UNIT-I

OPERATING SYSTEMS OVERVIEW R13

OPERATING SYSTEM STRUCTURE

PROCESSES

OPERATING SYSTEM:

Definition:

An operating system is a program that manages a computer's hardware. It also provides a basis for application programs and acts as an intermediary between the computer user and the computer hardware.

or

An operating system is a group of computer programs that coordinates all the activities among computer hardware devices. It is the first program loaded into the computer by a boot program and remains in memory at all times.

An amazing aspect of operating systems is how they vary in accomplishing these tasks. Mainframe operating systems are designed primarily to optimize utilization of hardware.

Personal computer (PC) operating systems support complex games, business applications, and everything in between.

Operating systems for mobile computers provide an environment in which a user can easily interface with the computer to execute programs.

Thus, some operating systems are designed to be convenient, others to be efficient, and others to be some combination of the two.

OPERATING SYSTEM FUNCTIONS:

The basic functions of an operating system are:

- Booting the computer
- Performs basic computer tasks eg managing the various peripheral devices eg mouse, keyboard
- Provides a user interface, e.g. command line, graphical user interface (GUI)
- Handles system resources such as computer's memory and sharing of the central processing unit (CPU) time by various applications or peripheral devices
- Provides file management which refers to the way that the operating system manipulates, stores, retrieves and saves data.

Booting the computer:

The process of starting or restarting the computer is known as booting. A cold boot is when you turn on a computer that has been turned off completely. A warm boot is the process of using the operating system to restart the computer.

Performs basic computer tasks:

The operating system performs basic computer tasks, such as managing the various peripheral devices such as the mouse, keyboard and printers. For example, most operating systems now are plug and play which means a device such as a printer will automatically be detected and configured without any user intervention.

Provides a user interface:

A user interacts with software through the user interface. The two main types of user interfaces are:

Command Line Interface (CLI):

With a command line interface, the user interacts with the operating system by typing commands to perform specific tasks. An example of a command line interface is DOS (disk operating system).

Graphical User Interface (GUI):

With a graphical user interface, the user interacts with the operating system by using a mouse to access windows, icons, and menus. An example of a graphical user interface is Windows Vista or Windows 7.

The operating system is responsible for providing a consistent application program interface (API) which is important as it allows a software developer to write an application on one computer and know that it will run on another computer of the same type even if the amount of memory or amount of storage is different on the two machines.

Handles system resources:

The operating system also handles system resources such as the computer's memory and sharing of the central processing unit (CPU) time by various applications or peripheral devices.

Programs and input methods are constantly competing for the attention of the CPU and demand memory, storage and input/output bandwidth. The operating system ensures that each application gets the necessary resources it needs in order to maximize the functionality of the overall system.

Provides file management:

The operating system also handles the organization and tracking of files and directories (folders) saved or retrieved from a computer disk. The file management system allows the user to perform such tasks as creating files and directories, renaming files, copying and moving files, and deleting files.

The operating system keeps track of where files are located on the hard drive through the type of file system. The type two main types of file system are File Allocation table (FAT) or New Technology File system (NTFS).

http://hsc.csu.edu.au/info_tech/compulsory/os/4014/basic_functions.htm

OPERATING SYSTEM STRUCTURE:

An operating system provides the environment within which programs are executed. Internally, operating systems vary greatly in their makeup, since they are organized along many different lines.

One of the most important aspects of operating systems is the ability to multiprogram. A single program cannot, in general, keep either the CPU or the I/O devices busy at all times. Single users frequently have multiple programs running. Multiprogramming increases CPU utilization by organizing jobs (code and data) so that the CPU always has one to execute.

The idea is as follows: The operating system keeps several jobs in memory simultaneously. Since, in general, main memory is too small to accommodate all jobs; the jobs are kept initially on the disk in the job pool. This pool consists of all processes residing on disk awaiting allocation of main memory.

The set of jobs in memory can be a subset of the jobs kept in the job pool. The operating system picks and begins to execute one of the jobs in memory. Eventually, the job may have to wait for some task, such as an I/O operation, to complete.

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

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

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

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

A time-shared operating system uses CPU scheduling and multiprogramming to provide each user with a small portion of a time-shared computer; each user has at least one separate program in memory. A program loaded into memory and executing is called a ***process***. When a process executes, it typically executes for only a short time before it either finishes or needs to perform I/O.

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

Time sharing and multiprogramming require that several jobs be kept simultaneously in memory. If several jobs are ready to be brought into memory, and if there is not enough room for all of them, then the system must choose among them. Making this decision involves ***job scheduling***.

When the operating system selects a job from the job pool, it loads that job into memory for execution. Having several programs in memory at the same time requires some form of memory management. In addition, if several jobs are ready to run at the same time, the system must choose which job will run first. Making this decision is **CPU scheduling**.

Finally, running multiple jobs concurrently requires that their ability to affect one another be limited in all phases of the operating system, including process scheduling, disk storage, and memory management. We discuss these considerations throughout the text.

In a time-sharing system, the operating system must ensure reasonable response time. This goal is sometimes accomplished through swapping, whereby processes are swapped in and out of main memory to the disk. A more common method for ensuring reasonable response time is virtual memory, a technique that allows the execution of a process that is not completely in memory.

The main advantage of the virtual-memory scheme is that it enables users to run programs that are larger than actual physical memory. Further, it abstracts main memory into a large, uniform array of storage, separating logical memory as viewed by the user from physical memory. This arrangement frees programmers from concern over memory-storage limitations.

OPERATING-SYSTEM OPERATIONS:

Modern operating systems are interrupt driven. If there are no processes to execute, no I/O devices to service, and no users to whom to respond, an operating system will sit quietly, waiting for something to happen. Events are almost always signaled by the occurrence of an interrupt or a trap.

A trap (or an exception) is a software-generated interrupt caused either by an error (for example, division by zero or invalid memory access) or by a specific request from a user program that an operating-system service be performed. The interrupt-driven nature of an operating system defines that system’s general structure. For each type of interrupt, separate segments of code in the operating system determine what action should be taken. An interrupt service routine is provided to deal with the interrupt.

Since the operating system and the users share the hardware and software resources of the computer system, we need to make sure that an error in a user program could cause problems only for the one program running. With sharing, many processes could be adversely affected by a bug in one program.

For example, if a process gets stuck in an infinite loop, this loop could prevent the correct operation of many other processes. More subtle errors can occur in a multiprogramming system, where one erroneous program might modify another program, the data of another program, or even the operating system itself.

Without protection against these sorts of errors, either the computer must execute only one process at a time or all output must be suspect. A properly designed operating system must ensure that an incorrect (or malicious) program cannot cause other programs to execute incorrectly.

DUAL-MODE AND MULTIMODE OPERATION:

In order to ensure the proper execution of the operating system, we must be able to distinguish between the execution of operating-system code and user defined code. The approach taken by most computer systems is to provide hardware support that allows us to differentiate among various modes of execution.

At the very least, we need two separate modes of operation: user mode and kernel mode (also called supervisor mode, system mode, or privileged mode). A bit, called the mode bit, is added to the hardware of the computer to indicate the current mode: kernel (0) or user (1). With the mode bit, we can distinguish between a task that is executed on behalf of the operating system and one that is executed on behalf of the user.

When the computer system is executing on behalf of a user application, the system is in user mode. However, when a user application requests a service from the operating system (via a system call) the system must transition from user to kernel mode to fulfill the request. This is shown in Figure 1.1.

At system boot time, the hardware starts in kernel mode. The operating system is then loaded and starts user applications in user mode. Whenever a trap or interrupt occurs, the hardware switches from user mode to kernel mode (that is, changes the state of the mode bit to 0).

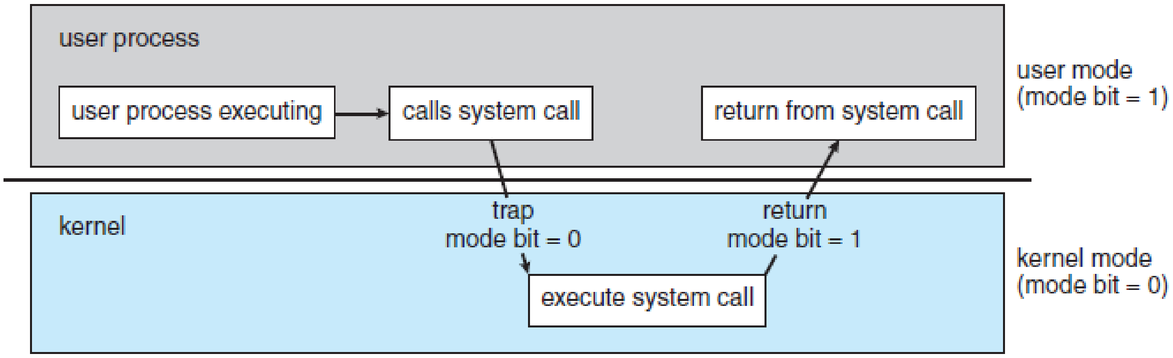


FIGURE 1.1: TRANSITION FROM USER TO KERNEL MODE

Thus, whenever the operating system gains control of the computer, it is in kernel mode. The system always switches to user mode (by setting the mode bit to 1) before passing control to a user program.

The dual mode of operation provides us with the means for protecting the operating system from errant users—and errant users from one another. We accomplish this protection by designating some of the machine instructions that may cause harm as privileged instructions.

The hardware allows privileged instructions to be executed only in kernel mode. If an attempt is made to execute a privileged instruction in user mode, the hardware does not execute the instruction but rather treats it as illegal and traps it to the operating system.

The instruction to switch to kernel mode is an example of a privileged instruction. Some other examples include I/O control, timer management, and interrupt management. As we shall see throughout the text, there are many additional privileged instructions.

The concept of modes can be extended beyond two modes (in which case the CPU uses more than one bit to set and test the mode). CPUs that support virtualization frequently have a separate mode to indicate when the virtual machine manager (VMM)—and the virtualization management software—are in control of the system.

In this mode, the VMM has more privileges than user processes but fewer than the kernel. It needs that level of privilege so it can create and manage virtual machines, changing the CPU state to do so. Sometimes, too, different modes are used by various kernel components.

TIMER:

We must ensure that the operating system maintains control over the CPU. We cannot allow a user program to get stuck in an infinite loop or to fail to call system services and never return control to the operating system. To accomplish this goal, we can use a timer. A timer can be set to interrupt the computer after a specified period.

The period may be fixed (for example, 1/60 second) or variable (for example, from 1 millisecond to 1 second). A variable timer is generally implemented by a fixed-rate clock and a counter. The operating system sets the counter. Every time the clock ticks, the counter is decremented.

When the counter reaches 0, an interrupt occurs. For instance, a 10-bit counter with a 1-millisecond clock allows interrupts at intervals from 1 millisecond to 1,024 milliseconds, in steps of 1 millisecond.

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

We can use the timer to prevent a user program from running too long. A simple technique is to initialize a counter with the amount of time that a program is allowed to run. A program with a 7-minute time limit, for example, would have its counter initialized to 420.

Every second, the timer interrupts, and the counter is decremented by 1. As long as the counter is positive, control is returned to the user program. When the counter becomes negative, the operating system terminates the program for exceeding the assigned time limit.

PROTECTION AND SECURITY:

If a computer system has multiple users and allows the concurrent execution of multiple processes, then access to data must be regulated. For that purpose, mechanisms ensure that 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. Device-control registers are not accessible to users, so the integrity of the various peripheral devices is protected.

Protection, then, is any mechanism for controlling the access of processes or users to the resources defined by a computer system. This mechanism must provide means to specify the controls to be imposed and to enforce the controls.

Protection can improve reliability by detecting latent errors at the interfaces between component subsystems. Early detection of interface errors can often prevent contamination of a healthy subsystem by another subsystem that is malfunctioning. Furthermore, an unprotected resource cannot defend against use (or misuse) by an unauthorized or incompetent user.

A protection-oriented system provides a means to distinguish between authorized and unauthorized usage. A system can have adequate protection but still be prone to failure and allow inappropriate access. Consider a user whose authentication information (her means of identifying herself to the system) is stolen. Her data could be copied or deleted, even though file and memory protection are working.

It is the job of ***security*** to defend a system from external and internal attacks. Such attacks spread across a huge range and include viruses and worms, denial-of service attacks (which use all of a system’s resources and so keep legitimate users out of the system), identity theft, and theft of service (unauthorized use of a system).

Prevention of some of these attacks is considered an operating-system function on some systems, while other systems leave it to policy or additional software. Due to the alarming rise in security incidents, operating-system security features represent a fast-growing area of research and implementation.

Protection and security require the system to be able to distinguish among all its users. Most operating systems maintain a list of user names and associated user identifiers (user IDs). In Windows parlance, this is a security ID (SID). These numerical IDs are unique, one per user.

When a user logs in to the system, the authentication stage determines the appropriate user ID for the user. That user ID is associated with all of the user’s processes and threads. When an ID needs to be readable by a user, it is translated back to the user name via the user name list.

In some circumstances, we wish to distinguish among sets of users rather than individual users. For example, the owner of a file on a UNIX system may be allowed to issue all operations on that file, whereas a selected set of users may be allowed only to read the file. To accomplish this, we need to define a group name and the set of users belonging to that group.

Group functionality can be implemented as a system-wide list of group names and group identifiers. A user can be in one or more groups, depending on operating-system design decisions. The user's group IDs is also included in every associated process and thread.

