

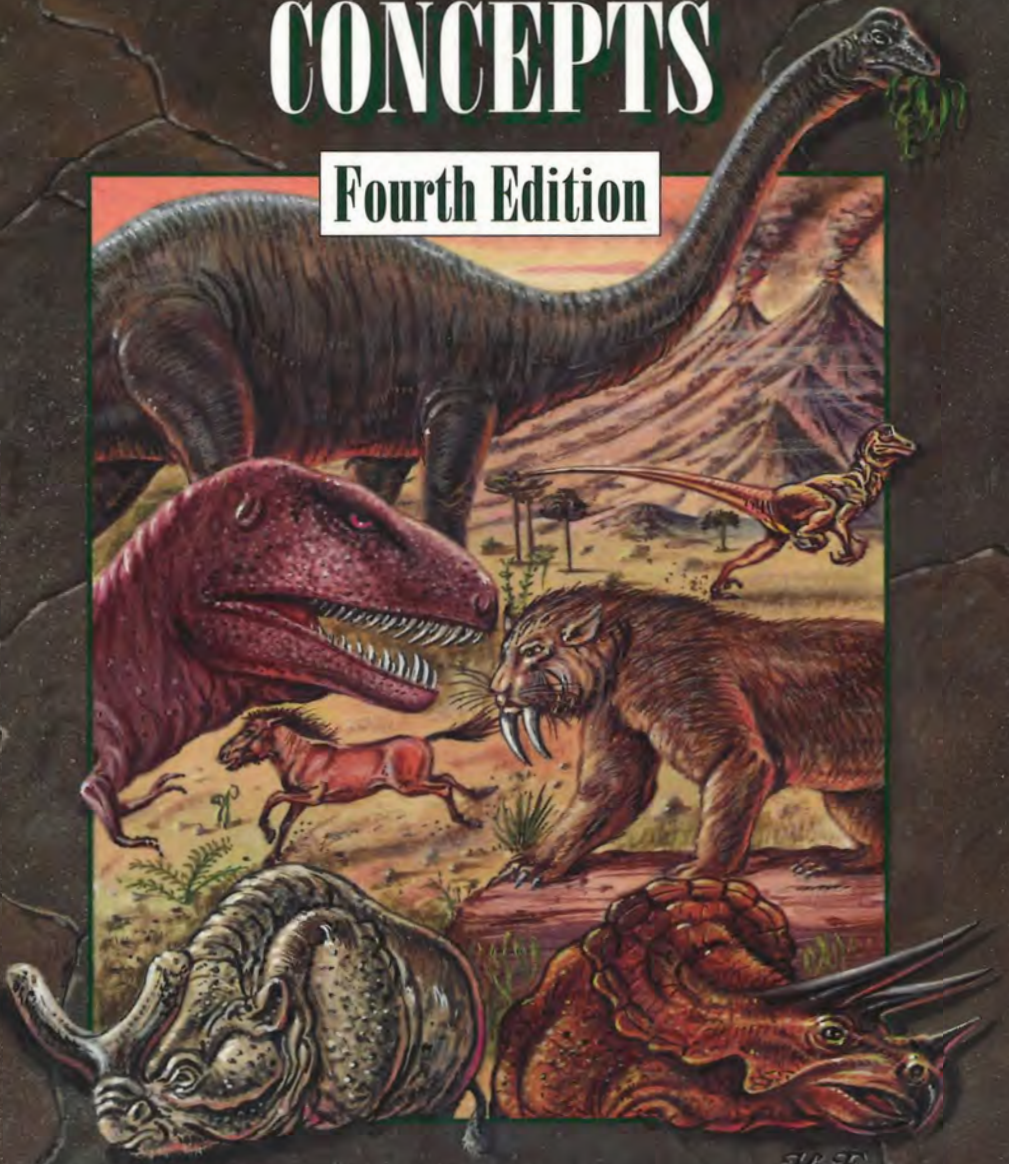
Kirkland & Ellis LLP Library



5 0628 01266978 1

OPERATING SYSTEM CONCEPTS

Fourth Edition



SILBERSCHATZ
GALVIN

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

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

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

Library of Congress Cataloging-in-Publication Data

Silberschatz, Abraham.

Operating system concepts / Abraham Silberschatz, Peter B. Galvin.
p. cm.

Includes bibliographical references and index.

