the client, TGS verifies that the ticket presented by the client is valid, and that the request has originated in real time and within the validity period of the ticket.

3. *Obtaining a service:* When user U makes a service request, C generates an authenticator and a nonce and exchanges the following messages with the server:

- 3.1 $C \rightarrow \text{Server} : T_{<\text{Server_id}>} , AU, E_{SK_{U,<\text{Server_id}>}}(<\text{service request}>, n_3)$
- 3.2 $\text{Server} \rightarrow C : E_{SK_{U,<\text{Server_id}>}}(n_3)$

The server performs the service if it finds that the ticket is valid, and the request originated in real time and within the validity period of the ticket. It returns the nonce n_3 to the client so that the client can authenticate it, if it so desires.

20.4.2 Secure Sockets Layer (SSL)

SSL is a message security protocol providing authentication and communication privacy. It works on top of a reliable transport protocol such as the TCP/IP. Its successor, the *transport layer security* (TLS) protocol, is based on SSL 3.0; we discuss features that are common to both. When a client wishes to communicate with a server, the *SSL handshake protocol* is used before message exchange can start. It uses RSA public-key encryption to authenticate the server and optionally authenticate the client, and generates symmetric session keys for message exchange between the client and the server. Actual message exchange is performed through the *SSL record protocol*, which performs symmetric encryption of messages and transmits them over the network. Thus, message communication between the client and the server is reliable due to the transport protocol, secure due to authentication performed by the handshake protocol, and private due to encryption performed by the record protocol. Authenticity of data is ensured through a *digital signature* on a message. If mere integrity checking is desired, it is provided through a *message authentication code* (MAC). Higher level application protocols such as HTTP and FTP can be implemented on top of the SSL.

The SSL handshake protocol performs the following functions:

1. It performs authentication of the server.
2. It allows the client and the server to select the cryptographic algorithms to be used during the session from among RC2, RC4, DES, triple-DES, and a few other algorithms, and digital signature and hash algorithms from among DSA, MD5 and SHA-1.
3. It optionally performs authentication of the client.
4. It enables the client and the server to generate a shared secret, which would be used to generate the session keys.

A simplified overview of the SSL handshake protocol is as follows: The client sends a *client_hello* message to the server. This message contains a specification of the cryptographic and compression options, and a 28-byte random number that we will call n_{client} . The server responds with a *server_hello* message, which contains another random number n_{server} . Immediately following the *server_hello* message, the server sends its certificate. SSL has a list of certificate authorities (CAs) on the client side, using which it ensures that the server's certificate is from one of the listed CAs, and verifies the server's authenticity using public key cryptography. The server, if it so wishes, asks for the client's certificate and verifies the client's identity in a similar manner. Following this, the client sends the *encrypted premaster secret* message, which contains a 48 byte *premaster secret* string encrypted by the public key of the server.

Both client and server now generate a 48-byte *master secret* string from n_{client} , n_{server} , and *premaster secret*, using a standard one-way function. Use of n_{client} and n_{server} , which are randomly chosen values, ensures that the master secret would be different for different sessions between the same client-server pair. The *master secret* string is used to obtain four symmetric session keys using a standard algorithm. These keys are used as follows: Keys $k_{c \rightarrow s}^{crypt}$ and $k_{s \rightarrow c}^{crypt}$ are used for encryption and decryption of messages sent by the client to the server, and by the server to the client, respectively, and keys $k_{c \rightarrow s}^{mac}$ and $k_{s \rightarrow c}^{mac}$ are used to generate message authentication codes for messages sent by the client and by the server, respectively. Following key generation, both client and server send *finished* messages to one another. At this time, the SSL handshake protocol is complete.

Exchange of messages is performed by the SSL record protocol using the session keys generated during the handshake. The steps in sending a message m from the client to the server are as follows:

1. The client generates MAC_m , which is a message authentication code for message m , using the key $k_{c \rightarrow s}^{mac}$.
2. The pair $\langle m, MAC_m \rangle$ is encrypted using the key $k_{c \rightarrow s}^{crypt}$, and the encrypted string is sent to the server.
3. When the server receives the encrypted string, it decrypts it using the key $k_{c \rightarrow s}^{crypt}$ to obtain the pair $\langle m, MAC_m \rangle$. It accepts m if its MAC computed using the key $k_{c \rightarrow s}^{mac}$ matches MAC_m .

The SSL protocol could be subverted by a *man-in-the-middle* attack, where an intruder intercepts a client's messages to a server in the SSL handshake protocol and masquerades as the server in all subsequent message exchanges. It may analogously masquerade as the client and set up a secured SSL connection with the server. The client and server processes must take precautions to defeat the man-in-the-middle attack during the initial handshake. When the server provides its certificate to the client in the SSL handshake protocol, the client must verify that the distinguished name and IP address mentioned in the server's certificate match those of the server

Index

- signal* operation, 431
wait operation, 431
- Absolute loader, 228
Abstract view of OS, 2–5
Access
 descriptor, 375
 privilege, 357, 359, 375
 privileges, 381
 time, 526
Access control list (ACL), 318, 377–378
Access control matrix (ACM), 375
Access method, 308, 311, 562–563
 functions, 563
 mechanisms and policies, 522
 physical organization, 522
Active files table (AFT), 326
Address
 linked, 227
 load time, 227
 logical, 214
 physical, 215
 translated, 227
 translation, 214
Address space, 85
Address translation, 218–219, 243
 summary, 252, 255
Address translation unit (ATU), 523
Adversary, 362
AES, 372–373
Aging of requests, 148
Agreement protocol, 715
Amoeba operating system, 613, 630–631
 capabilities, 388–389
Andrew, 743–745
 file sharing semantics, 743–745
 path name resolution, 744
 scalability, 743
 shared name space, 743
Application layer, 655
Arrival pattern, 183
Arrival time, 144, 145, 183
Assembly language, 227–228
Asynchronous transfer mode (ATM), 648–650
 switch, 649
Atomic action, 339–341
 example, 341
 implementation, 340
Atomic transaction, 718
 abort, 718
Attacks on cryptographic systems, 367, 369
Authentication, 15, 358, 360, 365, 661, 751
 by knowledge, 358
 by possession, 358, 373
 data base, 359
 in distributed system, 762–768
 token, 358
 using passwords, 373
Authenticity of information, 360, 750
Authorization, 358
 data base, 359
Availability, 660
Awaiting I/O completion, 541, 548
- B+ tree, 350
Back-up, 23, 336
 incremental, 336
Backward recovery, 714
Bakery algorithm, 430–431
Banker's algorithm, 501–506
 feasible request, 501
 safe allocation state, 502
 safe request, 501
Bare machine, 603
Basic computation, 685–686
 example, 686
Batch, 52
Batch monitor, 53
Batch processing, 52, 76
 command processor, 55–56
 system, 49, 50, 52–56
 turn-around time, 54