KERNEL DATA STRUCTURES:

Lists, Stacks, and Queues: An array is a simple data structure in which each element can be accessed directly. For example, main memory is constructed as an array. If the data item being stored is larger than one byte, then multiple bytes can be allocated to the item, and the item is addressed as item number × item size. But what about storing an item whose size may vary? And what about removing an item if the relative positions of the remaining items must be preserved? In such situations, arrays give way to other data structures.

After arrays, lists are perhaps the most fundamental data structures in computer science. Whereas each item in an array can be accessed directly, the items in a list must be accessed in a particular order. That is, a list represents a collection of data values as a sequence. The most common method for implementing this structure is a linked list, in which items are linked to one another. Linked lists are of several types:

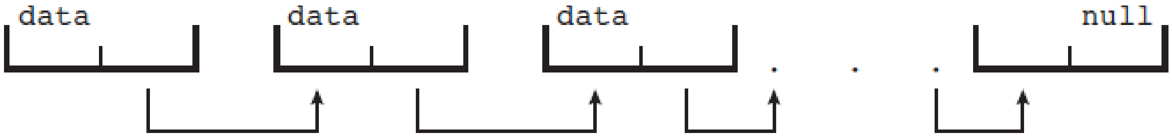


FIGURE 1.2 SINGLY LINKED LIST

- In a singly linked list, each item points to its successor in as illustrated in Figure 1.2
- In a doubly linked list, a given item can refer either to its predecessor or to its successor
- In a circularly linked list, the last element in the list refers to the first element, rather than to null

Linked lists accommodate items of varying sizes and allow easy insertion and deletion of items. One potential disadvantage of using a list is that performance for retrieving a specified item in a list of size *n* is linear — *O*(*n*), as it requires potentially traversing all *n* elements in the worst case. Lists are sometimes used directly by kernel algorithms. Frequently, though, they are used for constructing more powerful data structures, such as stacks and queues.

A **stack** is a sequentially ordered data structure that uses the last in, first out (**LIFO**) principle for adding and removing items, meaning that the last item placed onto a stack is the first item removed. The operations for inserting and removing items from a stack are known as ***push*** and ***pop***, respectively. An operating system often uses a stack when invoking function calls. Parameters, local variables, and the return address are pushed onto the stack when a function is called; returning from the function call pops those items off the stack.

A **queue**, in contrast, is a sequentially ordered data structure that uses the first in, first out (**FIFO**) principle: items are removed from a queue in the order in which they were inserted. There are many everyday examples of queues, including shoppers waiting in a checkout line at a store and cars waiting in line at a traffic signal. Queues are also quite common in operating systems—jobs that are sent to a printer are typically printed in the order in which they were submitted, for example.

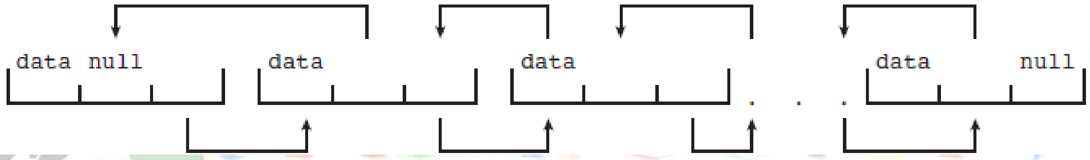


FIGURE 1.3 DOUBLY LINKED LIST

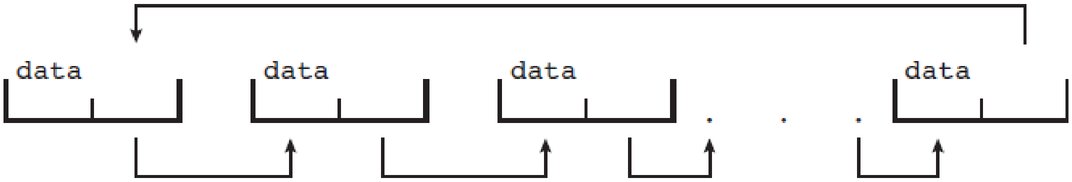


FIGURE 1.4 CIRCULARLY LINKED LIST

Trees:

A tree is a data structure that can be used to represent data hierarchically. Data values in a tree structure are linked through parent–child relationships. In a general tree, a parent may have an unlimited number of children.

In a binary tree, a parent may have at most two children, which we term the left child and the right child. A binary search tree additionally requires an ordering between the parent's two children in which left child <= right child.

Hash Functions and Maps:

A hash function takes data as its input, performs a numeric operation on this data, and returns a numeric value. This numeric value can then be used as an index into a table (typically an array) to quickly retrieve the data. Whereas searching for a data item through a list of size n can require up to O(n) comparisons in the worst case, using a hash function for retrieving data from table can be as good as O(1) in the worst case, depending on implementation details. Because of this performance, hash functions are used extensively in operating systems.

One potential difficulty with hash functions is that two inputs can result in the same output value—that is, they can link to the same table location. We can accommodate this hash collision by having a linked list at that table location that contains all of the items with the same hash value. Of course, the more collisions there are, the less efficient the hash function is.

One use of a hash function is to implement a hash map, which associates (or maps) [key: value] pairs using a hash function. For example, we can map the key operating to the value system. Once the mapping is established, we can apply the hash function to the key to obtain the value from the hash map (Figure 1.5).

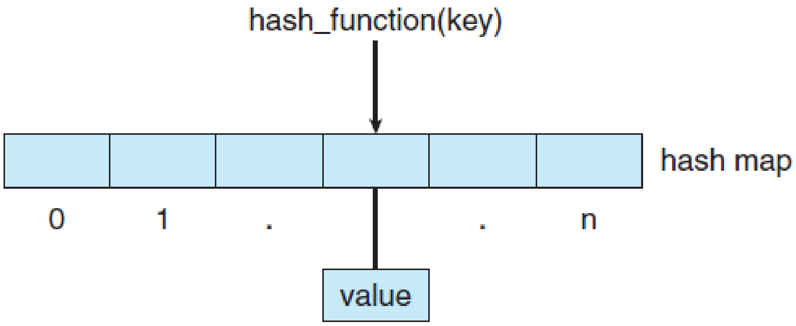


FIGURE 1.5: HASH MAP

For example, suppose that a user name is mapped to a password. Password authentication then proceeds as follows: a user enters his user name and password. The hash function is applied to the user name, which is then used to retrieve the password. The retrieved password is then compared with the password entered by the user for authentication.

Bitmaps: A bitmap is a string of n binary digits that can be used to represent the status of n items. For example, suppose we have several resources and the availability of each resource is indicated by the value of a binary digit: 0 means that the resource is available, while 1 indicates that it is unavailable (or vice-versa).

The value of the ith position in the bitmap is associated with the ith resource. As an example, consider the bitmap shown below:

001011101

Resources 2, 4, 5, 6, and 8 are unavailable; resources 0, 1, 3, and 7 are available. The power of bitmaps becomes apparent when we consider their space efficiency. If we were to use an eight-bit Boolean value instead of a single bit, the resulting data structure would be eight times larger. Thus, bitmaps are commonly used when there is a need to represent the availability of a large number of resources. Disk drives provide a nice illustration. A medium-sized disk drive might be divided into several thousand individual units, called disk blocks. A bitmap can be used to indicate the availability of each disk block.

COMPUTING ENVIRONMENTS:

Traditional Computing:

As computing has matured the lines separating many traditional computing environments have blurred. Consider the “typical office environment.” Just a few years ago, this environment consisted of PCs connected to a network, with servers providing file and print services. Remote access was awkward, and portability was achieved by use of laptop computers. Terminals attached to mainframes were prevalent at many companies as well, with even fewer remote access and portability options.

The current trend is toward providing more ways to access these computing environments. Web technologies and increasing WAN bandwidth are stretching the boundaries of traditional computing. Companies establish portals, which provide Web accessibility to their internal servers.

Network computers (or thin clients)—which are essentially terminals that understand web-based computing—are used in place of traditional workstations where more security or easier maintenance is desired.

Mobile computers can synchronize with PCs to allow very portable use of company information. Mobile computers can also connect to wireless networks and cellular data networks to use the company’s Web portal (as well as the myriad other Web resources).

At home, most users once had a single computer with a slow modem connection to the office, the Internet, or both. Today, network-connection speeds once available only at great cost are relatively inexpensive in many places, giving home users more access to more data.

These fast data connections are allowing home computers to serve up Web pages and to run networks that include printers, client PCs, and servers. Many homes use firewalls to protect their networks from security breaches.

In the latter half of the 20th century, computing resources were relatively scarce. (Before that, they were nonexistent!) For a period of time, systems were either batch or interactive.

Batch systems processed jobs in bulk, with predetermined input from files or other data sources. Interactive systems waited for input from users. To optimize the use of the computing resources, multiple users shared time on these systems. Time-sharing systems used a timer and scheduling algorithms to cycle processes rapidly through the CPU, giving each user a share of the resources.

Mobile Computing:

Mobile computing refers to computing on handheld smart phones and tablet computers. These devices share the distinguishing physical features of being portable and lightweight. Historically, compared with desktop and laptop computers, mobile systems gave up screen size, memory capacity, and overall functionality in return for handheld mobile access to services such as e-mail and web browsing. Over the past few years, however, features on mobile devices have become so rich that the distinction in functionality between, say, a consumer laptop and a tablet computer may be difficult to discern.

Two operating systems currently dominate mobile computing: Apple iOS and Google Android. iOS was designed to run on Apple iPhone and iPad mobile devices.

Distributed Systems:

A distributed system is a collection of physically separate, possibly heterogeneous, computer systems that are networked to provide users with access to the various resources that the system maintains. Access to a shared resource increases computation speed, functionality, data availability, and reliability.

Some operating systems generalize network access as a form of file access, with the details of networking contained in the network interface's device driver. Others make users specifically invoke network functions. Generally, systems contain a mix of the two modes—for example FTP and NFS. The protocols that create a distributed system can greatly affect that system's utility and popularity.

Distributed systems depend on networking for their functionality. Networks vary by the protocols used, the distances between nodes, and the transport media. TCP/IP is the most common network protocol, and it provides the fundamental architecture of the Internet. Most operating systems support TCP/IP, including all general-purpose ones.

Client–Server Computing:

As PCs have become faster, more powerful, and cheaper, designers have shifted away from centralized system architecture. Terminals connected to centralized systems are now being supplanted by PCs and mobile devices.

Server systems can be broadly categorized as compute servers and file servers:

The ***compute-server system*** provides an interface to which a client can send a request to perform an action (for example, read data). In response, the server executes the action and sends the results to the client. A server running a database that responds to client requests for data is an example of such a system.

The **file-server system** provides a file-system interface where clients can create, update, read, and delete files. An example of such a system is a web server that delivers files to clients running web browsers.

Peer-to-Peer Computing:

Another structure for a distributed system is the peer-to-peer (P2P) system model. In this model, clients and servers are not distinguished from one another. Instead, all nodes within the system are considered peers, and each may act as either a client or a server, depending on whether it is requesting or providing a service.

Peer-to-peer systems offer an advantage over traditional client-server systems. In a client-server system, the server is a bottleneck; but in a peer-to-peer system, services can be provided by several nodes distributed throughout the network.

Virtualization:

Virtualization is a technology that allows operating systems to run as applications within other operating systems.

Broadly speaking, virtualization is one member of a class of software that also includes emulation. Emulation is used when the source CPU type is different from the target CPU type. For example, when Apple switched from the IBM Power CPU to the Intel x86 CPU for its desktop and laptop computers, it included an emulation facility called “Rosetta,” which allowed applications compiled for the IBM CPU to run on the Intel CPU.

A common example of emulation occurs when a computer language is not compiled to native code but instead is either executed in its high-level form or translated to an intermediate form. This is known as interpretation.

With virtualization, in contrast, an operating system that is natively compiled for a particular CPU architecture runs within another operating system also native to that CPU.

Cloud Computing:

Cloud computing is a type of computing that delivers computing, storage, and even applications as a service across a network. In some ways, it’s a logical extension of virtualization, because it uses virtualization as a base for its functionality.

There are actually many types of cloud computing, including the following:

- Public cloud—a cloud available via the Internet to anyone willing to pay for the services
- Private cloud—a cloud run by a company for that company’s own use
- Hybrid cloud—a cloud that includes both public and private cloud components
- Software as a service (SaaS)—one or more applications (such as word processors or spreadsheets) available via the Internet
- Platform as a service (PaaS)—a software stack ready for application use via the Internet (for example, a database server)

- Infrastructure as a service (IaaS)—servers or storage available over the Internet (for example, storage available for making backup copies of production data)

Figure 1.6 illustrates a public cloud providing IaaS. Notice that both the cloud services and the cloud user interface are protected by a firewall.