ISBN 0-201-50480-4

1. Operating systems (Computers) I. Galvin, Peter B. II. Title.

QA76.76.063S5583 1994

005.4'3--dc20

93-24415

CIP

Reprinted with corrections January, 1995

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

Copyright © 1994 by Addison-Wesley Publishing Company, Inc.

All rights reserved. No part of this publication may be reproduced, stored in a retrieval system, or transmitted, in any form or by any means, electronic, mechanical, photocopying, recording, or otherwise, without the prior written permission of the publisher. Printed in the United States of America.

ISBN 0-201-50480-4

4 5 6 7 8 9 10-MA-99 98 97 96 95

PREFACE

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

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

Prerequisites

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

CHAPTER 1

INTRODUCTION



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

To understand what operating systems are, we must first understand how they have developed. In this chapter, we trace the development of operating systems from the first hands-on systems to current multiprogrammed and time-shared systems. As we move through the various stages, we see how the components of operating systems evolved as natural solutions to problems in early computer systems. Understanding the reasons behind the development of operating systems gives us an appreciation for what tasks an operating system does and how it does them.

1.1 ■ What Is an Operating System?

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

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

4 ■ Chapter 1: Introduction

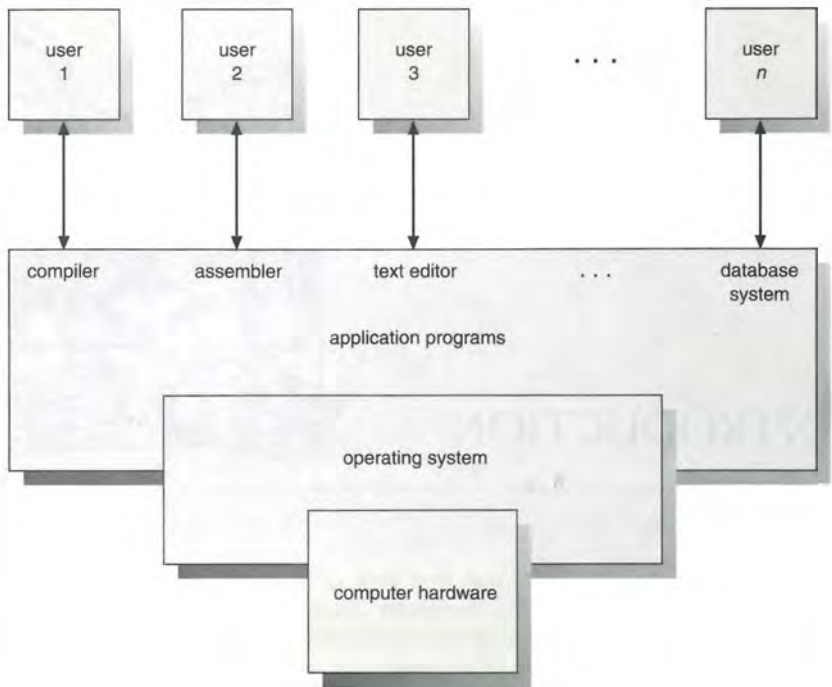


Figure 1.1 Abstract view of the components of a computer system.

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

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

We can view an operating system as a *resource allocator*. A computer system has many resources (hardware and software) that may be required to solve a problem: CPU time, memory space, file storage space, I/O devices, and so on. The operating system acts as the manager of these resources and allocates them to specific programs and users as necessary for their

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

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

In general, however, there is no completely adequate definition of an operating system. Operating systems exist because they are a reasonable way to solve the problem of creating a usable computing system. The fundamental goal of computer systems is to execute user programs and to make solving user problems easier. Toward this goal, computer hardware is constructed. Since bare hardware alone is not particularly easy to use, applications programs are developed. These various programs require certain common operations, such as those controlling the I/O devices. The common functions of controlling and allocating resources are then brought together into one piece of software: the operating system.

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

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

A secondary goal is *efficient operation of the computer system*. This goal is particularly important for large, shared multiuser systems. These systems are typically expensive, so it is desirable to make them as efficient as possible. These two goals, convenience and efficiency, are sometimes contradictory. In the past, efficiency considerations were often more important than convenience. Thus, much of operating-system theory concentrates on optimal use of computing resources.

To see what operating systems are and what operating systems do, let us consider how they have developed over the last 30 years. By tracing that evolution, we can identify the common elements of operating systems and see how and why these systems have developed as they have.

increment takes place in the internal register, the value of A differs in the various storage systems. The value of A becomes the same only after the new value of A is written back to the magnetic disk.