- Belady's anomaly, 272
 Best fit algorithm, 203
 Binary program, 226, 231
 Binary semaphore, 432
 Binding
 - dynamic, 194, 280
 - memory, 193–194
 - resource, 11
 - static, 194
 Biometric feature, 358, 373
 Birthday attack, 760
 Bitstream, 527
 Block, *see* Logical record
 Block cipher, 367, 370
 Blocked state, 96, 491–495
 Blocking, 21
 Blocking factor, 560–562
 - definition, 560
 Blocking of records, 311, 559–562
 Booting, 600
 Boundary tag, 205–207
 Bounded Concurrency, 434
 Bounded wait, 407
 Bridge, 647
 Buddy system, 289
 Buffer pool, 415
 Buffering, 21
 Buffering of records, 311, 350, 555–559
 - peak requirement, 559
 Bus, 36
 Busy wait, 407–408, 415, 416
 Byzantine
 - agreement problem, 714
 - consensus problem, 715
 - fault, 714–715
 Cache
 - level, 36
 Cache memory, 35–36
 - cache block, 36
 - hit ratio, 36
 - spatial locality, 36
 - temporal locality, 36
 Caching, 21
 Calloc (of C), 196, 199
 Capability
 - based addressing, 384–385
 - object id, 383
 - object table (OT), 384
 - based computer system, 383–385
 - based protection, 382–390
 confinement problem, 390
 definition, 382
 in Amoeba, 388–389
 in software, 388–390
 protection of, 386–388
 - capability segment, 386
 - tagged architecture, 386
 protection of objects, 385–388
 revocation, 390
 sharing of objects, 385–388
 structure, 382
 subset capability, 382
 Capability list (C-list), 378–379
 Capacity planning, 186
 Certification authority, 760
 Challenge-response protocol, 758–760
 Chandy-Lamport algorithm, 677–680
 - example, 678
 - properties of recorded state, 678
 Channel
 - interprocess communication, 666
 Channel state, 675
 Child process, 86, 138
 Ciphertext, 366, 367
 Circuit switching, 650–651
 Circular look (C-look)scheduling, 551
 Circular wait, 485, 486, 498
 Client node, 727
 Client stub, 641
 Client-server computing, 631, 638–640
 Cluster, 627
 - Sun, 629–630
 - Windows, 629
 Coda, 744–745
 Collision, 647
 Command processor, 55–56
 Commit, 339, 718
 Commit flag, 340
 Commit processing, 340
 Communication, *see* Communication protocol, Interprocess communication, Interprocess messages, Message passing
 Communication primitive
 - blocking send, 469
 - non-blocking send, 469
 Communication processor (CP), 645
 Communication protocol
 - HDLC, 654
 - TCP, 655
 - UDP, 654, 655
 Communications management, 599

- Completion time, [144](#)
- Computation
 - migration, [637](#)
 - speed-up, [74](#)
- Computation migration, [638](#)
- Computation speed-up, [87](#), [91](#)
- Computational load, [698](#)
- Computational structure, [3](#), [7–10](#)
 - co-executing programs, [10](#)
 - sequence of single programs, [9](#)
 - single program, [9](#)
- Computer system, [32–41](#)
- Concurrency, [86](#), [91–92](#), [107](#)
- Concurrent
 - event, [668](#), [671](#)
 - processes, [86](#)
 - program, [86](#)
 - programming, [3](#), [7](#)
- Concurrent system, [421–423](#)
- Condition code, [32](#), [33](#)
- Condition variable, [447–452](#)
- Conditional critical region (CCR), [443–447](#)
 - await statement, [444](#)
 - implementation, [446–447](#)
 - usage example
 - bounded buffer, [445](#)
 - readers-writers, [446](#)
- Conditions for deadlock, [485–486](#)
- Confinement problem, [389](#)
- Confusion, [367](#), [369–373](#)
- Connection, [650](#)
- Connection strategies, [650](#)
- Consensus problem, [215](#)
- Context save, [94](#), [95](#), [100](#)
- Contiguous memory allocation, [211–213](#)
- Control block, [514](#)
- Control computation, [685–686](#)
 - example, [686](#)
- Control function, [659](#)
- Control register, [32](#)
- Control statement, [55](#)
- Control synchronization, [134](#), [404](#)
 - definition, [134](#)
- Conversation key, *see* Session key
- Copy-on-write, [281](#), [291](#)
- CPU, [32–35](#)
 - condition code (CC), [32](#)
 - fence register, [37](#)
 - interrupt mask, [33](#)
 - lower bound register (LBR), [32](#), [201–202](#)
 - memory protection information, [33](#)
- privileged instruction, [33](#)
- privileged mode, [33](#), [58](#)
- program counter (PC), [32](#)
- program status word (PSW), [33](#)
- register, [32](#), [93](#), [99–101](#), [106](#), [107](#), [118](#), [125](#), [131](#), [138](#)
- relocation register, [200](#)
- state, [34](#)
- switching, [32](#)
- upper bound register (UBR), [37](#), [201–202](#)
- user mode, [33](#)
- CPU scheduling, [13](#)
- CPU state, [93](#)
- CPU utilization, [48](#)
- CPU-I/O overlap
 - in a process, [555](#)
- CPU-bound program, [59](#)
- Critical section (CS), [404–431](#)
 - algorithmic implementation, [423–431](#)
 - Bakery algorithm, [430–431](#)
 - Dekker's algorithm, [426–427](#)
 - Peterson's algorithm, [427–428](#)
 - definition, [405](#)
 - example, [405](#)
 - implementation, [411–414](#)
 - implementation using semaphore, [432–434](#)
 - properties of implementation, [406–407](#)
- Cryptography, [366](#)
 - attacks on, [368](#)
 - chosen plaintext attack, [369](#)
 - ciphertext only attack, [368](#)
 - exhaustive attack, [368](#)
 - known plaintext attack, [368](#)
 - terms in, [367](#)
- CSCAN policy, [552](#)
- CSMA/CD, [648](#)
- Current directory, [314](#)
- Current locality, [246](#)
- Cycle, [488](#), [506](#)–[510](#), [512](#)
- Cycle stealing, [524](#)
- Cyclic redundancy checksum (CRC), [527–529](#)
- Cylinder, [530](#)
- Cylinder group, [346](#)
- Cylinder skewing, [532](#)
- Data
 - distributed, [637](#)
 - migration, [637](#), [638](#)
 - replication, [637](#)
- Data access synchronization, [131–134](#), [404](#)

- Data recovery, 713
 Data-link layer, 654
 Deadline, 71, 145
 Deadline estimation, 171
 Deadline overrun, 144, 145
 Deadlock, 20, 408, 419, *see also* Resource state modeling, 483–518
 conditions for, 485–486
 definition, 483
 in resource allocation, 484–489
 necessary conditions, 507–511
 processes in, 511
 resource class model, 506
 resource request model, 507
 sufficient conditions, 507–511
 MIMR system, 509–511
 MISR system, 508–509
 SIMR system, 509
 SISR system, 507–508
 Deadlock handling, 490–518
 Banker's algorithm, 501–506
 deadlock avoidance, 490, 500–506
 deadlock characterization, 506–513
 using WPGs, 512–513
 deadlock detection, 490–493
 algorithm, 492–495
 deadlock prevention, 490, 496–500
 all requests together, 498–499
 resource ranking, 499–500
 deadlock resolution, 495–496
 in Unix, 514–515
 MIMR system, 509–511
 MISR system, 508–509
 practical approach, 513–515
 SIMR system, 509
 SISR system, 507–508
 Deblocking actions, 560
 Degree of multiprogramming, 64
 Dekker's algorithm, 426–427
 Demand paging, 241–266
 Denial of service, 362, 751
 DES, 367, 371–372, 375
 cipher block chaining, 371
 Device controller, 33, 523
 Device driver, 550–551
 Unix, 565–566
 Diffusion, 367, 369–373
 Diffusion computation, 694–696
 engaging query, 694
 non-engaging query, 694
 Digital signature, 762
 Dijkstra, 431
 Dining philosophers problem, 419–421
 Direct access file, 309–310
 Directory
 as a file, 313
 current directory, 314
 hierarchy, 313–314
 home directory, 314
 multi-level structure, 313
 names cache, 350
 structures, 311–318
 Directory caching, 732
 Directory FCB pointer, 327
 Directory names cache, 742
 Dirty page, 255, 256, 287, 288
 Disk, *see also* RAID, 529–533
 access time, 530
 cache, 351
 cylinder, 530
 Cylinder skewing, 532
 data staggering, 530–533
 Head skewing, 532
 head switching, 531
 mirroring, 337–338
 mirroring in RAID, 534–536
 pack, 530
 record
 address, 530
 SCSI, 351
 tagged command queuing, 552–554
 sector, 530
 sector interleaving, 531–533
 seek time, 530
 status map, 324
 striping, 534
 throughput, 532–533
 track, 529
 Disk cache, 565
 unified, 564–565
 Disk scheduling, 21, 548, 551–554
 circular look (C-look), 551
 CSCAN, 552
 FCFS, 551
 in SCSI disk, 552–554
 look, 551
 SCAN, 551
 SSTF, 551
 summary, 552
 Disk space allocation, 322–325
 file map table, 324
 indexed allocation, 324–325