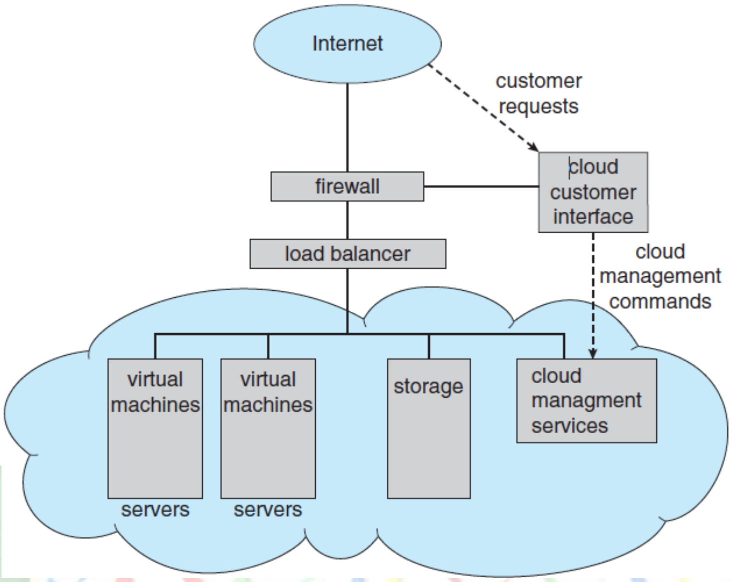


FIGURE 1.6: CLOUD COMPUTING

Real-Time Embedded Systems:

Embedded computers are the most prevalent form of computers in existence. These devices are found everywhere, from car engines and manufacturing robots to DVDs and microwave ovens. They tend to have very specific tasks. The systems they run on are usually primitive, and so the operating systems provide limited features. Usually, they have little or no user interface, preferring to spend their time monitoring and managing hardware devices, such as automobile engines and robotic arms.

OPEN-SOURCE OPERATING SYSTEMS:

Open-source operating systems are those available in source-code format rather than as compiled binary code. Linux is the most famous open source operating system, while Microsoft Windows is a well-known example of the opposite closed-source approach. Apple's Mac OS X and iOS operating systems comprise a hybrid approach. They contain an open-source kernel named Darwin yet include proprietary, closed-source components as well.

Starting with the source code allows the programmer to produce binary code that can be executed on a system. Doing the opposite—reverse engineering the source code from the binaries—is quite a lot of work and useful items such as comments are never recovered. Learning operating systems by examining the source code has other benefits as well.

There are many benefits to open-source operating systems, including a community of interested (and usually unpaid) programmers who contribute to the code by helping to debug it, analyze it, provide support, and suggest changes.

Arguably, open-source code is more secure than closed-source code because many more eyes are viewing the code. Certainly, open-source code has bugs, but open-source advocates argue that bugs tend to be found and fixed faster owing to the number of people using and viewing the code.

Companies that earn revenue from selling their programs often hesitate to open-source their code, but Red Hat and a myriad of other companies are doing just that and showing that commercial companies benefit, rather than suffer, when they open-source their code. Revenue can be generated through support contracts and the sale of hardware on which the software runs, for example.

HISTORY:

In the early days of modern computing (that is, the 1950s), a great deal of software was available in open-source format. The original hackers (computer enthusiasts) at MIT's Tech Model Railroad Club left their programs in drawers for others to work on. "Homebrew" user groups exchanged code during their meetings. Later, company-specific user groups, such as Digital Equipment Corporation's DEC, accepted contributions of source-code programs, collected them onto tapes, and distributed the tapes to interested members.

Computer and software companies eventually sought to limit the use of their software to authorized computers and paying customers. Releasing only the binary files compiled from the source code, rather than the source code itself, helped them to achieve this goal, as well as protecting their code and their ideas from their competitors.

Another issue involved copyrighted material. Operating systems and other programs can limit the ability to play back movies and music or display electronic books to authorized computers. Such copy protection or digital rights management (DRM) would not be effective if the source code that implemented these limits were published.

Laws in many countries, including the U.S. Digital Millennium Copyright Act (DMCA), make it illegal to reverse-engineer DRM code or otherwise try to circumvent copy protection.

To counter the move to limit software use and redistribution, Richard Stallman in 1983 started the GNU project to create a free, open-source, UNIX compatible operating system.

In 1985, he published the GNU Manifesto, which argues that all software should be free and open-sourced.

He also formed the Free Software Foundation (FSF) with the goal of encouraging the free exchange of software source code and the free use of that software.

Rather than copyright its software, the FSF "copylefts" the software to encourage sharing and improvement.

The GNU General Public License (GPL) codifies copylefting and is a common license under which free software is released.

Fundamentally, GPL requires that the source code be distributed with any binaries and that any changes made to the source code are released under the same GPL license.

LINUX:

As an example of an open-source operating system, consider GNU/Linux. The GNU project produced many UNIX-compatible tools, including compilers, editors, and utilities, but never released a kernel.

In 1991, a student in Finland, Linus Torvalds, released a rudimentary UNIX-like kernel using the GNU compilers and tools and invited contributions worldwide. The advent of the Internet meant that anyone interested could download the source code, modify it, and submit changes to Torvalds. Releasing updates once a week allowed this so-called Linux operating system to grow rapidly, enhanced by several thousand programmers.

BSD UNIX:

BSD UNIX has a longer and more complicated history than Linux. It started in 1978 as a derivative of AT&T's UNIX. Releases from the University of California at Berkeley (UCB) came in source and binary form, but they were not open source because a license from AT&T was required.

BSD UNIX's development was slowed by a lawsuit by AT&T, but eventually a fully functional, open-source version, 4.4BSD-lite, was released in 1994. Just as with Linux, there are many distributions of BSD UNIX, including FreeBSD, NetBSD, OpenBSD, and DragonflyBSD.

SOLARIS:

Solaris is the commercial UNIX-based operating system of Sun Microsystems. Originally, Sun's SunOS operating system was based on BSD UNIX. Sun moved to AT&T's System V UNIX as its base in 1991. In 2005, Sun open-sourced most of the Solaris code as the Open Solaris project. The purchase of Sun by Oracle in 2009, however, left the state of this project unclear.

OPERATING SYSTEM STRUCTURES:

An operating system provides the environment within which programs are executed. Internally, operating systems vary greatly in their makeup, since they are organized along many different lines. The design of a new operating system is a major task. It is important that the goals of the system be well defined before the design begins. These goals form the basis for choices among various algorithms and strategies.

We can view an operating system from several vantage points. One view focuses on the services that the system provides; another, on the interface that it makes available to users and programmers; a third, on its components and their interconnections.

OPERATING-SYSTEM SERVICES:

An operating system provides an environment for the execution of programs. It provides certain services to programs and to the users of those programs.

The specific services provided, of course, differ from one operating system to another, but we can identify common classes.

These operating system services are provided for the convenience of the programmer, to make the programming task easier.

Figure 1.7 shows one view of the various operating-system services and how they interrelate. One set of operating system services provides functions that are helpful to the user.

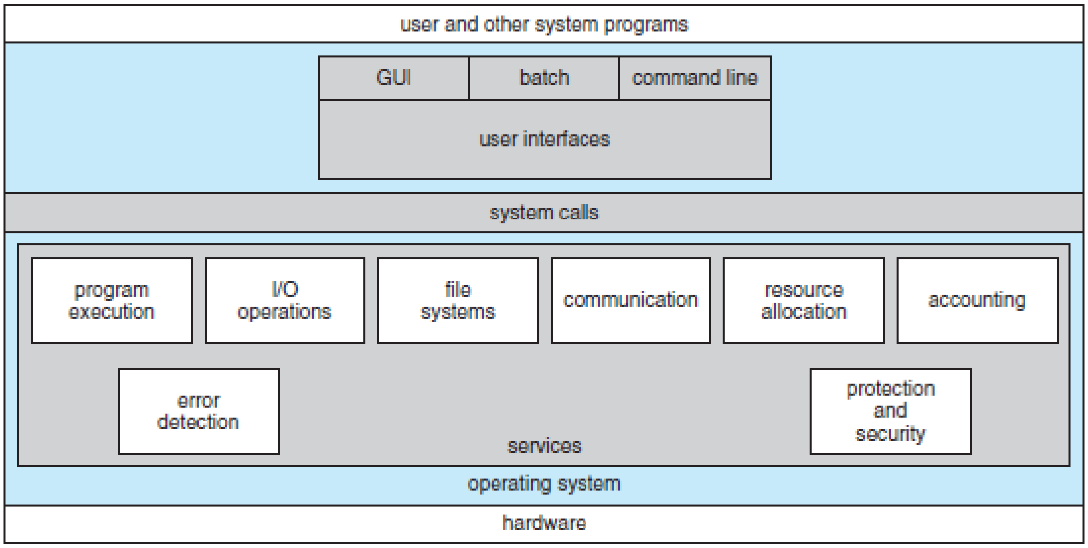


FIGURE 1.7 A VIEW OF OPERATING SYSTEM SERVICES

USER INTERFACE: Almost all operating systems have a user interface (UI). This interface can take several forms.

One is a command-line interface (CLI), which uses text commands and a method for entering them (say, a keyboard for typing in commands in a specific format with specific options).

Another is a batch interface, in which commands and directives to control those commands are entered into files, and those files are executed.

Most commonly, a graphical user interface (GUI) is used. Here, the interface is a window system with a pointing device to direct I/O, choose from menus, and make selections and a keyboard to enter text. Some systems provide two or all three of these variations.

PROGRAM EXECUTION: The system must be able to load a program into memory and to run that program. The program must be able to end its execution, either normally or abnormally (indicating error).

I/O OPERATIONS: A running program may require I/O, which may involve a file or an I/O device. For specific devices, special functions may be desired (such as recording to a CD or DVD drive or blanking a display screen). For efficiency and protection, users usually cannot control I/O devices directly. Therefore, the operating system must provide a means to do I/O.

FILE-SYSTEM MANIPULATION: The file system is of particular interest. Obviously, programs need to read and write files and directories. They also need to create and delete them by name, search for a given file, and list file information. Finally, some operating systems include permissions management to allow or deny access to files or directories based on file ownership.

COMMUNICATIONS: There are many circumstances in which one process needs to exchange information with another process. Such communication may occur between processes that are executing on the same computer or between processes that are executing on different computer systems tied together by a computer network.

Communications may be implemented via shared memory, in which two or more processes read and write to a shared section of memory, or message passing, in which packets of information in predefined formats are moved between processes by the operating system.

ERROR DETECTION: The operating system needs to be detecting and correcting errors constantly. Errors may occur in the CPU and memory hardware (such as a memory error or a power failure), in I/O devices (such as a parity error on disk, a connection failure on a network, or lack of paper in the printer), and in the user program (such as an arithmetic overflow, an attempt to access an illegal memory location, or a too-great use of CPU time). For each type of error, the operating system should take the appropriate action to ensure correct and consistent computing.

Another set of operating system functions exists not for helping the user but rather for ensuring the efficient operation of the system itself. Systems with multiple users can gain efficiency by sharing the computer resources among the users.

RESOURCE ALLOCATION: When there are multiple users or multiple jobs running at the same time, resources must be allocated to each of them. The operating system manages many different types of resources.

Some (such as CPU cycles, main memory, and file storage) may have special allocation code, whereas others (such as I/O devices) may have much more general request and release code. For instance, in determining how best to use the CPU, operating systems have CPU-scheduling routines that take into account the speed of the CPU, the jobs that must be executed, the number of registers available, and other factors. There may also be routines to allocate printers, USB storage drives, and other peripheral devices.

ACCOUNTING: We want to keep track of which users use how much and what kinds of computer resources. This record keeping may be used for accounting (so that users can be billed) or simply for accumulating usage statistics. Usage statistics may be a valuable tool for researchers who wish to reconfigure the system to improve computing services.

PROTECTION AND SECURITY: The owners of information stored in a multiuser or networked computer system may want to control use of that information. When several separate 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 requiring each user to authenticate him or her to the system, usually by means of a password, to gain access to system resources. It extends to defending external I/O devices, including network adapters, from invalid access attempts and to recording all such connections for detection of break-ins.

USER AND OPERATING-SYSTEM INTERFACE:

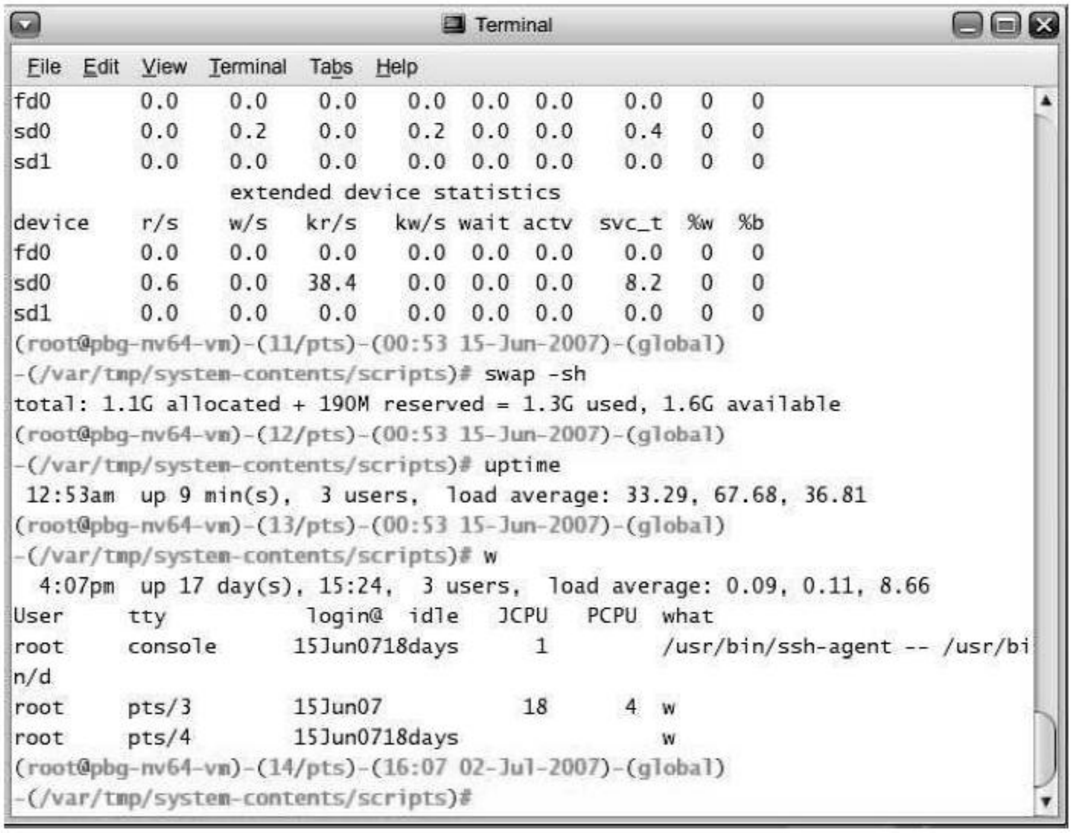
There are several ways for users to interface with the operating system. Here, we discuss two fundamental approaches. One provides a command-line interface, or command interpreter, that allows users to directly enter commands to be performed by the operating system. The other allows users to interface with the operating system via a graphical user interface, or GUI.