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

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

In a distributed environment, the situation becomes even more complex. In such an environment, several copies (replicas) of the same file can be kept on different computers that are distributed in space. Since the various replicas may be accessed and updated concurrently, we must ensure that when a replica is updated in one place, then all other replicas are brought up to date as soon as possible. There is a variety of ways for achieving this guarantee, as will be discussed in Chapter 17.

2.5 ■ Hardware Protection

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

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

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

storage) during execution. Because main memory is too small to accommodate all data and programs, and its data are lost when power is lost, the computer system must provide *secondary storage* to back up main memory. Most modern computer systems use disks as the principle on-line storage medium, for both programs and data. Most programs — including compilers, assemblers, sort routines, editors, and formatters — are stored on a disk until loaded into memory, and then use the disk as both the source and destination of their processing. Hence, the proper management of disk storage is of central importance to a computer system.

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

- Free-space management
- Storage allocation
- Disk scheduling

Because secondary storage is used frequently, it must be used efficiently. The entire speed of operation of a computer may hinge on the disk subsystem and the algorithms which manipulate it. Techniques for secondary-storage management will be discussed in detail in Chapter 12.

3.1.4 I/O System Management

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

- A buffer-caching system
- A general device-driver interface
- Drivers for specific hardware devices

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

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

3.1.5 File Management

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

physical organization. Each medium is controlled by a device, such as a disk drive or tape drive, with its own unique characteristics. These properties include speed, capacity, data transfer rate, and access method (sequential or random access method).

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

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

The operating system implements the abstract concept of a file by managing mass storage media, such as tapes and disks, and the devices which control them. Also, files are normally organized into directories to ease their use. Finally, when multiple users have access to files, it may be desirable to control by whom and in what ways files may be accessed.

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

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

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

3.1.6 Protection System

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

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

usage statistics. Usage statistics may be a valuable tool for researchers who wish to reconfigure the system to improve computing services.

- **Protection:** The owners of information stored in a multiuser computer system may want to control its use. When several disjoint processes execute concurrently, it should not be possible for one process to interfere with the others, or with the operating system itself. Protection involves ensuring that all access to system resources is controlled. *Security* of the system from outsiders is also important. Such security starts with each user having to authenticate himself or herself to the system, usually by means of a password, to be allowed access to the resources. It extends to defending external I/O devices, including modems and network adapters, from invalid access attempts, and to recording all such connections for detection of breakins. If a system is to be protected and secure, precautions must be instituted throughout it. A chain is only as strong as its weakest link.

3.3 ■ System Calls

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

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

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

As an example of how system calls are used, consider writing a simple program to read data from one file and to copy them to another file. The first input that the program will need is the names of the two files: the input file and the output file. These names can be specified in many ways, depending on the operating-system design. One approach is for the program to ask the user for the names of the two files. In an interactive system, this approach will require a sequence of system calls, first to write a prompting message on the screen, and then to read from the keyboard the characters that define the two files. Another approach, frequently used for batch systems, is to specify the names of the files with control cards. In

this case, there must be a mechanism for passing these parameters from the control cards to the executing program. On mouse-based and icon-based systems, a menu of file names is usually displayed in a window. The user can then use the mouse to select the source name, and a window can be opened for the destination name to be specified.

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

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

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

Most users never see this level of detail, however. The run-time support system for most programming languages provides a much simpler interface. For example, a *write* statement in Pascal or FORTRAN probably is compiled into a call to a run-time support routine that issues the necessary system calls, checks for errors, and finally returns to the user program. Thus, most of the details of the operating-system interface are hidden from the programmer by the compiler and by the run-time support package.

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

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

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

3.3.1 Process and Job Control

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

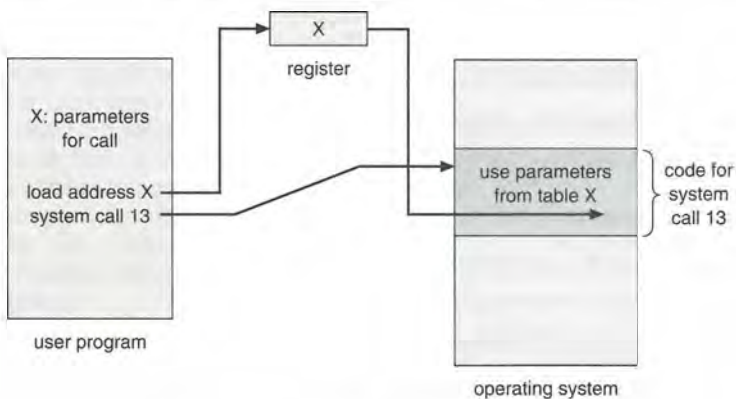


Figure 3.1 Passing of parameters as a table.