- linked allocation, [323–324](#)
- Dispatching, [94](#), [95](#), [101](#)
- Distributed computation paradigm, [637–644](#)
 - client–server computing, 638–640
 - remote evaluation, 642–644
 - remote procedure call, 640–642
- Distributed control, [75](#)
- Distributed control algorithm, [684–708](#)
 - control action, [684](#)
 - control data, [684](#)
 - correctness criteria, [686–688](#)
 - liveness, [686](#)
 - safety, [686](#)
 - distributed mutual exclusion, 688–693
 - edge chasing algorithm, [696–697](#)
 - operation of, [684–686](#)
- Distributed control function, [22](#)
- Distributed data, [717](#), 720–721
- Distributed deadlock detection, [685](#), 694–697
- Distributed deadlock handling, [693–698](#)
- Distributed deadlock prevention, 697–698
- Distributed election, [685](#)
- Distributed file system, [23](#), [725–745](#), *see also*
 - File caching, Scalability, Transparency Andrew, 743–745
 - availability, 731–733
 - cache manager, 737
 - cache validation, 737
 - client initiated, [739](#)
 - server initiated, [739](#)
 - Coda, 744–745
 - design issues, [725–728](#)
 - fault tolerance, [727](#), 730–735
 - file caching, 737–739
 - file migration, [726](#)
 - file server structure, [736](#)
 - file sharing semantics, 729–730
 - GPFS, [745–746](#)
 - hint, [736](#)
 - path name resolution, 731, [732](#)
 - performance, [727](#), [735–737](#)
 - remote mount, 744
 - session semantics, 729
 - stateless file server, 734–735
 - Sun NFS, 740–742
 - transaction semantics, 729
 - Unix semantics, 729
 - Windows, [747](#)
- Distributed mutual exclusion, [685](#), 688–693
 - fully distributed approach, 688–689
 - token-based approach, 689–693
- for ring topology, 690–691
- Raymond's algorithm, 691–693
- Distributed operating system, [22–23](#), [49](#), [51–52](#), [73–75](#), 628–661, *see also* Distributed file system
 - advantages, [73](#)
 - design issues, 659–661
 - load balancing, [660](#)
 - recovery, [660–661](#)
 - reliability, [660–661](#), [717–723](#)
 - resource allocation, [205–206](#)
 - scalability, 626, 639, [645](#)
 - security, 661, 750–769
- Distributed scheduling, [685](#), [698–701](#)
 - receiver initiated algorithm, [699](#)
 - sender initiated algorithm, [699](#)
 - stability, [699](#)
 - symmetrically initiated algorithm, 699–701
- Distributed system, *see also* Event precedence, Distributed control algorithm
 - authentication, [762–768](#)
 - communication, 626
 - computation speed-up, 626
 - cut of a system, 674–676
 - definition, [74](#)
 - fault tolerance, [717–721](#)
 - global state, [665](#)
 - incremental growth, 626
 - local state, [665](#)
 - model, [657–659](#)
 - logical model, [657](#)
 - physical model, [657](#)
 - node model, [627](#)
 - recovery, [710–722](#)
 - reliability, 626
 - resource sharing, 626
 - security, [750–769](#)
 - state, [665–673](#)
 - state recording, 673–680
- Distributed system security, [23](#)
- Distributed termination condition, [701](#)
- Distributed termination detection, [685](#), 701–703
- Distributed transaction, 720–721
- DMA, [37](#), [58](#), [75](#), [524](#)
 - bus mastering, [525](#)
 - first party DMA, [524](#)
 - third party DMA, [524](#)
- Domain name system (DNS), [632](#), 644
- Domino effect, [717](#)

- Duplicate request, 635
 Dynamic
 allocation, 193, 195
 binding, 194
 data, 198
 linking, 195
 loading, 195
 memory allocation, 194
 routing, 652
 Dynamic binding, 280
 Dynamic scheduling, 170
 Dynamically linked library (DLL), 195
 Edge
 allocation edge, 486
 AND edge, 512
 in distributed system model, 657–659
 OR edge, 512
 request edge, 486
 wait-for edge, 486
 Effective utilization, 5–7, 31–52
 EIA-232D, 654
 Election algorithm, 703–705
 Bully algorithm, 704–705
 for ring topologies, 703–704
 Elevator algorithm, 551
 Encryption, 365–373, 753–757
 asymmetric, 366
 distribution of keys, 756–758
 key, 366
 private key encryption, 753–754
 public key encryption, 754–756
 RSA encryption, 755
 session keys, 756
 symmetric, 366
 Error, 711
 Error detection, 529
 Ethernet, 647–648
 Event, 17, 102–106, 666
 handling, 102–106
 Event control block (ECB), 104–105, 175
 Event handling, 95, 600–601
 Event precedence, 667–668, 670, 671
 causal relationship, 667
 partial order, 667
 transitive precedence, 667
 Exokernel, 613
 Export list, 657
 Extended machine, 605
 Extensibility, 603
 External data representation (XDR), 643, 655
 External fragmentation, 204, 211, 213
 Failure, 711
 Fair share, 145
 Fault, 711
 amnesia, 712
 Byzantine, 714–715
 fail-stop, 712
 Fault tolerance, 23, 72, 337–341, 713, 717–721
 FCFS scheduling, 148–149
 performance analysis, 185
 Feasible request, 501
 Feasible schedule, 169
 Fence register, 37
 Field, 306
 FIPO page replacement, 268, 273–274
 Fifty percent rule, 206–207
 File, 19, 306, 514
 access, 326–331
 attribute, 306
 availability, 730
 caching, 726, 736–739
 cache validation, 739
 close, 307, 330–331
 control block, 319–322
 creation, 307
 deletion, 307
 internal id, 326
 map table, 324
 memory mapping, 282–284
 open, 307, 326–329
 operations, 306
 organization, 307–311
 primary copy, 732
 protection, 318–319, 375–382, 390–391
 read/write, 307
 recoverability, 730
 renaming, 307
 replication, 732–733
 robustness, 730
 sharing, 331–333
 sharing semantics, 331–333, 729–730
 File allocation table (FAT), 323–324
 File control block (FCB), 319–322, 326–331
 File PCB pointer, 322
 File management, 599
 File map table (FMT), 324
 File processing, 305
 File replication, 732
 File server, 727

- File sharing semantics, [331–333](#), 729–730
 session semantics, 729–730
 transaction semantics, 730
 Unix semantics, 729
- File system, [302–347](#), *see also* File access, File sharing, File protection, Atomic action, Disk space allocation, Disk mirroring
 actions at file close, [330–331](#)
 actions at file open, [326–329](#)
 back-up, [336](#)
 fault tolerance, [337–341](#)
 integrity, [334–336](#)
 interface with IOCS, [319–322](#)
 log-structured, [351–352](#)
 logical organization, [304](#)
 mechanisms and policies, [522](#)
 memory mapped files, [282–284](#)
 meta-data, [342](#)
 mounted files, [328–329](#)
 mounting, [317–318](#)
 operations, [320](#)
 performance, [350–352](#)
 recovery, [336–337](#)
 overhead, [336](#)
 reliability, [333–337](#), *see also* Recovery, Fault tolerance, Atomic actions
 Unix, [343–347](#)
 virtual file system, [341–343](#)
 Windows, [348–350](#)
- File transfer protocol (FTP), [638](#), [655](#)
- File type
 byte-stream file, [305](#)
 structured file, [305](#)
- First fit algorithm, [203](#)
- Forward recovery, [714](#)
- Free frames list, [241](#), [257](#)
- Free list, [202–206](#), [208](#), [209](#), [221–223](#), [225](#)
- Fully connected network, [647](#)
- Garbage collection, [389](#)
- Gateway, [645](#)
- Global clock, [666–667](#)
- GPPS, [745–746](#)
- Graceful degradation, [72](#)
- Graph
 RRAG, [486](#)
 allocation edge, [486](#)
 request edge, [486](#)
 WFG, [486–487](#)
- Graphical user interface, [2](#)
- guest OS, [607](#)
- Hash table, [262–263](#), [350](#), [566–567](#)
- Hashing function, [262](#)
- Head skewing, [532](#)
- Heap, [196–204](#)
- High level data link control (HDLC), [654](#)
- Hint, [736](#)
- Hit ratio
 in memory, [245](#)
 in TLB, [252](#)
- Hold-and-wait, [485](#), [486](#), [498](#)
- Home directory, [314](#)
- Host, [626](#)
- I/O, [37–38](#)
 asynchronous mode, [524](#)
 buffer, [555](#)
 channel
 multiplexor channel, [38](#)
 completion, [547–548](#)
 completion processing, [539–540](#)
 device, [525–537](#)
 address, [523](#)
 block mode, [525](#)
 character mode, [525](#)
 controller, [523](#)
 error detection, [527–528](#)
 random access, [525](#), [529–533](#)
 sequential, [525](#), [528–529](#)
- DMA mode, [37](#), [38](#)
- initiation, [484](#), [538–540](#), [546–547](#)
- instruction, [523](#)
- interrupt, [38](#)
 interrupt mode, [37](#), [38](#)
 organization, [523–525](#), [537](#)
 programmed mode, [37](#), [38](#)
 programming, [538–540](#)
 scatter-and-gather, [256](#)
 scheduler, [549](#)
 status, [538](#)
 status information, [522](#)
 synchronous mode, [524](#)
 time, [526](#)
 volume, [525](#)
- I/O device
 access time, [526](#)
 error correction, [528](#)
 error detection, [528](#)
- I/O devices, [513](#)
- I/O fixing, [256](#)