COMMAND INTERPRETERS:

Some operating systems include the command interpreter in the kernel. Others, such as Windows and UNIX, treat the command interpreter as a special program that is running when a job is initiated or when a user first logs on (on interactive systems).

On systems with multiple command interpreters to choose from, the interpreters are known as shells. For example, on UNIX and Linux systems, a user may choose among several different shells, including the Bourne shell, C shell, Bourne-Again shell, Korn shell, and others.

Third-party shells and free user-written shells are also available. Most shells provide similar functionality, and a user’s choice of which shell to use is generally based on personal preference. Figure 1.8 shows the Bourne shell command interpreter being used on Solaris 10.



```
fd0        0.0    0.0    0.0     0.0  0.0  0.0     0.0   0   0
sd0        0.0    0.2    0.0     0.2  0.0  0.0     0.4   0   0
sd1        0.0    0.0    0.0     0.0  0.0  0.0     0.0   0   0
                  extended device statistics
device     r/s    w/s   kr/s    kw/s wait actv  svc_t  %w  %b
fd0        0.0    0.0    0.0     0.0  0.0  0.0     0.0   0   0
sd0        0.6    0.0   38.4     0.0  0.0  0.0     8.2   0   0
sd1        0.0    0.0    0.0     0.0  0.0  0.0     0.0   0   0
(root@pbg-nv64-vm)-(11/pts)-(00:53 15-Jun-2007)-(global)
-(/var/tmp/system-contents/scripts)# swap -sh
total: 1.1G allocated + 190M reserved = 1.3G used, 1.6G available
(root@pbg-nv64-vm)-(12/pts)-(00:53 15-Jun-2007)-(global)
-(/var/tmp/system-contents/scripts)# uptime
 12:53am  up 9 min(s),  3 users,  load average: 33.29, 67.68, 36.81
(root@pbg-nv64-vm)-(13/pts)-(00:53 15-Jun-2007)-(global)
-(/var/tmp/system-contents/scripts)# w
  4:07pm  up 17 day(s), 15:24,  3 users,  load average: 0.09, 0.11, 8.66
User     tty           login@  idle   JCPU   PCPU  what
root     console      15Jun0718days      1         /usr/bin/ssh-agent -- /usr/bi
n/d
root     pts/3        15Jun07           18      4  w
root     pts/4        15Jun0718days                w
(root@pbg-nv64-vm)-(14/pts)-(16:07 02-Jul-2007)-(global)
-(/var/tmp/system-contents/scripts)#
```

FIGURE 1.8: THE BOURNE SHELL COMMAND INTERPRETER IN SOLARIS 10

The main function of the command interpreter is to get and execute the next user-specified command. Many of the commands given at this level manipulate files: create, delete, list, print, copy, execute, and so on. The MS-DOS and UNIX shells operate in this way. These commands can be implemented in two general ways.

In one approach, the command interpreter itself contains the code to execute the command. For example, a command to delete a file may cause the command interpreter to jump to a section of its code that sets up the parameters and makes the appropriate system call. In this case, the number of commands that can be given determines the size of the command interpreter, since each command requires its own implementing code.

An alternative approach—used by UNIX, among other operating systems —implements most commands through system programs. In this case, the command interpreter does not understand the command in any way; it merely uses the command to identify a file to be loaded into memory and executed.

Thus, the UNIX command to delete a file **rm file.txt** would search for a file called rm, load the file into memory, and execute it with the parameter file.txt. The function associated with the rm command would be defined completely by the code in the file rm.

In this way, programmers can add new commands to the system easily by creating new files with the proper names. The command-interpreter program, which can be small, does not have to be changed for new commands to be added.

GRAPHICAL USER INTERFACES:

A second strategy for interfacing with the operating system is through a user-friendly graphical user interface, or GUI. Here, rather than entering commands directly via a command-line interface, users employ a mouse-based window-and- menu system characterized by a desktop metaphor.

The user moves the mouse to position its pointer on images, or icons, on the screen (the desktop) that represent programs, files, directories, and system functions. Depending on the mouse pointer’s location, clicking a button on the mouse can invoke a program, select a file or directory—known as a folder—or pull down a menu that contains commands.

Graphical user interfaces first appeared due in part to research taking place in the early 1970s at Xerox PARC research facility. The first GUI appeared on the Xerox Alto computer in 1973. However, graphical interfaces became more widespread with the advent of Apple Macintosh computers in the 1980s.

The user interface for the Macintosh operating system (Mac OS) has undergone various changes over the years, the most significant being the adoption of the Aqua interface that appeared with Mac OS X.

Microsoft’s first version of Windows—Version 1.0—was based on the addition of a GUI interface to the MS-DOS operating system. Later versions of Windows have made cosmetic changes in the appearance of the GUI along with several enhancements in its functionality.

Because a mouse is impractical for most mobile systems, smart phones and handheld tablet computers typically use a touch screen interface. Here, users interact by making gestures on the touch screen—for example, pressing and swiping fingers across the screen.

Traditionally, UNIX systems have been dominated by command-line interfaces. Various GUI interfaces are available, however. These include the Common Desktop Environment (CDE) and X-Windows systems, which are common on commercial versions of UNIX, such as Solaris and IBM's AIX system.

In addition, there has been significant development in GUI designs from various open-source projects, such as K Desktop Environment (or KDE) and the GNOME desktop by the GNU project. Both the KDE and GNOME desktops run on Linux and various UNIX systems and are available under open-source licenses, which mean their source code is readily available for reading and for modification under specific license terms.

Choice of Interface:

The choice of whether to use a command-line or GUI interface is mostly one of personal preference. System administrators who manage computers and power users who have deep knowledge of a system frequently use the command-line interface. For them, it is more efficient, giving them faster access to the activities they need to perform.

Indeed, on some systems, only a subset of system functions is available via the GUI, leaving the less common tasks to those who are command-line knowledgeable. Further, command line interfaces usually make repetitive tasks easier, in part because they have their own programmability.

For example, if a frequent task requires a set of command-line steps, those steps can be recorded into a file, and that file can be run just like a program. The program is not compiled into executable code but rather is interpreted by the command-line interface. These shell scripts are very common on systems that are command-line oriented, such as UNIX and Linux.

The user interface can vary from system to system and even from user to user within a system. It typically is substantially removed from the actual system structure. The design of a useful and friendly user interface is therefore not a direct function of the operating system.

SYSTEM CALLS:

System calls provide an interface to the services made available by an operating system. These calls are generally available as routines written in C and C++, although certain low-level tasks (for example, tasks where hardware must be accessed directly) may have to be written using assembly-language instructions.

Before we discuss how an operating system makes system calls available, let's first use an example to illustrate how system calls are used: writing a simple program to read data from one file and 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.

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.

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. This sequence requires many I/O system calls.

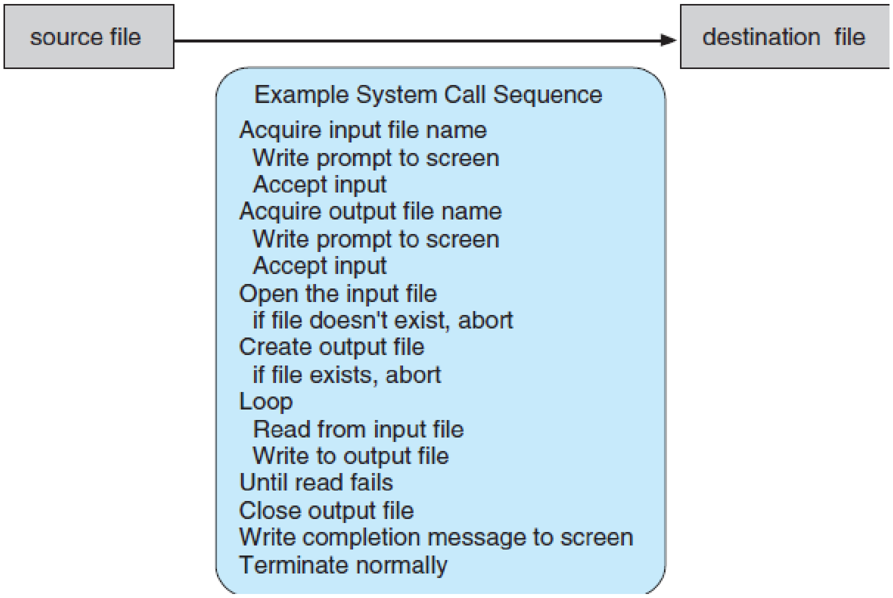
Once the two file names have been obtained, the program must open the input file and create the output file. Each of these operations requires another system call. Possible error conditions for each operation can require additional system calls.

When the program tries to open the input file, for example, 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 (yet another system call). This system-call sequence is shown in Figure 1.9

As you can see, even simple programs may make heavy use of the operating system. Frequently, systems execute thousands of system calls per second. Most programmers never see this level of detail, however. Typically, application developers design programs according to an application programming interface (API).

The API specifies a set of functions that are available to an application programmer, including the parameters that are passed to each function and the return values the programmer can expect.



```
Acquire input file name
  Write prompt to screen
  Accept input
Acquire output file name
  Write prompt to screen
  Accept input
Open the input file
  if file doesn't exist, abort
Create output file
  if file exists, abort
Loop
  Read from input file
  Write to output file
Until read fails
Close output file
Write completion message to screen
Terminate normally
```

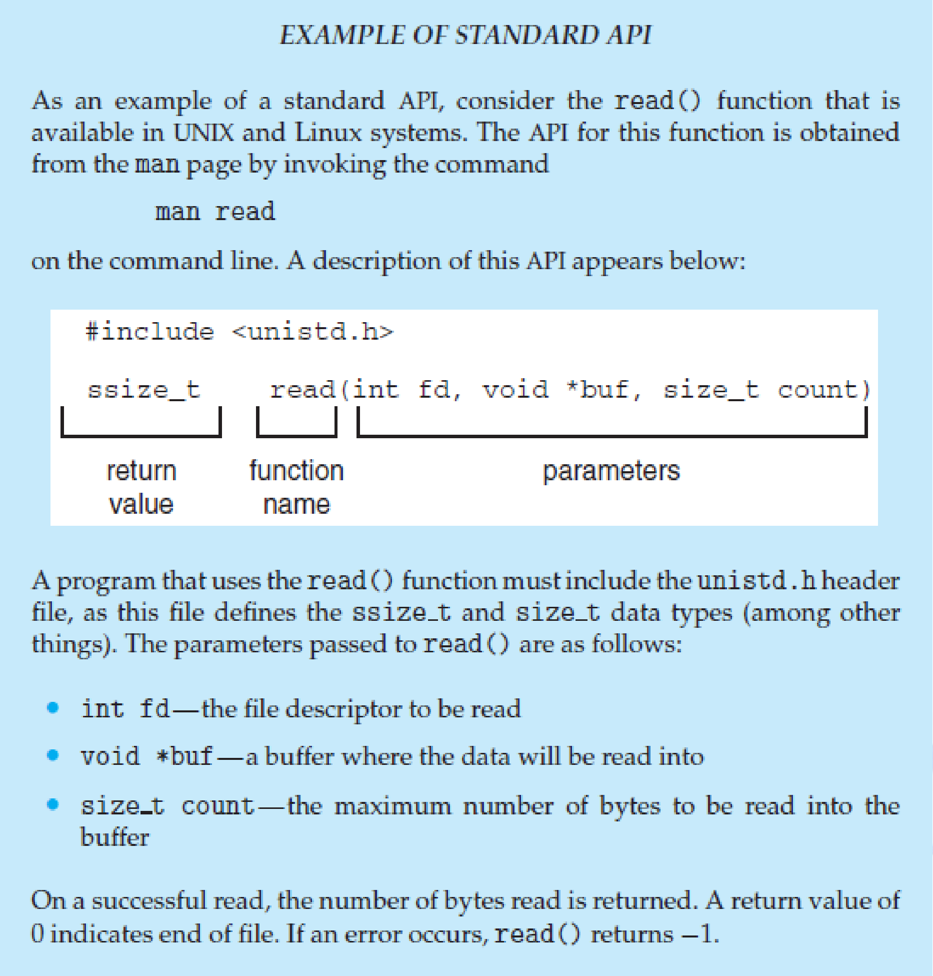
FIGURE 1.9: EXAMPLE OF HOW SYSTEM CALLS ARE USED

Three of the most common APIs available to application programmers are the Windows API for Windows systems, the POSIX API for POSIX-based systems (which include virtually all versions of UNIX, Linux, and Mac OSX), and the Java API for programs that run on the Java virtual machine. A programmer accesses an API via a library of code provided by the operating system.