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

Figure 3.2 Types of system calls.

CHAPTER 4

PROCESSES



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

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

4.1 ■ Process Concept

One hindrance to the discussion of operating systems is the question of what to call all the CPU activities. A **batch system executes jobs**, whereas a **time-shared system has user programs, or tasks**. Even on a single-user system, such as MS-DOS and Macintosh OS, a user may be able to run several programs at one time: one interactive and several batch programs.

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

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

4.1.1 The Process

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

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

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

Although two processes may be associated with the same program, they are nevertheless considered two separate execution sequences. For instance, several users may be running copies of the mail program, or the same user may invoke many copies of the editor program. Each of these is

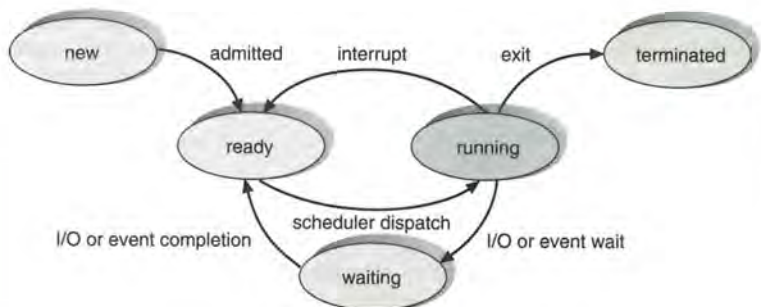


Figure 4.1 Diagram of process state.

a separate process, and, although the text sections are equivalent, the data sections will vary. It is also common to have a process that spawns many processes as it runs. This issue will be further discussed in Section 4.4.

4.1.2 Process State

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

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

These names are arbitrary, and vary between operating systems. The states that they represent are found on all systems, however. Certain operating systems also distinguish among more finely delineating process states. It is important to realize that only one process can be running on any processor at any instant. Many processes may be ready and waiting, however. The state diagram corresponding to these states is presented in Figure 4.1.

4.1.3 Process Control Block

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

- **Process state:** The state may be new, ready, running, waiting, halted, and so on.
- **Program counter:** The counter indicates the address of the next instruction to be executed for this process.
- **CPU registers:** The registers vary in number and type, depending on the computer architecture. They include accumulators, index registers, stack pointers, and general-purpose registers, plus any condition-code information. Along with the program counter, this state information must be saved when an interrupt occurs, to allow the process to be continued correctly afterward (Figure 4.3).

8.1.2 Dynamic Loading

To obtain better memory-space utilization, we can use *dynamic loading*. With dynamic loading, a routine is not loaded until it is called. All routines are kept on disk in a relocatable load format. The main program is loaded into memory and is executed. When a routine needs to call another routine, the calling routine first checks to see whether the other routine has been loaded. If it has not been, the relocatable linking loader is called to load the desired routine into memory and to update the program's address tables to reflect this change. Then, control is passed to the newly loaded routine.

The advantage of dynamic loading is that an unused routine is never loaded. This scheme is particularly useful when large amounts of code are needed to handle infrequently occurring cases, such as error routines. In this case, although the total program size may be large, the portion that is actually used (and hence actually loaded) may be much smaller.

Dynamic loading does not require special support from the operating system. It is the responsibility of the users to design their programs to take advantage of such a scheme. Operating systems may help the programmer, however, by providing library routines to implement dynamic loading.

8.1.3 Dynamic Linking

Notice that Figure 8.1 also shows *dynamically linked* libraries. Most operating systems support only *static linking*, in which system language libraries are treated like any other object module and are combined by the loader into the binary program image. The concept of dynamic linking is similar to that of dynamic loading. Rather than loading being postponed until execution time, linking is postponed. This feature is usually used with system libraries, such as language subroutine libraries. Without this facility, all programs on a system need to have a copy of their language library (or at least the routines referenced by the program) included in the executable image. This requirement wastes both disk space and main memory. With dynamic linking, a *stub* is included in the image for each library-routine reference. This stub is a small piece of code that indicates how to locate the appropriate memory-resident library routine, or how to load the library if the routine is not already present.