- I/O management, 599
 I/O programming
 advanced I/O programming, 554–562
 blocking, 559–562
 buffering, 555–559
 I/O-bound program, 59
 IBM 360/67, 273
 IBM 370, 286
 Idempotent computation, 637
 Idempotent operation, 340
 Immutable shared file, 331
 Impersonation, 362
 Import list, 657
 Incremental back-up, 336
 Index block, 325
 Index sequential file, 310–311
 Indexed allocation, 324–325
 Indivisible instruction, 412, 442
 Indivisible operation, 431
 definition, 411
 implementation, 411–414
 Inode, 343
 Input output control system (IOCS), 302–305
 difference with file system, 304
 library, 303
 operations, 320
 physical organization, 520
 Input output control system (IOCS), 520–522
 Integrity of information, 360, 750
 intel 80386, 265
 Intentions list, 339
 Inter-record gap, 529
 Internal fragmentation, 204, 211–212
 Internal id, 326
 Internet, 2
 Internet protocol (IP), 654
 Interprocess communication, 137
 Unix
 message queue, 477
 pipe, 476
 socket, 478
 Interprocess communication protocol, 631–637
 asynchronous protocol, 634
 at-least-once semantics, 633
 at-most-once semantics, 633
 blocking protocol, 634
 exactly-once semantics, 633, 635
 non-blocking protocol, 634
 RR protocol, 636–637
 RRA protocol, 634–636
 synchronous protocol, 634
 Interprocess message, 90, 466–481, 514
 asymmetric naming, 468
 buffering, 470–472
 control block (IMCB), 470
 delivery, 472–474
 exceptional condition, 470
 indirect naming, 474
 symmetric naming, 468
 Interrupt, 17, 38, 41
 action, 40–41
 classes, 39
 code, 39, 42
 external, 41
 I/O, 39, 41
 mask, 33, 39–40
 priority, 38
 processing, 42–45
 nested, 43
 program, 39, 41
 software, 39, 41, 45
 timer, 39
 vector, 40–41, 608
 Interrupt handling, 600
 Intruder, 15, 362, 365, 751
 IOCS, 21, 302, *see* Physical IOCS, Access method
 IP address, 632
 IPC semantics, 633, 661, *see also* Interprocess communication protocol
 ISO protocol layers, 654–655
 operation, 653
 ISO reference model, 653–654
 Java, 455–456
 remote method invocation, 643–644
 Job, 47
 scheduling, 158
 step, 47
 Kerberos, 764–766
 authenticator, 764
 ticket, 764
 Kernel, 2, 32, 610–612
 configuration and installation, 614–615
 interrupt driven operation, 601
 memory allocation, 221–226
 synchronization support, 442–443
 Key
 encryption, 366
 in record, 306
 Key distribution center (KDC), 756
 Knot, 508

- LAN, *see* Local area network (LAN)
 Latency, [530](#), [657](#)
 rotational, [552](#)–[554](#)
 Layered OS design, 604–607
 Light weight process (LWP), [123](#)–[125](#)
 Link, [316](#)
 hard link, [347](#)
 symbolic link, [347](#)
 Linked allocation, [323](#)–[324](#)
 Linked list, [202](#)–[203](#), [206](#), [208](#), [223](#), [225](#), [233](#)
 Linker, [226](#)
 definition, [195](#)
 Linking, [226](#)–[230](#)
 definition, [229](#)
 dynamic, [195](#)
 dynamic linking, [229](#)–[230](#)
 ENTRY statement, [229](#)
 external reference, [229](#)
 EXTRN statement, [229](#)
 program relocation, [227](#)–[228](#)
 definition, [228](#)
 public definition, [229](#)
 static linking, [229](#)–[230](#)
 Linux, [246](#), [615](#)
 active list, [180](#)
 child process, [126](#)
 clone system call, [126](#)
 exhausted list, [180](#)
 file processing, [568](#)–[569](#)
 anticipatory scheduler, [569](#)
 deadline scheduler, [569](#)
 file system, [348](#)
 advisory lock, [348](#)
 block group, [348](#)
 ext2, [348](#)
 lease, [348](#)
 mandatory lock, [348](#)
 futex, [126](#), [458](#)
 loadable kernel modules, [392](#)
 O(1) scheduling, [180](#)
 password security, [375](#)
 process, [125](#)–[128](#)
 process synchronization, [457](#)–[458](#)
 scheduling, [178](#)–[180](#)
 security, [391](#)–[392](#)
 security enhanced linux, [392](#)
 thread, [125](#)–[128](#)
 virtual memory, [289](#)–[290](#)
 file backed region, [289](#)
 inactive laundered page, [289](#)
 zero filled region, [289](#)
 Little's formula, [184](#)
 Livelock, [426](#)
 livelock, [419](#)
 Liveness, [687](#)
 Load balancing, [698](#)
 Loader, [226](#)
 absolute loader, [228](#)
 definition, [195](#)
 relocating loader, [228](#)
 Loading
 dynamic, [195](#)
 Local area network (LAN), 644–646
 Local clock, [666](#)
 Locality of reference, [246](#)–[248](#)
 Lock variable, [412](#)–[413](#), [442](#), [443](#)
 Log, [717](#)
 Log-structured file system, [351](#)–[352](#)
 Logical address, [214](#)
 Logical address space, [215](#), [217](#)
 Logical clock, 669–671
 synchronization, 669
 Logical device, 542–543
 Logical organization, 4–5, [214](#)
 Logical record, [560](#)
 Long-term scheduling, [160](#)
 Long-term scheduling, [157](#)–[158](#)
 Look scheduling, [551](#)
 Lower bound register (LBR), [37](#), [201](#)–[202](#)
 LRU page replacement, [268](#)–[270](#), [272](#)–[273](#)
 Mach operating system
 microkernel, [612](#)–[613](#), [616](#)
 Magnetic tape, [528](#)
 streaming tape, [529](#)
 Mailbox, [474](#)–[476](#)
 Malloc (of C), [196](#), [199](#)
 Mean response time, [145](#)
 Mean turnaround time, [145](#)
 Mechanism, [601](#)–[602](#)
 authentication, [373](#)
 capability, [382](#)
 communication mechanism, [611](#)
 context save, [161](#)
 dispatching, [159](#)
 event handling, [548](#)
 I/O mechanism, [611](#)
 IOCS mechanism, [303](#), [304](#), [319](#), [522](#)
 memory management mechanism, [611](#)
 paging mechanism, [266](#)
 process migration, [706](#)
 protection mechanism, [360](#)