In the case of UNIX and Linux for programs written in the C language, the library is called ***libc***. Note that—unless specified—the system-call names used throughout this text are generic examples. Each operating system has its own name for each system call. Behind the scenes, the functions that make up an API typically invoke the actual system calls on behalf of the application programmer.

For example, the Windows function CreateProcess() (which unsurprisingly is used to create a new process) actually invokes the NTCreateProcess() system call in the Windows kernel.

There are several reasons for an application programmer to prefer programming according to an API rather than invoking actual system calls; one benefit concerns program portability. An application programmer designing a program using an API can expect her program to compile and run on any system that supports the same API.



EXAMPLE OF STANDARD API

As an example of a standard API, consider the `read()` function that is available in UNIX and Linux systems. The API for this function is obtained from the `man` page by invoking the command

```
man read
```

on the command line. A description of this API appears below:

```
#include <unistd.h>

ssize_t    read(int fd, void *buf, size_t count)
```

A program that uses the `read()` function must include the `unistd.h` header file, as this file defines the `ssize_t` and `size_t` data types (among other things). The parameters passed to `read()` are as follows:

- `int fd`—the file descriptor to be read
- `void *buf`—a buffer where the data will be read into
- `size_t count`—the maximum number of bytes to be read into the buffer

On a successful read, the number of bytes read is returned. A return value of 0 indicates end of file. If an error occurs, `read()` returns −1.

FIGURE 1.10: EXAMPLE OF STANDARD API

Furthermore, actual system calls can often be more detailed and difficult to work with than the API available to an application programmer. Nevertheless, there often exists a strong correlation between a function in the API and its associated system call within the kernel.

For most programming languages, the run-time support system (a set of functions built into libraries included with a compiler) provides a system call interface that serves as the link to system calls made available by the operating system.

The system-call interface intercepts function calls in the API and invokes the necessary system calls within the operating system. Typically, a number is associated with each system call, and the system-call interface maintains a table indexed according to these numbers.

The system call interface then invokes the intended system call in the operating-system kernel and returns the status of the system call and any return values.

The caller need know nothing about how the system call is implemented or what it does during execution. Rather, the caller need only obey the API and understand what the operating system will do as a result of the execution of that system call. Thus, most of the details of the operating-system interface are hidden from the programmer by the API and are managed by the run-time support library.

The relationship between an API, the system-call interface, and the operating system is shown in Figure 1.11, which illustrates how the operating system handles a user application invoking the open() system call.

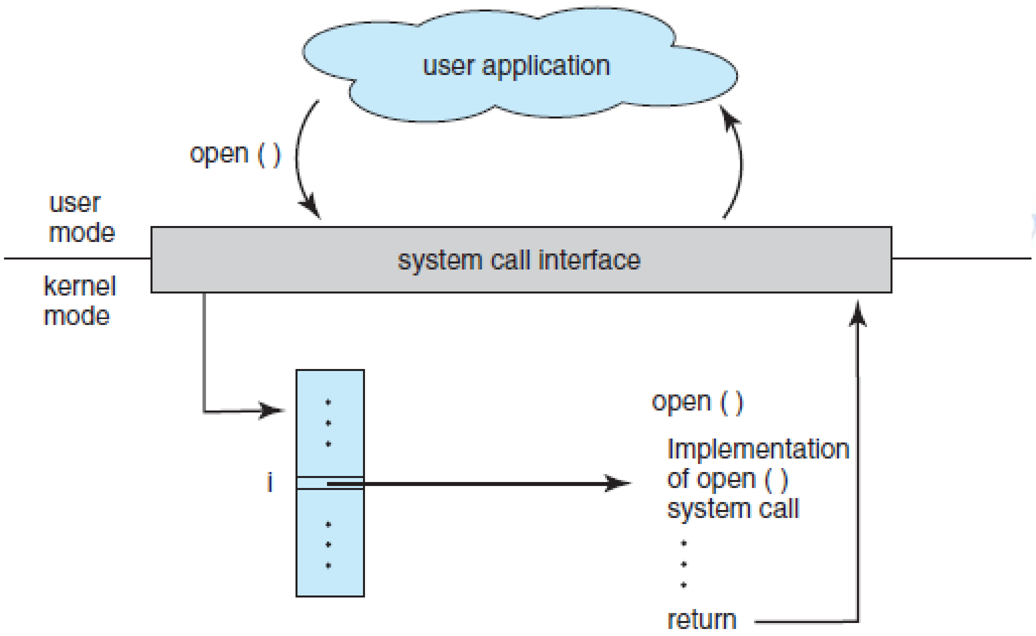


FIGURE 1.11: THE HANDLING OF A USER APPLICATION INVOKING THE open() SYSTEM CALL

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 1.12). *This is the approach taken by Linux and Solaris.*

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 method because those approaches do not limit the number or length of parameters being passed.

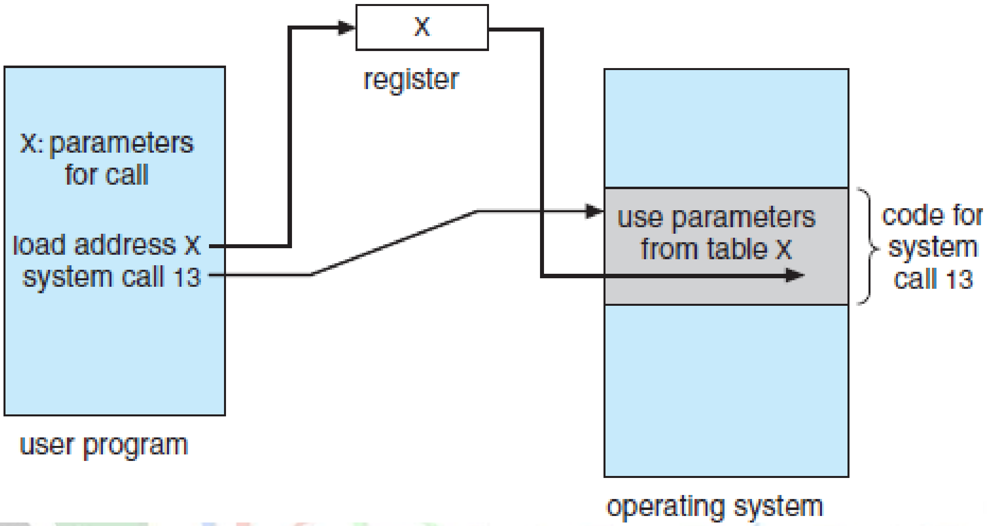


FIGURE 1.12: PASSING OF PARAMETERS AS A TABLE

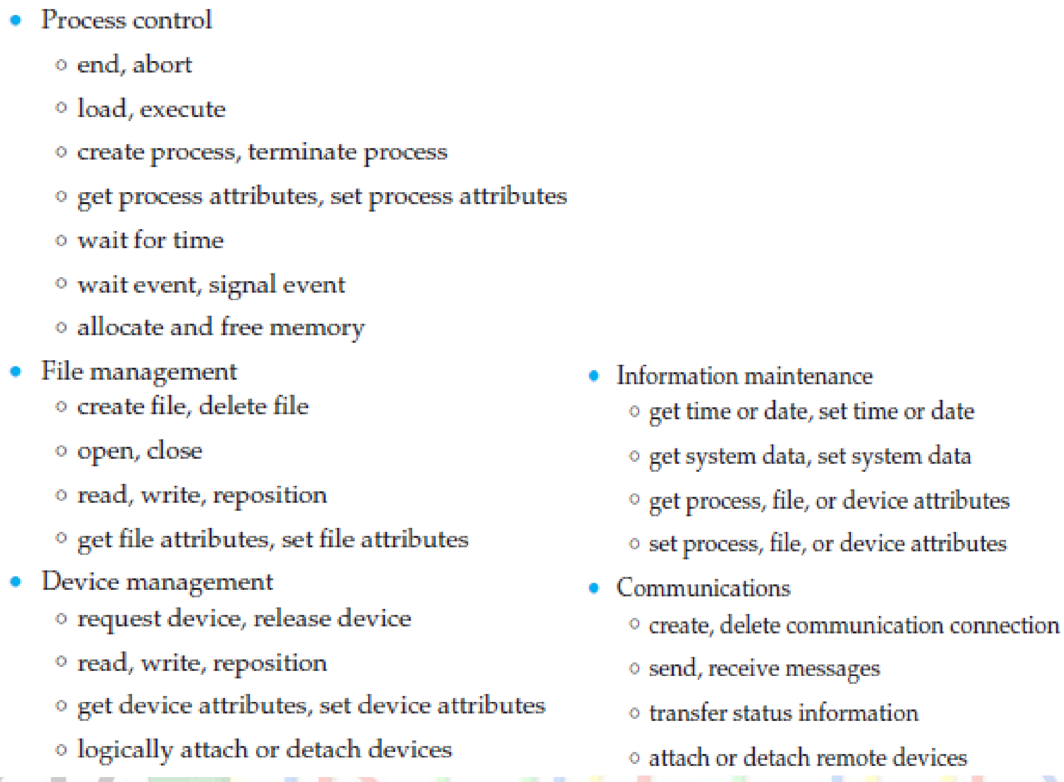
TYPES OF SYSTEM CALLS:

System calls can be grouped roughly into six major categories: *process control*, *file manipulation*, *device manipulation*, *information maintenance*, *communications*, and *protection*. Figure 1.13 summarizes the types of system calls normally provided by an operating system.

Process 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—a system program designed to aid the programmer in finding and correcting errors, or bugs—to determine the cause of the problem.

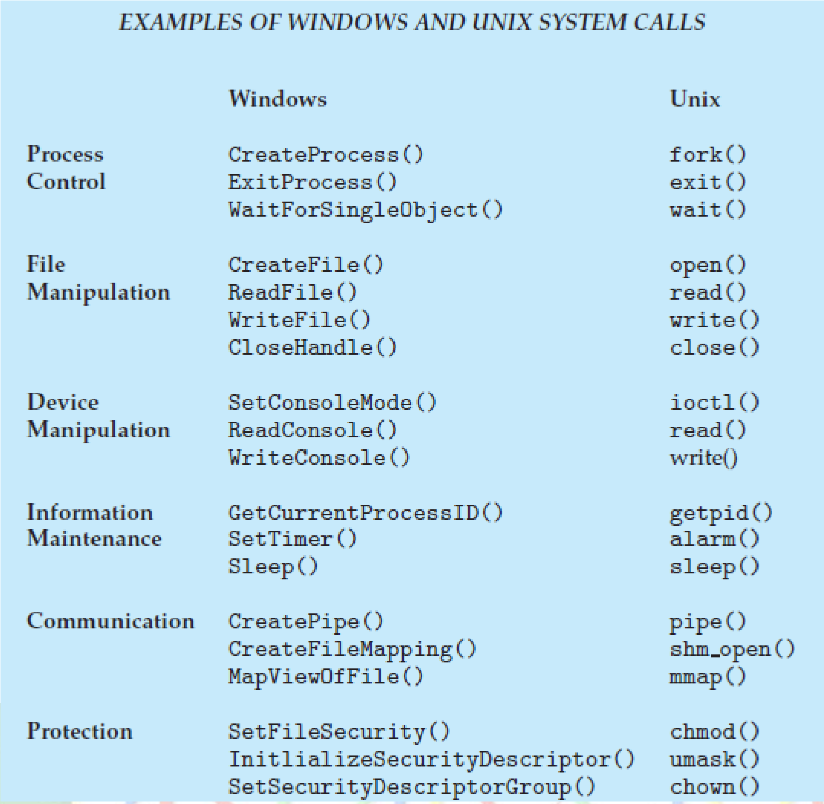


- Process control
    - end, abort
    - load, execute
    - create process, terminate process
    - get process attributes, set process attributes
    - wait for time
    - wait event, signal event
    - allocate and free memory
- File management
    - create file, delete file
    - open, close
    - read, write, reposition
    - get file attributes, set file attributes
- Device management
    - 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 1.13: TYPES OF SYSTEM CALLS

Under either normal or abnormal circumstances, the operating system must transfer control to the invoking command interpreter. The command interpreter then reads the next command. In an interactive system, the command interpreter simply continues with the next command; it is assumed that the user will issue an appropriate command to respond to any error.

In a GUI system, a pop-up window might alert the user to the error and ask for guidance. In a batch system, the command interpreter usually terminates the entire job and continues with the next job. Some systems may allow for special recovery actions in case an error occurs. If the program discovers an error in its input and wants to terminate abnormally, it may also want to define an error level. More severe errors can be indicated by a higher-level error parameter.