When this stub is executed, it checks to see whether the needed routine is already in memory. If the routine is not in memory, the program loads it into memory. Either way, the stub replaces itself with the address of the routine, and executes the routine. Thus, the next time that that code segment is reached, the library routine is executed directly, incurring no cost for dynamic linking. Under this scheme, all processes that use a language library execute only one copy of the library code.

This feature can be extended to library updates (such as bug fixes). A library may be replaced by a new version, and all programs that reference the library will automatically use the new version. Without dynamic linking, all such programs would need to be relinked to gain access to the new library. So that programs will not accidentally execute new, incompatible versions of libraries, version information is included in both the program and the library. More than one version of a library may be loaded into memory, and each program uses its version information to decide which copy of the library to use. Minor changes retain the same version number, whereas major changes increment the version number. Thus, only programs that are compiled with the new library version are affected by the incompatible changes incorporated in it. Other programs linked before the new library was installed will continue using the older library. This system is also known as *shared libraries*.

Unlike dynamic loading, dynamic linking generally requires some help from the operating system. If the processes in memory are protected from one another (Section 8.4.1), then the operating system is the only entity that can check to see whether the needed routine is in another processes' memory space, and can allow multiple processes to access the same memory addresses. This concept is expanded when used in conjunction with paging, as discussed in Section 8.5.5.

8.1.4 Overlays

In our discussion so far, the entire program and data of a process must be in physical memory for the process to execute. The size of a process is limited to the size of physical memory. So that a process can be larger than the amount of memory allocated to it, a technique called *overlays* is sometimes used. The idea of overlays is to keep in memory only those instructions and data that are needed at any given time. When other instructions are needed, they are loaded into space that was occupied previously by instructions that are no longer needed.

As an example, consider a two-pass assembler. During pass 1, it constructs a symbol table; then, during pass 2, it generates machine-language code. We may be able to partition such an assembler into pass 1 code, pass 2 code, the symbol table, and common support routines used by both pass 1 and pass 2. Assume that the sizes of these components are as follows (K stands for "kilobyte," which is 1024 bytes):

Pass 1	70K
Pass 2	80K
Symbol table	20K
Common routines	30K

To load everything at once, we would require 200K of memory. If only

2. **Data-link layer:** The data-link layer is responsible for handling the *frames*, or fixed-length parts of packets, including any error detection and recovery that occurred in the physical layer.
3. **Network layer:** The network layer is responsible for providing connections and for routing packets in the communication network, including handling the address of outgoing packets, decoding the address of incoming packets, and maintaining routing information for proper response to changing load levels. Routers work at this layer.
4. **Transport layer:** The transport layer is responsible for low-level access to the network and for transfer of messages between the clients, including partitioning messages into packets, maintaining packet order, controlling flow, and generating physical addresses.
5. **Session layer:** The session layer is responsible for implementing sessions, or process-to-process communications protocols. Typically, these protocols are the actual communications for remote logins, and for file and mail transfers.
6. **Presentation layer:** The presentation layer is responsible for resolving the differences in formats among the various sites in the network, including character conversions, and half duplex–full duplex modes (character echoing).
7. **Application layer:** The application layer is responsible for interacting directly with the users. This layer deals with file transfer, remote-login protocols, and electronic mail, as well as with schemas for distributed databases.

Figure 15.11 summarizes the *ISO protocol stack* (a set of cooperating protocols), showing the physical flow of data. As mentioned, logically each layer of a protocol stack communicates with the equivalent layer on other systems. But physically, a message starts at or above the application layer, and is passed through each lower level in turn. Each of these layers may modify the message and include message header data for the equivalent layer on the receiving side. Finally, the message makes it to the data-network layer and is transferred as one or more packets (Figure 15.12). These data are received by the data link layer of the target system, and the message is moved up through the protocol stack, being analyzed, modified, and stripped of headers as it progresses. It finally reaches the application layer for use by the receiving process.

The ISO model formalizes some of the earlier work done in network protocols, but was developed in late 1970s and is not yet in widespread use. Part of the basis for ISO is the more timeworn and widely used protocol stack developed under UNIX for use in the Arpanet (which became the Internet.)