- scheduling mechanism, 160, 611
 security mechanism, 360
 signal, 90–91, 119–120, 137–139
 swapping, 213
 Medium-term scheduling, 160
 Medium-term scheduling, 157–158
 Memory, 513
 allocation
 noncontiguous, 18
 fragmentation, 18
 reuse, 202–207
 Memory allocation, 13–14, *see also* First fit algorithm, Best fit algorithm, Virtual memory
 as a binding, 193
 contiguous allocation, 211–213
 dynamic allocation, 194, 195
 free list, 202–206, 208, 209, 221–223, 225
 kernel memory allocation, 221–226
 noncontiguous allocation, 213–220
 static allocation, 194, 195
 to a process, 198–200
 Memory allocators
 Buddy system allocator, 208–209
 lazy buddy allocator, 223–224
 McKusick–Karels allocator, 222–223
 powers-of-two allocator, 209–210
 slab allocator, 224–226
 Memory bound register, 37
 Memory compaction, 207, 212
 Memory controller, 33, 523
 Memory fragmentation, 204, 211–212, 226
 definition, 204
 external fragmentation, 211, 213
 internal fragmentation, 211–212
 Memory hierarchy, 35–37
 Memory management, 191–235, 599
 Memory management unit (MMU), 18, 33, 35, 219, 238, 244, 250, 254, 255, 291
 Memory map, 53
 Memory mapped files, 282–284
 Memory protection, 37, 58, 201–202, 216, 254–255
 bound registers, 201–202
 Memory utilization factor, 210
 Memoryless property, 183
 Merging free memory, 205
 Message, *see also* Interprocess message
 acknowledgment, 631
 orphan, 716
 queue, 477
 replay, 758–759
 retransmission, 631
 security, 753–760
 switching, 651
 Message authentication code (MAC), 761–762
 Message digest, 760
 Message passing, 20, 90, 137, 466–481
 issues, 468
 Unix, 476
 Message security, 661
 Microkernel, 612–614
 Migration
 computation, 698–701, 729
 file, 726, 729
 process, 698–701
 volume, 743
 MIMR system, 509–511
 MISR system, 508–509
 Mitchell–Merritt algorithm, 696–697
 Modern operating system, 75–76
 Modify bit, 273, 288
 Monitor, 447–454
 signal statement, 448
 wait statement, 448
 condition variable, 447–452
 disk scheduler case study, 452–455
 in Java, 455–456
 mutual exclusion, 447–448
 usage example
 producers–consumers, 450, 451
 semaphore, 448, 449
 Motorola 68030, 266
 Mount point, 317
 MS-DOS, 235
 Multi-site transaction, *see* Distributed transaction
 Multi-threading, 3, 7
 MULTICS, 343
 protection domain, 381–382
 segmentation, 293
 Multilevel scheduling, 163–166
 CTSS, 165–166
 Multiple instance resource, 507
 Multiple request, 507
 Multiprocessor operating system, 21
 Multiprogramming, 50, 52, 56–65
 architectural support, 57
 degree of multiprogramming, 59, 64
 kernel functions, 56, 58–65
 program classification, 59–60

- program mix, [59–60](#)
- program priority, [60–64](#)
- scheduling, [59–64](#)
- schematic, [57](#)
- system, [49](#)
- Mutable files
 - multiple image, [332](#)
 - single image, [332](#)
- Mutex, *see also* Binary semaphore
- Mutual authentication, [759–760](#)
- Mutual exclusion, [133](#). *see also* Critical section, Distributed mutual exclusion, [406, 422](#)
 - algorithmic approaches, [408](#)
 - concurrent programming constructs, [408](#)
 - critical section, [404–431](#)
 - in monitors, [447–448](#)
 - software primitives, [408](#)
 - using conditional critical region (CCR), [443–447](#)
 - using semaphore, [432–434](#)
- Name server, [632](#), [641](#), [657](#), [705–706](#)
- Name space, [743](#)
- Naming
 - freedom, [311](#)
 - in communication, [632–633](#)
 - in file system, [312–313](#), [728](#)
 - in message passing, [468–469](#)
 - of processes, *see* Process id
- Necessary conditions for deadlock, [507–511](#)
- MIMR system, [509–511](#)
- MISR system, [508–509](#)
- SIMR system, [509](#)
- SISR system, [507–508](#)
- Nested transaction, [722](#)
- Network
 - connection, [645](#)
 - file system, [740–742](#)
 - fully connected, [647](#)
 - layer, [654](#)
 - local area (LAN), [644–646](#)
 - operating system, [627–628](#)
 - partially connected, [647](#)
 - partition, [647](#)
 - protocol, [645](#)
 - ring, [647](#)
 - routing, [645](#)
 - star, [647](#)
 - topology, [645–647](#)
 - type, [645](#)
- wide area (WAN), [644](#)
- Network bandwidth, [657](#)
- Network protocol, [653](#)
 - ISO reference model, [653–654](#)
- Networking, [599](#), [644](#)
- New (of Pascal), [200](#)
- Newcastle connection, [628](#)
- Next fit algorithm, [203](#)
- Node, [626](#)
- Non-preemptible server, [148](#)
- Non-preemptive scheduling, [148–150](#)
- Non-shareable resource, [486](#)
- Non-shareable resources, [496](#)
- Nonce, [759](#)
- Noncontiguous memory allocation, [213–218](#)
- Object module, [196](#), [199](#), [226](#), [231](#)
- One-way function, [367](#), [368](#)
- One-way hash function, [375](#)
- Open system, [625](#), [762](#)
- Operating system
 - and effective utilization, [5–7](#)
 - classes of, [49–52](#)
 - batch processing, [50](#)
 - distributed, [51](#)
 - multiprogramming, [50](#)
 - real time, [51](#)
 - time sharing, [51](#)
 - functions, [599](#)
 - event handling, [95](#)
 - goals, [5–7](#)
 - mechanism, [601–602](#)
 - operation of, [7–10](#)
 - overhead, [6](#)
 - policy, [601–602](#)
 - resident part, [601](#)
 - resource management, [11–15](#)
 - structure, [21](#)
 - kernel-based, [604](#)
 - kernel-based OS, [610–612](#)
 - layered, [4](#), [604–607](#)
 - microkernel-based, [604](#)
 - microkernel-based OS, [612–614](#)
 - monolithic, [603–604](#)
 - virtual machine OS, [607–608](#)
 - transient part, [601](#)
 - user convenience
 - good service, [50](#)
 - necessity, [50](#)
 - resource sharing, [50](#)
- Optimal page replacement, [268](#)

- Origin
specification, 226
- Orphan message, 716
- Overhead, 6, 42, 72
scheduling, 42
- Overlay, 239
- Overlay structured program, 195
- Packet switching, 651
- Page, 215, 239
fault, 242–250, 252, 263
characteristic, 249
frame, 240
optimal page size, 249
reference string, 267
replacement, 245, 246, 255–256
table, 241, 257
misc info, 242, 257
modified bit, 242, 257, 259
prot info, 242, 257
ref info, 242, 257, 259
valid bit, 242
traffic, 245
- Page cache, 565
- Page fault characteristic, 248
- Page frame, 217
- Page reference string, 267
- Page replacement policies, 266–274
clock algorithm, 273
FIFO approximation, 273–274
FIFO replacement, 268
LRU approximation, 272–273
LRU replacement, 268–270
optimal replacement, 268
second-chance algorithm, 273–275
stack property, 270–271
Unix clock algorithm, 286
- Page table, 218, 242, 243
inverted page table, 260–263
multi-level page table, 265–266
two-level page table, 263–265
- Page table address register (PTAR), 254
- Page table size register, 255
- Page-in operation, 244
- Page-out operation, 244
- Paging, 215, 217–219, 239, *see also* Demand paging, Page replacement policies
address translation, 217–219, 241–246, 250–254
address translation buffer, 250–253
current locality, 246
demand paging, 241–266
free frames list, 257
I/O fixing of pages, 256–257
I/O operations, 256–257
scatter-and-gather I/O, 256
inverted page table, 260–263
memory protection, 254–255
in logical address space, 255
multi-level page table, 265–266
overview, 240–241
page reference string, 267
page table, 257
sharing of pages, 255, 279–281
two-level page table, 263–265
Intel 80386, 265
- Paging hardware, 250–256
- Paging software, 257–281
- Parallelism, 91–92
- Parbegin–Parend, 424
- Parent process, 138
- Parity bit, 527–529
- Partial order, 667
- Partially connected network, 647
- Partition, 12
- Password, 373
aging, 374
encryption, 374
security, 364–365
- Password security, 374–375
- Path
in RRAG, 488
in WFG, 487–488
- Path name, 314–315
absolute path name, 315
relative path name, 315
resolution, 327–329, 731, 732, 742, 744
- Performance analysis, 181–186
FCFS scheduling, 185
mathematical modeling, 182
queueing theory, 182
response ratio scheduling, 185
round-robin scheduling, 185
simulation, 182
SRN scheduling, 185
- Peterson's algorithm, 427–428
- Phantom deadlock, 518, 687, 688, 693–694
- Physical address, 215
- Physical address space, 215
- Physical IOCS, 540–568
data structures, 543–545
design aims, 540