EXAMPLES OF WINDOWS AND UNIX SYSTEM CALLS

	Windows	Unix

Process Control	`CreateProcess()`
	`ExitProcess()`
	`WaitForSingleObject()`
		`fork()`
		`exit()`
		`wait()`

File Manipulation	`CreateFile()`
	`ReadFile()`
	`WriteFile()`
	`CloseHandle()`
		`open()`
		`read()`
		`write()`
		`close()`

Device Manipulation	`SetConsoleMode()`
	`ReadConsole()`
	`WriteConsole()`
		`ioctl()`
		`read()`
		`write()`

Information Maintenance	`GetCurrentProcessID()`
	`SetTimer()`
	`Sleep()`
		`getpid()`
		`alarm()`
		`sleep()`

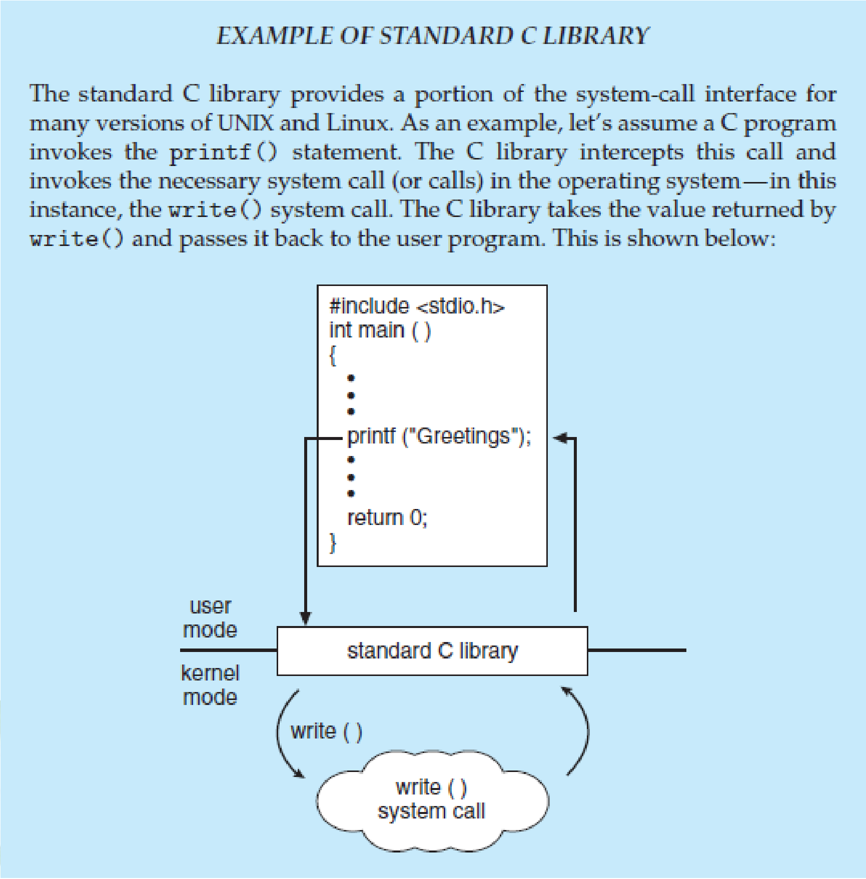
Communication	`CreatePipe()`
	`CreateFileMapping()`
	`MapViewOfFile()`
		`pipe()`
		`shm_open()`
		`mmap()`

Protection	`SetFileSecurity()`
	`InitlializeSecurityDescriptor()`
	`SetSecurityDescriptorGroup()`
		`chmod()`
		`umask()`
		`chown()`

A process or job executing one program may want to load() and execute() another program. This feature allows the command interpreter to execute a program as directed by, for example, a user command, the click of a mouse, or a batch command.

If control returns to the existing program when the new program terminates, we must save the memory image of the existing program; thus, we have effectively created a mechanism for one program to call another program. If both programs continue concurrently, we have created a new job or process to be Multiprogrammed. Often, there is a system call specifically for this purpose (create process() or submit job()).

If we create a new job or process, or perhaps even a set of jobs or processes, we should be able to control its execution. This control requires the ability to determine and reset the attributes of a job or process, including the job's priority, its maximum allowable execution time, and so on (get process attributes() and set process attributes()).We may also want to terminate a job or process that we created (terminate process()) if we find that it is incorrect or is no longer needed.



EXAMPLE OF STANDARD C LIBRARY

The standard C library provides a portion of the system-call interface for many versions of UNIX and Linux. As an example, let's assume a C program invokes the `printf()` statement. The C library intercepts this call and invokes the necessary system call (or calls) in the operating system—in this instance, the `write()` system call. The C library takes the value returned by `write()` and passes it back to the user program. This is shown below:

```
#include <stdio.h>
int main ( )
{
    •
    •
    •
    printf ("Greetings");
    •
    •
    •
    return 0;
}
```

Having created new jobs or processes, we may need to wait for them to finish their execution. We may want to wait for a certain amount of time to pass (wait time()). More probably, we will want to wait for a specific event to occur (wait event()). The jobs or processes should then signal when that event has occurred (signal event()).

Quite often, two or more processes may share data. To ensure the integrity of the data being shared, operating systems often provide system calls allowing a process to lock shared data. Then, no other process can access the data until the lock is released. Typically, such system calls include acquire lock() and release lock().

File Management:

We first need to be able to create() and delete() files. Either system call requires the name of the file and perhaps some of the file's attributes. Once the file is created, we need to open() it and to use it. We may also read(), write(), or reposition() (rewind or skip to the end of the file, for example).

Finally, we need to close() the file, indicating that we are no longer using it. We may need these same sets of operations for directories if we have a directory structure for organizing files in the file system. In addition, for either files or directories, we need to be able to determine the values of various attributes and perhaps to reset them if necessary.

File attributes include the file name, file type, protection codes, accounting information, and so on. At least two system calls, get file attributes() and set file attributes(), are required for this function. Some operating systems provide many more calls, such as calls for file move() and copy().

Others might provide an API that performs those operations using code and other system calls, and others might provide system programs to perform those tasks. If the system programs are callable by other programs, then each can be considered an API by other system programs.

Device Management:

A process may need several resources to execute—main memory, disk drives, access to files, and so on. If the resources are available, they can be granted, and control can be returned to the user process. Otherwise, the process will have to wait until sufficient resources are available.

The various resources controlled by the operating system can be thought of as devices. Some of these devices are physical devices (for example, disk drives), while others can be thought of as abstract or virtual devices (for example, files).

A system with multiple users may require us to first request() a device, to ensure exclusive use of it. After we are finished with the device, we release() it. These functions are similar to the open() and close() system calls for files. Other operating systems allow unmanaged access to devices.

Once the device has been requested (and allocated to us), we can read(), write(), and (possibly) reposition() the device, just as we can with files. In fact, the similarity between I/O devices and files is so great that many operating systems, including UNIX, merge the two into a combined file–device structure. In this case, a set of system calls is used on both files and devices. Sometimes, I/O devices are identified by special file names, directory placement, or file attributes.

The user interface can also make files and devices appear to be similar, even though the underlying system calls are dissimilar.

Information Maintenance:

Many system calls exist simply for the purpose of transferring information between the user program and the operating system. For example, most systems have a system call to return the current time() and date(). Other system calls may return information about the system, such as the number of current users, the version number of the operating system, the amount of free memory or disk space, and so on.

Another set of system calls is helpful in debugging a program. Many systems provide system calls to dump() memory. This provision is useful for debugging. A program trace lists each system call as it is executed. Even microprocessors provide a CPU mode known as single step, in which a trap is executed by the CPU after every instruction. The trap is usually caught by a debugger.

Communication:

There are two common models of inter process communication: the message passing model and the shared-memory model. In the message-passing model, the communicating processes exchange messages with one another to transfer information.

Messages can be exchanged between the processes either directly or indirectly through a common mailbox. Before communication can take place, a connection must be opened. The name of the other communicator must be known, be it another process on the same system or a process on another computer connected by a communications network.

Each computer in a network has a host name by which it is commonly known. A host also has a network identifier, such as an IP address. Similarly, each process has a process name, and this name is translated into an identifier by which the operating system can refer to the process. The get_hostid() and get_processid() system calls do this translation.

The identifiers are then passed to the general-purpose open() and close() calls provided by the file system or to specific open connection() and close connection() system calls, depending on the system's model of communication. The recipient process usually must give its permission for communication to take place with an accept connection() call.

In the shared-memory model, processes use shared memory create() and shared memory attach() system calls to create and gain access to regions of memory owned by other processes. Recall that, normally, the operating system tries to prevent one process from accessing another process's memory.

Shared memory requires that two or more processes agree to remove this restriction. They can then exchange information by reading and writing data in the shared areas. The form of the data is determined by the processes and is not under the operating system's control. The processes are also responsible for ensuring that they are not writing to the same location simultaneously.

Protection:

Protection provides a mechanism for controlling access to the resources provided by a computer system. Historically, protection was a concern only on multiprogrammed computer systems with several users. However, with the advent of networking and the Internet, all computer systems, from servers to mobile handheld devices, must be concerned with protection.

Typically, system calls providing protection include set permission() and get permission(), which manipulate the permission settings of resources such as files and disks. The allow user() and deny user() system calls specify whether particular users can—or cannot—be allowed access to certain resources.

SYSTEM PROGRAMS:

Another aspect of a modern system is its collection of system programs. According to logical computer hierarchy, at the lowest level is hardware. Next is the operating system, then the system programs, and finally the application programs. System programs, also known as system utilities, provide a convenient environment for program development and execution. Some of them are simply user interfaces to system calls. Others are considerably more complex. They can be divided into these categories:

File management: These programs create, delete, copy, rename, print, dump, list, and generally manipulate files and directories.

Status information: Some programs simply ask the system for the date, time, amount of available memory or disk space, number of users, or similar status information. Others are more complex, providing detailed performance, logging, and debugging information. Typically, these programs format and print the output to the terminal or other output devices or files or display it in a window of the GUI. Some systems also support a registry, which is used to store and retrieve configuration information.

File modification: Several text editors may be available to create and modify the content of files stored on disk or other storage devices. There may also be special commands to search contents of files or perform transformations of the text.

Programming-language support: Compilers, assemblers, debuggers, and interpreters for common programming languages (such as C, C++, Java, and PERL) are often provided with the operating system or available as a separate download.

Program loading and execution: Once a program is assembled or compiled, it must be loaded into memory to be executed. The system may provide absolute loaders, relocatable loaders, linkage editors, and overlay loaders. Debugging systems for either higher-level languages or machine language are needed as well.

Communications: These programs provide the mechanism for creating virtual connections among processes, users, and computer systems. They allow users to send messages to one another's screens, to browse Web pages, to send e-mail messages, to log in remotely, or to transfer files from one machine to another.

Background services: All general-purpose systems have methods for launching certain system-program processes at boot time. Some of these processes terminate after completing their tasks, while others continue to run until the system is halted. Constantly running system-program processes are known as services, subsystems, or daemons. One example is the network daemon.

Along with system programs, most operating systems are supplied with programs that are useful in solving common problems or performing common operations. Such application programs include Web browsers, word processors and text formatters, spreadsheets, database systems, compilers, plotting and statistical-analysis packages, and games.

The view of the operating system seen by most users is defined by the application and system programs, rather than by the actual system calls.

OPERATING SYSTEM STRUCTURE:

A system as large and complex as a modern operating system must be engineered carefully if it is to function properly and be modified easily. A common approach is to partition the task into small components, or modules, rather than have one monolithic system. Each of these modules should be a well-defined portion of the system, with carefully defined inputs, outputs, and functions.

SIMPLE STRUCTURE:

Many operating systems do not have well-defined structures. Frequently, such systems started as small, simple, and limited systems and then grew beyond their original scope. MS-DOS is an example of such a system. It was originally designed and implemented by a few people who had no idea that it would become so popular. It was written to provide the most functionality in the least space, so it was not carefully divided into modules. Figure 1.14 shows its structure:

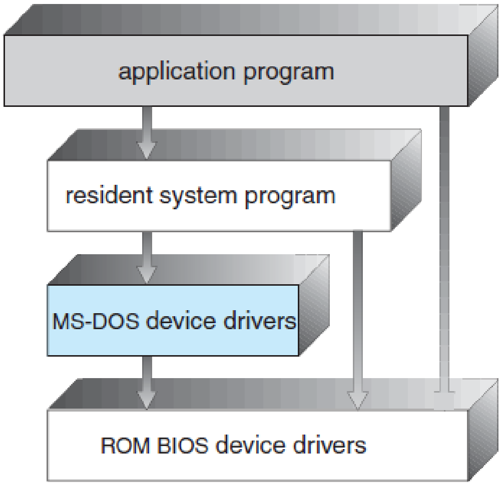


FIGURE 1.14: MS-DOS LAYER STRUCTURE

In MS-DOS, the interfaces and levels of functionality are not well separated. For instance, application programs are able to access the basic I/O routines to write directly to the display and disk drives. Such freedom leaves MS-DOS vulnerable to errant (or malicious) programs, causing entire system crashes when user programs fail.

Of course, MS-DOS was also limited by the hardware of its era. Because the Intel 8088 for which it was written provides no dual mode and no hardware protection, the designers of MS-DOS had no choice but to leave the base hardware accessible.