- I/O completion, [547–548](#)
- I/O control block (IOCB), [543](#)
- I/O initiation, [546–547](#)
- I/O queue, [543](#)
- library, [545–546](#)
- logical device, [542](#)
- logical device table (LDT), [543](#)
- mechanisms and policies, [522](#)
- optimization of device performance, [548](#)
- physical device table (PDT), [543](#)
- physical organization, [522](#)
- Physical layer, [654](#)
- Physical organization, [4–5, 214, 522](#)
- Physical record, [560](#)
- Pipe, [476](#)
- Plaintext, [366, 367](#)
- Platter, [529](#)
- Poisson distribution, [183](#)
- Policy, [601–602](#)
- Pool
 - based allocation, [12–13, 216](#)
 - of buffers, [415](#)
- Port, [643](#)
- Portability, [21, 602, 612](#)
- Power management, [167–168](#)
- Preemption, [13, 61, 145, 146, 498](#)
- Preemptive scheduling, [76, 151–156](#)
- Presentation layer, [655](#)
- Primary copy, [337](#)
- Print server, [15](#)
- Priority, [87, 145, 147–148](#)
 - definition, [60](#)
 - dynamic, [147](#)
 - static, [147](#)
 - variation of, [147, 176–177, 181](#)
- Priority based scheduling, [161–166](#)
- Priority inheritance protocol, [408](#)
- Priority inversion, [408](#)
- Priority-based scheduling, [59](#)
- Privacy, [360, 750](#)
- Private key, [753–755, 764–765](#)
- Private key encryption, [753–754](#)
- Privileged instruction, [33](#)
- Privileged mode, [33, 40, 43](#)
- Process, [18, 46, 47, 84–125](#)
 - advantages of child processes, [87](#)
 - and program, [84–86](#)
 - blocked state, [96](#)
 - communication, [90](#)
 - concurrent processes, [86](#)
 - creation, [87–91, 102](#)
- daemon, [115](#)
- data sharing, [90](#)
- definition, [93](#)
- environment, [93–94](#)
- fault, [712](#)
- id, [93](#)
- independent processes, [131](#)
- interacting processes, [90–91, 130–137](#)
 - control synchronization, [134–136](#)
 - data access synchronization, [131–134](#)
 - definition, [131](#)
 - summary of interaction, [90](#)
- interprocess communication, [466–481](#)
- interprocess message, [90](#)
- Linux process, [125–128](#)
- main process, [86](#)
- OS view, [92–106](#)
- precedence, [168](#)
- programmer view, [87–91](#)
- ready state, [96](#)
- recovery, [713](#)
- running state, [96](#)
- scheduling, [94–95, 101, 160–168](#)
- signal, [90–91](#)
- stack, [93](#)
- state, [92, 95–99](#)
 - definition, [95](#)
- state transition, [95–98](#)
- status, [87–91](#)
- suspend state, [98](#)
- switching, [101](#)
- synchronization, [90](#)
- terminated state, [96](#)
- termination, [87–91, 102–103](#)
- tree, [86](#)
- Unix, [115–123](#)
- Windows, [128–130](#)
- Process control block (PCB), [93, 94, 99–100, 102, 104, 111, 115, 125, 138, 139, 160–162, 165, 221](#)
- Process environment, [93–94](#)
- Process management, [599](#)
- Process migration, [638, 698–701, 706–707](#)
- Process precedence graph, [168–169](#)
- Process scheduling
 - Linux, [178–180](#)
 - Windows, [181](#)
- Process state, [92, 95](#)
- Process synchronization, [20, 403–463, see also Race condition, Critical section, Semaphores, Conditional critical region,](#)

- Monitor
 - classic problems, 414–421
 - dining philosophers, 419–421
 - producers–consumers, 415–417
 - readers–writers, 417–419
 - control synchronization, 134–136, 404
 - data access synchronization, 131–134, 404
- Producers–consumers problem, 415–417, 435–437, 445
 - indivisible operations, 417
 - outline, 416, 417
 - statement of problem, 415–417
 - using CCR, 445
 - using monitor, 450–452
 - using semaphore, 435–437
- Program, 46, 47
 - address space, 85
 - counter, 32, 40
 - kernel program, 2
 - loading, 200–201
 - mix, 59–60
 - multi-segment, 220, 221
 - non-kernel program, 2
 - preemption, 43, 49
 - definition, 61
 - priority, 49, 60–64
 - relocation, 200–201
 - start address, 227
- Program controlled dynamic data, 196, 199
- Program counter, 33
- Program execution
 - definition, 92
- Program mix, 59
- Program status word (PSW), 32–37, 99–101
- Programmed I/O, 37
- Progress condition, 407
- Protection, 15–16, 358, *see also* Capability, Memory protection, File protection
 - access control list, 377–378
 - access control matrix, 375
 - access privilege, 357, 359, 375
 - Access privileges, 381
 - capability list, 377–379
 - domain, 379–382, 391
 - change of, 380–382, 391
 - MULTICS, 381–382
 - Unix, 380–381, 390–391
- goals, 360
- granularity, 376
- information, 313
- mechanism, 360
- policy, 360
- ring, 381
- structure, 377–382, 390–391
- Protocol, *see* Communication protocol
- Proximity region, 246
- PSW, 93
- Public key, 754–757
 - distribution, 757
- Public key certificate, 761
- Public key encryption, 754–756
- Queue
 - I/O, 543
 - scheduling, 104, 148, 152, 154, 158, 163, 183
- Queuing theory, 182
- Quorum algorithm, 719–720
- Race condition, 132–134, 404
 - definition, 132
 - example, 132–133
 - in control synchronization, 409–411
 - processes containing race condition, 133
- RAID, 533–536, 731
 - level 0 (disk striping), 534
 - level 1 (disk mirroring), 534–536
 - level 2, 536
 - level 3 (bit interleaving), 536
 - level 4 (block interleaving), 536
- Random events, 183–184
- Raymond algorithm, 691–693
- Read–write head, 529
- Readers–writers problem, 417–419, 438–440, 444–446
 - outline, 419
 - statement of problem, 417–419
 - using CCR, 444–446
 - using semaphores, 438–440
- Ready list, 159, 161, 163, 165, 166
- Ready queue, *see* Ready list
- Ready state, 26
- Real time application, 70–73
 - deadline, 21
 - definition, 70
 - example, 21, 86
 - response requirement, 21
- Real time operating system, 49, 51, 52, 70–73
 - fault tolerance, 72
 - graceful degradation, 72
- Real time scheduling, 49, 168–174
 - dynamic scheduling, 170