Another example of limited structuring is the original UNIX operating system. Like MS-DOS, UNIX initially was limited by hardware functionality. It consists of two separable parts: the kernel and the system programs. The kernel is further separated into a series of interfaces and device drivers, which have been added and expanded over the years as UNIX has evolved. We can view the traditional UNIX operating system as being layered to some extent, as shown in Figure 1.15:

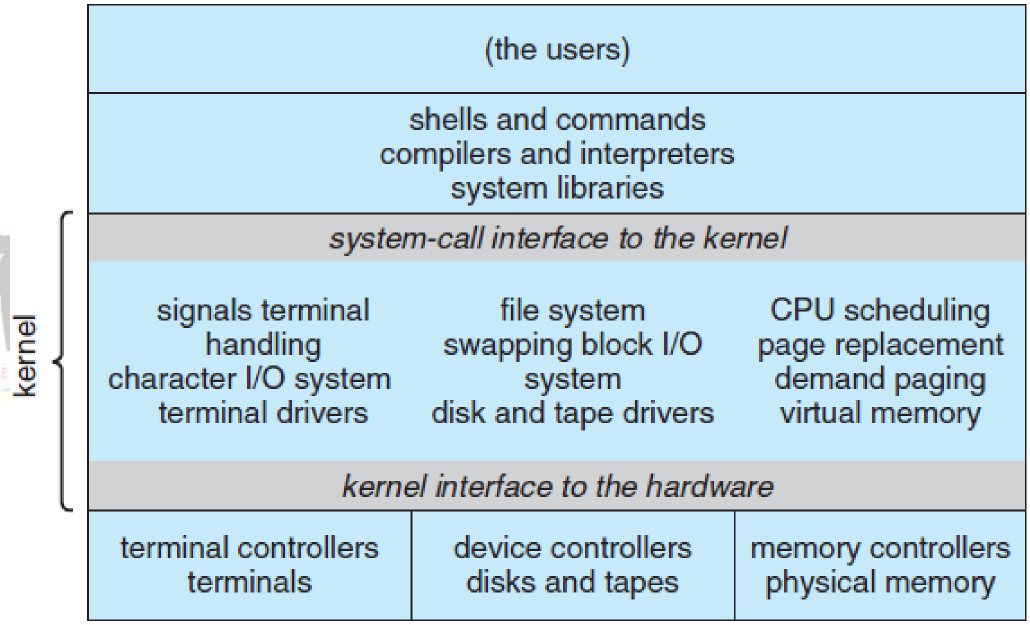


FIGURE 1.15: TRADITIONAL UNIX SYSTEM STRUCTURE

Everything below the system-call interface and above the physical hardware is the kernel. The kernel provides the file system, CPU scheduling, memory management, and other operating-system functions through system calls. Taken in sum, that is an enormous amount of functionality to be combined into one level.

This monolithic structure was difficult to implement and maintain. It had a distinct performance advantage, however: there is very little overhead in the system call interface or in communication within the kernel. We still see evidence of this simple, monolithic structure in the UNIX, Linux, and Windows operating systems.

LAYERED APPROACH:

A system can be made modular in many ways. One method is the layered approach, in which the operating system is broken into a number of layers (levels). The bottom layer (layer 0) is the hardware; the highest (layer N) is the user interface. This layering structure is depicted in Figure 1.16:

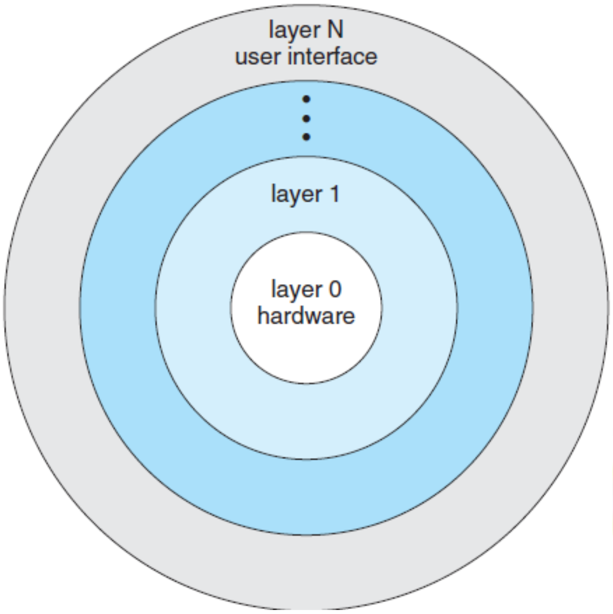


FIGURE 1.16: A LAYERED OPERATING SYSTEM

An operating-system layer is an implementation of an abstract object made up of data and the operations that can manipulate those data. A typical operating-system layer—say, layer M—consists of data structures and a set of routines that can be invoked by higher-level layers. Layer M, in turn, can invoke operations on lower-level layers.

The main advantage of the layered approach is simplicity of construction and debugging. The layers are selected so that each uses functions (operations) and services of only lower-level layers. This approach simplifies debugging and system verification.

The first layer can be debugged without any concern for the rest of the system, because, by definition, it uses only the basic hardware (which is assumed correct) to implement its functions. Once the first layer is debugged, its correct functioning can be assumed while the second layer is debugged, and so on.

If an error is found during the debugging of a particular layer, the error must be on that layer, because the layers below it are already debugged. Thus, the design and implementation of the system are simplified.

Each layer is implemented only with operations provided by lower-level layers. A layer does not need to know how these operations are implemented; it needs to know only what these operations do. Hence, each layer hides the existence of certain data structures, operations, and hardware from higher-level layers.

The major difficulty with the layered approach involves appropriately defining the various layers. Because a layer can use only lower-level layers, careful planning is necessary. Another problem with layered implementations is that they tend to be less efficient than other types.

MICROKERNELS:

In the mid-1980s, researchers at Carnegie Mellon University developed an operating system called Mach that modularized the kernel using the microkernel approach. This method structures the operating system by removing all nonessential components from the kernel and implementing them as system and user-level programs.

The result is a smaller kernel. There is little consensus regarding which services should remain in the kernel and which should be implemented in user space. Typically, however, Microkernel's provide minimal process and memory management, in addition to a communication facility. Figure 1.17 illustrates the architecture of a typical microkernel.

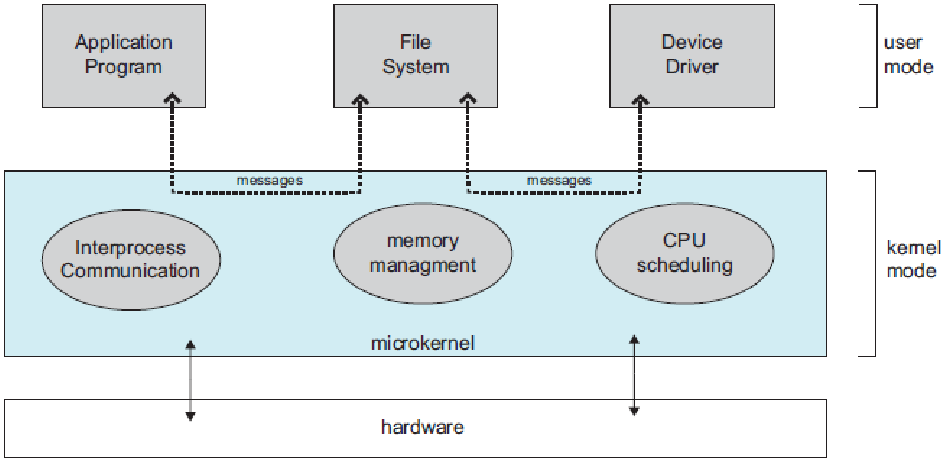


FIGURE 1.17: ARCHITECTURE OF A TYPICAL MICROKERNEL

The main function of the microkernel is to provide communication between the client program and the various services that are also running in user space. Communication is provided through message passing.

One benefit of the microkernel approach is that it makes extending the operating system easier. All new services are added to user space and consequently do not require modification of the kernel. When the kernel does have to be modified, the changes tend to be fewer, because the microkernel is a smaller kernel. The resulting operating system is easier to port from one hardware design to another.

The microkernel also provides more security and reliability, since most services are running as user—rather than kernel— processes. If a service fails, the rest of the operating system remains untouched.

Unfortunately, the performance of microkernel's can suffer due to increased system-function overhead.

MODULES:

Perhaps the best current methodology for operating-system design involves using loadable kernel modules. Here, the kernel has a set of core components and links in additional services via modules, either at boot time or during run time.

This type of design is common in modern implementations of UNIX, such as Solaris, Linux, and Mac OS X, as well as Windows. The idea of the design is for the kernel to provide core services while other services are implemented dynamically, as the kernel is running.

Linking services dynamically is preferable to adding new features directly to the kernel, which would require recompiling the kernel every time a change was made. Thus, for example, we might build CPU scheduling and memory management algorithms directly into the kernel and then add support for different file systems by way of loadable modules.

The overall result resembles a layered system in that each kernel section has defined, protected interfaces; but it is more flexible than a layered system, because any module can call any other module. The approach is also similar to the microkernel approach in that the primary module has only core functions and knowledge of how to load and communicate with other modules; but it is more efficient, because modules do not need to invoke message passing in order to communicate.

HYBRID SYSTEMS:

In practice, very few operating systems adopt a single, strictly defined structure. Instead, they combine different structures, resulting in hybrid systems that address performance, security, and usability issues. For example, both Linux and Solaris are monolithic, because having the operating system in a single address space provides very efficient performance.

However, they are also modular, so that new functionality can be dynamically added to the kernel. Windows is largely monolithic as well (again primarily for performance reasons), but it retains some behavior typical of microkernel systems, including providing support for separate subsystems (known as operating-system personalities) that run as user-mode processes. Windows systems also provide support for dynamically loadable kernel modules.

OPERATING-SYSTEM DEBUGGING:

Debugging is the activity of finding and fixing errors in a system, both in hardware and in software. Performance problems are considered bugs, so debugging can also include performance tuning, which seeks to improve performance by removing processing bottlenecks.

FAILURE ANALYSIS:

If a process fails, most operating systems write the error information to a log file to alert system operators or users that the problem occurred. The operating system can also take a core dump—a capture of the memory of the process— and store it in a file for later analysis. (Memory was referred to as the “core” in the early days of computing.) Running programs and core dumps can be probed by a debugger, which allows a programmer to explore the code and memory of a process.

A failure in the kernel is called a crash. When a crash occurs, error information is saved to a log file, and the memory state is saved to a crash dump. Operating-system debugging and process debugging frequently use different tools and techniques due to the very different nature of these two tasks.

PERFORMANCE TUNING:

Performance tuning seeks to improve performance by removing processing bottlenecks. To identify bottlenecks, we must be able to monitor system performance.

Thus, the operating system must have some means of computing and displaying measures of system behavior. In a number of systems, the operating system does this by producing trace listings of system behavior. All interesting events are logged with their time and important parameters and are written to a file.

Later, an analysis program can process the log file to determine system performance and to identify bottlenecks and inefficiencies. These same traces can be run as input for a simulation of a suggested improved system. Traces also can help people to find errors in operating-system behavior

Another approach to performance tuning uses single-purpose, interactive tools that allow users and administrators to question the state of various system components to look for bottlenecks; One such tool employs the UNIX command ***top*** to display the resources used on the system, as well as a sorted list of the "top" resource-using processes. Other tools display the state of disk I/O, memory allocation, and network traffic.

The Windows Task Manager is a similar tool for Windows systems. The task manager includes information for current applications as well as processes, CPU and memory usage, and networking statistics.

DTRACE:

DTrace is a facility that dynamically adds probes to a running system, both in user processes and in the kernel. These probes can be queried via the D programming language to determine an astonishing amount about the kernel, the system state, and process activities.

SYSTEM BOOT:

After an operating system is generated, it must be made available for use by the hardware. But how does the hardware know where the kernel is or how to load that kernel? The procedure of starting a computer by loading the kernel is known as booting the system. On most computer systems, a small piece of code known as the bootstrap program or bootstrap loader locates the kernel, loads it into main memory, and starts its execution.

Some computer systems, such as PCs, use a two-step process in which a simple bootstrap loader fetches a more complex boot program from disk, which in turn loads the kernel.

When a CPU receives a reset event—for instance, when it is powered up or rebooted—the instruction register is loaded with a predefined memory location, and execution starts there. At that location is the initial bootstrap program. This program is in the form of read-only memory (ROM), because the RAM is in an unknown state at system startup. ROM is convenient because it needs no initialization and cannot easily be infected by a computer virus.

The bootstrap program can perform a variety of tasks. Usually, one task is to run diagnostics to determine the state of the machine. If the diagnostics pass, the program can continue with the booting steps. It can also initialize all aspects of the system, from CPU registers to device controllers and the contents of main memory. Sooner or later, it starts the operating system.

Some systems—such as cellular phones, tablets, and game consoles—store the entire operating system in ROM. Storing the operating system in ROM is suitable for small operating systems, simple supporting hardware, and rugged operation. A problem with this approach is that changing the bootstrap code requires changing the ROM hardware chips. Some systems resolve this problem by using erasable programmable read-only memory (EPROM), which is read-only except when explicitly given a command to become writable. All forms of ROM are also known as firmware.