- EDF scheduling, [172–173](#)
- feasible schedule, [169](#)
- priority-based scheduling, [170](#)
- process precedence, [168–169](#)
- rate monotonic scheduling, [173–174](#)
- static scheduling, [169](#)
- Real time system
 - hard, [71](#)
 - soft, [71](#)
- Receive operation, [137, 466–481](#)
- Record, [306](#)
 - logical record, [560](#)
 - physical record, [560](#)
- Recovery, [23, 710–722](#)
 - backward recovery, [714](#)
 - definition, [711](#)
 - forward recovery, [714](#)
- Redo log, [717](#)
- Reentrant program, [231–233](#)
- Reference bit, [259, 272–274](#)
- Register
 - CPU, [99–101, 106, 107, 118, 125, 131, 138](#)
- Reliability, [73–74](#)
 - file system, [333–337](#)
 - in communication protocol, [631–632](#)
 - in distributed operating system, [660](#)
 - in distributed system, [626, 658, 665](#)
 - in network, [646](#)
- Relocating loader, [200](#)
- Relocating logic, [231](#)
- Relocation
 - definition, [228](#)
 - of program, [200](#)
 - register, [200](#)
- Remote
 - data access, [638](#)
 - file processing, [727](#)
- Remote data access, [637](#)
- Remote evaluation, [631, 639, 642–644](#)
- Remote procedure call (RPC), [75, 631, 639–642](#)
- Replicated data, [717–720](#)
- Request, [144](#)
 - reordering, [146](#)
- Request queue, [148, 152](#)
- Resiliency, [713, 721–723](#)
- Resolution of deadlock, [495–496](#)
- Resource
 - allocation, [11–15](#)
 - dynamic, [11](#)
 - partitioning, [11–13](#)
 - pool based allocation, [11–13](#)
 - static, [11](#)
- knot, [510](#)
- preemption, [13](#)
- ranking, [499–500](#)
- sharing, [23](#)
- state modeling, [486–487](#)
 - matrix model, [489](#)
 - RRAG, [486](#)
 - WFG, [486](#)
- Resource allocation, [484–489](#)
 - events, [484](#)
- Resource request and allocation graph (RRAG), [486](#)
- Resource sharing, [76](#)
- Response ratio, [145, 150](#)
- Response ratio scheduling, [150](#)
 - performance analysis, [185](#)
- Response time, [48, 49, 144, 145](#)
 - variation with time slice, [153–154](#)
- Reuse of memory, [212](#)
- Ricart–Agrawala algorithm, [688–689](#)
- Ring network, [647](#)
- Roll-back, [713](#)
- Root directory, [313](#)
- Rotational latency, [530](#)
- Round-robin scheduling, [49, 66, 151–154, 163](#)
 - performance analysis, [185](#)
- Routing, [652–653](#)
- RS-232C, [654](#)
- Running state, [96, 491–495](#)
- Safe allocation state, [502](#)
- Safe request, [501](#)
- Safety, [687](#)
- Saved PSW information, [40–43, 45](#)
- Scalability, [727, 735, 736, 740](#)
- Schedule length, [145, 146](#)
- Scheduler, [43](#)
- Scheduling, [13, 41, 43, 59, 65–68, 94, 95, 101, 143–188, 600](#)
 - concepts, [145](#)
 - CTSS, [165–166](#)
 - disk, [548, 551–554](#)
 - EDF scheduling, [172–173](#)
 - events
 - arrival, [144](#)
 - completion, [144](#)
 - preemption, [144](#)
 - scheduling, [144](#)

- fair share scheduling, 166–167, 177–178
 FCFS scheduling, 148–149
 fundamentals, 143–148
 HRN scheduling, 150, 151
 in batch processing, 159
 in multiprogramming, 59–65, 159
 in time sharing, 65–68, 159
 job scheduling, 158
 LCN scheduling, 151, 155–156
 Linux, 178–180
 long-term scheduling, 158
 lottery scheduling, 167
 medium-term scheduling, 158
 Multilevel adaptive scheduling, 165–166
 non-preemptive, 148–150
 performance analysis, 181–186
 preemptive, 151–156
 process scheduling, 160–168
 multilevel scheduling, 163–166
 multiprogramming, 161–162
 real time scheduling, 168–174
 time sharing, 163
 Unix, 175–178
 Windows, 181
 queue, 104, 148, 162, 154, 158, 163, 183
 rate monotonic scheduling, 173–174
 real time, 49
 real time scheduling, 168–174
 request, 144
 response ratio scheduling, 150
 round-robin scheduling, 151–154
 short-term scheduling, 158–168
 SRN scheduling, 149–150
 STG scheduling, 151, 156
 terms, 145
 using resource consumption information, 154–156
 Scheduling list, 66, 67, 69, 70
 Scheduling overhead, 42
 Scheduling policy, 18
 Secondary copy, 337
 Secrecy, 360, 750
 Sector, 530
 Sector interleaving, 531–533
 Secure sockets layer (SSL), 766–768
 Security, 15–16, 358, 750–769
 authentication, 358
 token, 358
 authenticity, 360, 750
 authorization, 358
 birthday attack, 760
 distributed system, 23
 encryption, 359
 goals, 360, 361
 in distributed systems, 750–769
 integrity, 360, 750
 mechanism, 360, 364, 751
 password, 359
 policy, 360, 364, 751
 privacy, 360, 750
 secrecy, 360, 750
 Security attack, 362–364
 denial of service, 362
 impersonation, 362
 masquerading, 362
 Security attacks, 752–753
 eavesdropping, 252
 impersonation, 752
 man-in-the-middle attack, 261
 message replay, 752, 758–759
 message tampering, 752
 Security threats
 denial of service, 751
 leakage, 751
 stealing, 751
 tampering, 751
 Seek time, 530
 Segment, 216
 Segment link table (SLT), 292
 Segment table, 219, 292
 Segment table address register (STAR), 292
 Segmentation, 215, 219–220, 239, 291–295
 fragmentation, 294
 MULTICS, 293
 with paging, 220–221
 Segmentation with paging, 294–295
 fragmentation, 294
 Self-relocating program, 231–232
 Semantic gap, 604, 605
 Semaphore, 431–443
 binary semaphore, 432
 definition, 431
 implementation, 440–443
 architectural assistance, 443
 hybrid, 443
 kernel-level, 443
 lock variable, 443
 pseudo-code, 420, 442
 user-level, 443
 Unix, 456–457
 usage example
 mutual exclusion, 432–434

- producers-consumers, [435–437](#)
- readers-writers, [438–441](#)
- use for bounded concurrency, [434–435](#)
- use for mutual exclusion, [432](#)
- use for signaling, [435](#)
- Windows, [459](#)
- Send operation, [137, 466–481](#)
- Sequential file, [308–309](#)
- Server, [638](#)
- Server node, [727](#)
- Server stub, [641](#)
- Service pattern, [183](#)
- Service time, [144, 145, 184](#)
- Session key, [756–758](#)
 - distribution, [757](#)
- Session layer, [655](#)
- Session semantics, [729–730](#)
- Setuid bit, [381, 391](#)
- Sharing
 - CPU, [13](#)
 - data, [90, 131–134](#)
 - disk, [14](#)
 - file, [331–333, 729–730](#)
 - memory, [13](#)
 - page, [255](#)
 - program, [232](#)
 - resource
 - dynamic, [13](#)
 - static, [13](#)
 - segment, [294](#)
- Sharing of pages, [279–281](#)
- Short-term scheduling, [160](#)
- Short-term scheduling, [157–159](#)
- Signaling, [409, 435](#)
- Signals, [90, 119–120, 137–139](#)
- SIMR system, [509](#)
- Simulation, [182, 491](#)
- Single instance resource, [507](#)
- Single request, [507](#)
- SISR system, [507–508, 693](#)
- Site, [626](#)
- Snapshot of concurrent system, [422](#)
 - example, [423, 434, 437, 450, 452](#)
 - pictorial conventions, [422](#)
- Socket, [478–480](#)
- Software capability, [388–390](#)
- Software interrupt, [45, 137](#)
- Solaris
 - kernel thread, [123](#)
 - light weight process (LWP), [123–125](#)
 - thread, [123–125](#)
- user thread, [123](#)
- Spooling, [49, 514](#)
- SRN scheduling, [149–150](#)
 - performance analysis, [185](#)
- SSTF policy, [552](#)
- Stable property, [680](#)
- Stable storage, [337–338](#)
- Stack, [92, 93, 106, 112, 196–198, 200](#)
- Stack property, [270–271](#)
- Star network, [647](#)
- Starvation, [148, 150, 406, 417, 434, 552](#)
 - in priority-based scheduling, [147](#)
- State, [23](#)
 - global, [665](#)
 - local, [665](#)
 - of channel, [675](#)
 - of distributed system, [665–673](#)
 - consistent recording, [673](#)
 - of process, [95–99](#)
 - transition, [95–98](#)
- Stateless file server, [732, 734–735](#)
- Static
 - memory allocation, [194](#)
- Static allocation, [193, 195](#)
- Static binding, [194](#)
- Static data, [198](#)
- Static scheduling, [169](#)
- STG scheduling, [156](#)
- Store-and-forward, [652, 657](#)
- Stream cipher, [367, 370–371](#)
- Subrequest, [47, 144](#)
- Substitution cipher, [368](#)
- Sufficient conditions for deadlock, [507–511](#)
- Sun
 - cluster, [629–630](#)
- Sun NFS, [740–742](#)
 - architecture, [741](#)
 - file handle, [740](#)
 - file sharing semantics, [742](#)
 - Mount protocol, [740–741](#)
 - path name resolution, [742](#)
- Sun RPC, [643](#)
 - port, [643](#)
- Sun Spare, [266](#)
- Superpage, [253–254](#)
 - demotion, [254](#)
 - promotion, [254](#)
- Supervisor, *see also* Kernel
- Suspend state, [98](#)
- Swap instruction, [414](#)
- Swap space, [242, 256, 258–259](#)

- Swapped state, 98
 Swapping, 14, 70, 76, 213
 schematic, 70
 Synchronization
 data access synchronization, 133
 of clocks, 669, 671
 of processes, *see* Process synchronization
 System area, 32
 System call, 42, 45–46
 communication related, 46
 file related, 46
 information related, 46
 program related, 46
 resource related, 46

 Tagged command queuing, 552–554
 TCP/IP protocol, 655–657
 Terminated state, 96
 Test-and-set instruction, 413
 THE operating system, 606–607
 Thrashing, 248–249
 definition, 248
 Thread, 7, 18, 92, 106–125
 advantages, 107–108
 coding, 108
 definition, 106
 hybrid threads, 113–114
 kernel-level threads, 109–110
 Linux threads, 125–128
 Posix, 111
 Solaris threads, 123–125
 stack, 106
 state, 107
 user-level threads, 110–113
 Windows thread, 128
 Thread control block (TCB), 106, 109
 Threads library, 110–113
 Throughput, 48, 59, 145, 146, 149, 150
 Time, 665
 Time sharing, 51, 52, 65–70
 memory management, 70
 scheduling, 65–68
 time slicing, 66
 operation, 67
 Time sharing system, 49
 Time slice, 49, 66
 variation, 147
 Time slicing, 66–69, 76
 Time-stamp, 668–671, 764
 Timeout, 631
 Timer interrupt, 66

 Timing diagram, 668
 Token ring, 648
 Total order, 667
 Track, 529
 Transaction semantics, 729, 730
 Transfer time, 526
 Translation look-aside buffer, 250–253
 Transparency, 49, 75, 659, 726–729
 location independence, 726
 location transparency, 659, 726
 Transport control protocol (TCP), 655
 Transport layer, 655
 Trojan horse, 15, 363
 Turn around time, 48, 144, 148
 Turn-around time, 48
 Turnaround time, 145
 Two phase commit, 720–721

 Undo log, 718
 Unified disk cache, 565
 Unix
 exec, 116, 285, 380, 391
 fork, 116, 285
 proc structure, 115
 semget, 456
 semop, 456
 setuid, 365, 380–381, 391
 u area, 115
 architecture, 615–616
 buffer cache, 566–568
 clock algorithm, 286
 copy-on-write, 285
 daemon, 115
 deadlock handling, 514–515
 device driver, 565–566
 directory mounting, 346–347
 disk quota, 344
 event address, 175
 file allocation, 345–346
 file allocation table, 345
 file descriptor, 343
 file processing, 566–568
 file sharing semantics, 345
 file structure, 343
 file system, 343–347
 free list, 346
 inode, 343
 interprocess message, 476–478
 interrupt processing, 118
 kernel memory allocation, 222–226
 kernel process, 116

- main process, 116
- message queue, 477
- page fixing, 287
- pageout daemon, 286–288
- password security, 375
- pipe, 476
- process, 115–123
 - creation, 116
 - kernel interruptible mode, 175
 - kernel non-interruptible mode, 175
 - kernel running state, 122
 - priority, 175
 - termination, 117
 - user mode, 175
 - user running state, 122
- process scheduling, 175–178
 - nice value, 176
- process state transitions, 121–123
- protection domain, 380–381, 391
- semaphore, 456–463
- signals, 119–120
- socket, 478–480
- super block, 346
- swap space, 284–285
- swapping, 288
- symbolic link, 347
- system call, 118
- user process, 115
- virtual memory, 284–288
- zombie process, 117
- Unix semantics, 729
- Upper bound register (UBR), 37, 201–202
- User area, 32
- User datagram protocol (UDP), 655
- User group, 376
- User interface, 8–10, 599
- User service, 54, 65
 - in batch processing, 54
 - in multiprogramming, 58–59
 - in time sharing, 65
- Utilization factor, 183
- Vector clock, 671–672
- Vernam cipher, 370
- Virtual address, *see* Logical address
- Virtual circuit, 652
- Virtual devices, 53, 497
- Virtual file system, 341–343, 740
- Virtual machine, 15, 607
- Virtual machine OS, 607–608
- Virtual memory, 18, 195, 237–240, *see also* Paging, Segmentation, Working set
 - definition, 238
 - demand loading, 239
 - effective memory access time, 245, 251–252
 - example of operation, 239–240
 - hit ratio, 252
- Virtual memory handler, 238, 257–260
 - data structures, 257
 - functions, 258
 - protection, 259
- Virtual resource, 14–15
- Virus, 16, 363
- VM/370, 608
- Vnode, 342
- Volume, 349, 743
- Volume migration, 743
- Wait-for graph (WFG), 486–487, 693–697
 - AND edge, 512
 - OR edge, 512
- Wait-or-die, 697
- WAN, *see* Wide area network (WAN)
- Weighted turn around, 144, 148
- Weighted turnaround, 145, 150
- Wide area network (WAN), 644
- Windows
 - access token, 393
 - architecture, 618–619
 - cache manager, 570
 - cluster server, 629
 - copy-on-write, 291
 - DFS namespaces, 747
 - Discretionary access control list, 393
 - DLL, 619
 - environment subsystems, 619
 - event, 459
 - event pair, 481
 - executive, 618
 - file processing, 570–571
 - file system, 348–350
 - master file table, 349
 - hardware abstraction layer (HAL), 618
 - impersonation, 393
 - kernel, 618
 - LPC, 480–481
 - message passing, 480–481
 - mutex, 459
 - port, 480
 - process, 128–130

- creation, 128–129
- process synchronization, 458–459
- real time threads, 181
- scheduling, 181
- security, 392–394
- security descriptor, 393
- security identifier, 392
- semaphore, 459
- synchronization objects, 458–459
- thread
 - state transitions, 129–130
- variable priority threads, 181
- virtual memory, 290–291
 - shared pages, 291
- VM manager, 570
- Windows thread, 128
- Windows NT, 246
- Working set, 276–278
 - definition, 276
 - implementation, 278–279
 - window, 276
- Worm, 363, 364
- Worst fit algorithm, 233
- Wound-or-wait, 698

2
Second Edition

OPERATING SYSTEMS

A CONCEPT-BASED APPROACH

Salient Features of the Second Edition

- ▲ Entirely reworked and restructured chapters on
 - Processes and Threads
 - Scheduling
 - File Systems
 - Virtual Memory
 - Process Synchronization
 - Security and Protection
- ▲ New schematic diagrams provide design overviews
- ▲ Tables summarize terms and techniques
- ▲ Case studies describe design of Unix, Linux and Windows

Dedicated web site: <http://mhhe.com/dhamdhere/os>

Visit us at : www.tatamcgrawhill.com

ISBN 0-07-061194-7



9 780070 611948



Tata McGraw-Hill

Material chroniony prawem autorskim