For large operating systems (including most general-purpose operating systems like Windows, Mac OS X, and UNIX) or for systems that change frequently, the bootstrap loader is stored in firmware, and the operating system is on disk. In this case, the bootstrap runs diagnostics and has a bit of code that can read a single block at a fixed location (say block zero) from disk into memory and execute the code from that boot block. The program stored in the boot block may be sophisticated enough to load the entire operating system into memory and begin its execution.

PROCESSES:

A process can be thought of as a program in execution. A process will need certain resources—such as CPU time, memory, files, and I/O devices —to accomplish its task. These resources are allocated to the process either when it is created or while it is executing.

A process is the unit of work in most systems. Systems consist of a collection of processes: operating-system processes execute system code, and user processes execute user code. All these processes may execute concurrently.

Although traditionally a process contained only a single thread of control as it ran, most modern operating systems now support processes that have multiple threads. The operating system is responsible for several important aspects of process and thread management: the creation and deletion of both user and system processes; the scheduling of processes; and the provision of mechanisms for synchronization, communication, and deadlock handling for processes.

PROCESS CONCEPT:

A batch system executes jobs, whereas a time-shared system has user programs, or tasks. Even on a single-user system, a user may be able to run several programs at one time: a word processor, a Web browser, and an e-mail package.

And even if a user can execute only one program at a time, such as on an embedded device that does not support multitasking, the operating system may need to support its own internal programmed activities, such as memory management. In many respects, all these activities are similar, so we call all of them processes.

The terms job and process are used almost interchangeably.

THE PROCESS

Informally, a process is a program in execution. A process is more than the program code, which is 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, which contains temporary data (such as function parameters, return addresses, and local variables), and a data section, which contains global variables.

A process may also include a heap, which is memory that is dynamically allocated during process run time. The structure of a process in memory is shown in Figure 1.18. We emphasize that a program by itself is not a process. A program is a passive entity, such as a file containing a list of instructions stored on disk (often called an executable file).

In contrast, a process is an active entity, with a program counter specifying the next instruction to execute and a set of associated resources. A program becomes a process when an executable file is loaded into memory.

Two common techniques for loading executable files are double-clicking an icon representing the executable file and entering the name of the executable file on the command line (as in prog.exe or a.out). Although two processes may be associated with the same program, they are nevertheless considered two separate execution sequences.

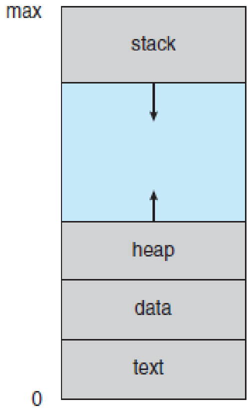


FIGURE 1.18: PROCESS IN MEMORY

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. A 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 they vary across operating systems. The states that they represent are found on all systems, however. Certain operating systems also more finely delineate 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 1.19.

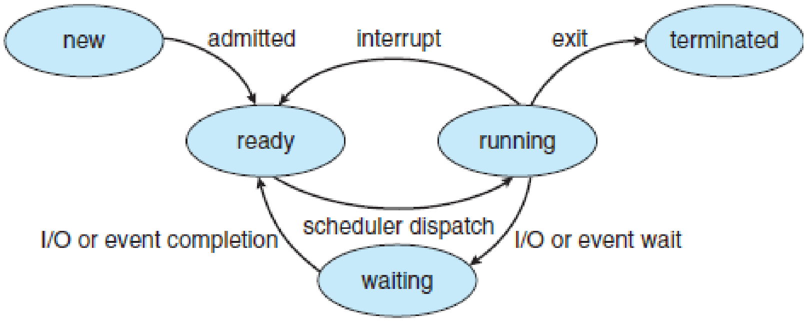


FIGURE 1.19: DIAGRAM OF PROCESS STATE

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 1.20. 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.
- CPU-scheduling information. This information includes a process priority, pointers to scheduling queues, and any other scheduling parameters.
- Memory-management information. This information may include such items as the value of the base and limit registers and the page tables, or the segment tables, depending on the memory system used by the operating system.
- Accounting information. This information includes the amount of CPU and real time used, time limits, account numbers, job or process numbers, and so on.
- I/O status information. This information includes the list of I/O devices allocated to the process, a list of open files, and so on.

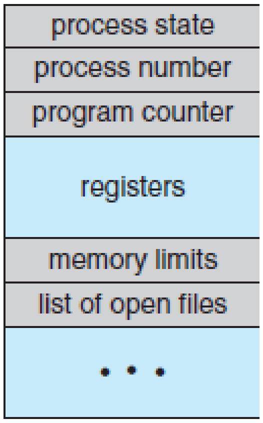


FIGURE 1.20: PROCESS CONTROL BLOCK (PCB)

THREADS:

The process model discussed so far has implied that a process is a program that performs a single thread of execution. For example, when a process is running a word-processor program, a single thread of instructions is being executed.

This single thread of control allows the process to perform only one task at a time. The user cannot simultaneously type in characters and run the spell checker within the same process, for example.

Most modern operating systems have extended the process concept to allow a process to have multiple threads of execution and thus to perform more than one task at a time. This feature is especially beneficial on multicore systems, where multiple threads can run in parallel.

On a system that supports threads, the PCB is expanded to include information for each thread. Other changes throughout the system are also needed to support threads.

PROCESS SCHEDULING:

The objective of multiprogramming is to have some process running at all times, to maximize CPU utilization. The objective of time sharing is to switch the CPU among processes so frequently that users can interact with each program while it is running.

To meet these objectives, the **process scheduler** selects an available process (possibly from a set of several available processes) for program execution on the CPU. For a single-processor system, there will never be more than one running process.

If there are more processes, the rest will have to wait until the CPU is free and can be rescheduled.

SCHEDULING QUEUES:

As processes enter the system, they are put into a job queue, which consists of all processes in the system. The processes that are residing in main memory and are ready and waiting to execute are kept on a list called the ready queue.

This queue is generally stored as a linked list. A ready-queue header contains pointers to the first and final PCBs in the list. Each PCB includes a pointer field that points to the next PCB in the ready queue.

The system also includes other queues. When a process is allocated the CPU, it executes for a while and eventually quits, is interrupted, or waits for the occurrence of a particular event, such as the completion of an I/O request.

The list of processes waiting for a particular I/O device is called a device queue. Each device has its own device queue (Figure 1.21).

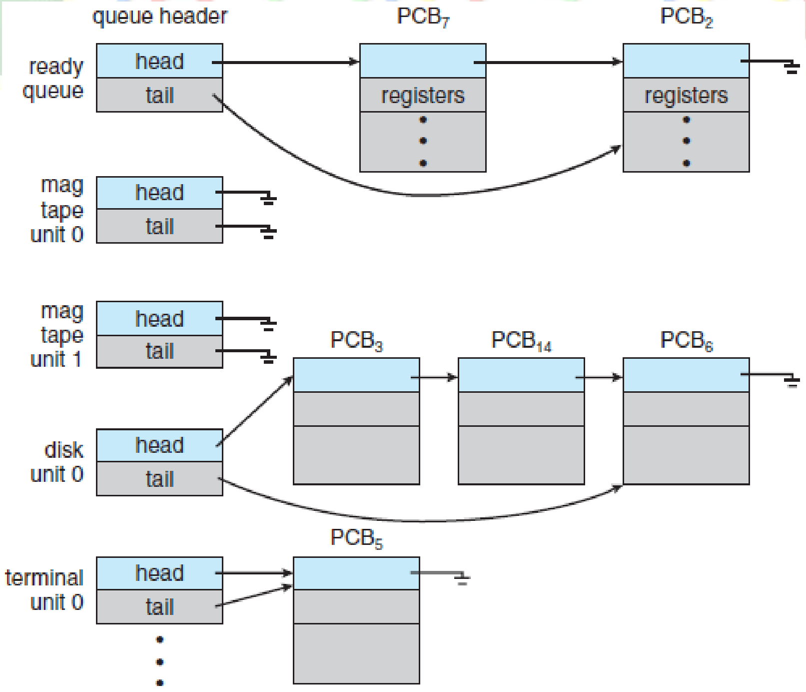


FIGURE 1.21: THE READY QUEUE AND VARIOUS I/O DEVICE QUEUES

A common representation of process scheduling is a **queueing diagram**, such as that in Figure 1.22. Each rectangular box represents a queue. Two types of queues are present: the ready queue and a set of device queues. The circles represent the resources that serve the queues, and the arrows indicate the flow of processes in the system.

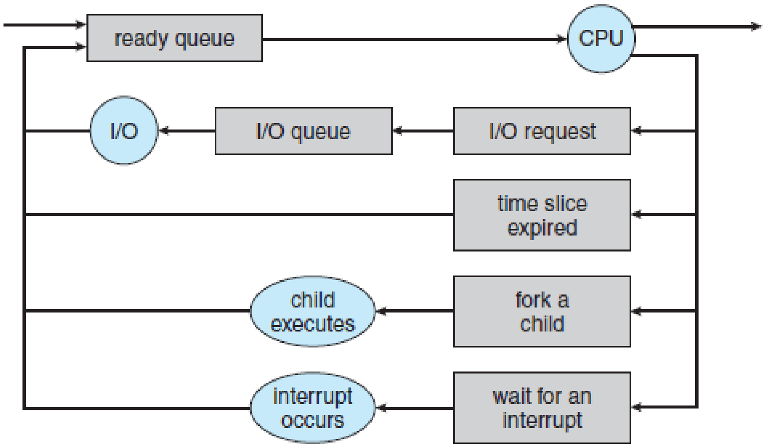


FIGURE 1.22: QUEUEING-DIAGRAM REPRESENTATION OF PROCESS SCHEDULING

A new process is initially put in the ready queue. It waits there until it is selected for execution, or dispatched. Once the process is allocated the CPU and is executing, one of several events could occur:

- The process could issue an I/O request and then be placed in an I/O queue.
- The process could create a new child process and wait for the child's termination.
- The process could be removed forcibly from the CPU, as a result of an interrupt, and be put back in the ready queue.

In the first two cases, the process eventually switches from the waiting state to the ready state and is then put back in the ready queue. A process continues this cycle until it terminates, at which time it is removed from all queues and has its PCB and resources deallocated.

SCHEDULERS: A process migrates among the various scheduling queues throughout its lifetime. The operating system must select, for scheduling purposes, processes from these queues in some fashion.

The selection process is carried out by the appropriate scheduler. Often, in a batch system, more processes are submitted than can be executed immediately. These processes are spooled to a mass-storage device (typically a disk), where they are kept for later execution.

The ***long-term scheduler***, *or* ***job scheduler***, selects processes from this pool and loads them into memory for execution.

The ***short-term scheduler***, *or* ***CPU scheduler***, selects from among the processes that are ready to execute and allocates the CPU to one of them.

The primary distinction between these two schedulers lies in frequency of execution. The short-term scheduler must select a new process for the CPU frequently. A process may execute for only a few milliseconds before waiting for an I/O request.

The long-term scheduler executes much less frequently; minutes may separate the creation of one new process and the next. The long-term scheduler controls the degree of multiprogramming (the number of processes in memory). If the degree of multiprogramming is stable, then the average rate of process creation must be equal to the average departure rate of processes leaving the system.

It is important that the long-term scheduler make a careful selection. In general, most processes can be described as either I/O bound or CPU bound.

An I/O-bound process is one that spends more of its time doing I/O than it spends doing computations. A CPU-bound process, in contrast, generates I/O requests infrequently, using more of its time doing computations.

Some operating systems, such as time-sharing systems, may introduce an additional, intermediate level of scheduling. The key idea behind a medium-term scheduler is that sometimes it can be advantageous to remove a process from memory (and from active contention for the CPU) and thus reduce the degree of multiprogramming. Later, the process can be reintroduced into memory, and its execution can be continued where it left off. This scheme is called swapping.

The process is swapped out, and is later swapped in, by the medium-term scheduler. Swapping may be necessary to improve the process mix or because a change in memory requirements has overcommitted available memory, requiring memory to be freed up.

CONTEXT SWITCH:

Interrupts cause the operating system to change a CPU from its current task and to run a kernel routine. Such operations happen frequently on general-purpose systems. When an interrupt occurs, the system needs to save the current ***context*** of the process running on the CPU so that it can restore that context when its processing is done, essentially suspending the process and then resuming it. The context is represented in the PCB of the process. It includes the value of the CPU registers, the process state, and memory-management information.

Switching the CPU to another process requires performing a state save of the current process and a state restore of a different process. This task is known as a **context switch**.

When a context switch occurs, the kernel saves the context of the old process in its PCB and loads the saved context of the new process scheduled to run. Context-switch time is pure overhead, because the system does no useful work while switching.

Switching speed varies from machine to machine, depending on the memory speed, the number of registers that must be copied, and the existence of special instructions (such as a single instruction to load or store all registers). A typical speed is a few milliseconds. Context-switch times are highly dependent on hardware support.

OPERATIONS ON PROCESSES:

The processes in most systems can execute concurrently, and they may be created and deleted dynamically. Thus, these systems must provide a mechanism for process creation and termination.

PROCESS CREATION:

During the course of execution, a process may create several new processes; the creating process is called a parent process, and the new processes are called the children of that process. Each of these new processes may in turn create other processes, forming a tree of processes.

Most operating systems (including UNIX, Linux, and Windows) identify processes according to a unique **process identifier** (or **pid**), which is typically an integer number. The pid provides a unique value for each process in the system, and it can be used as an index to access various attributes of a process within the kernel.

Figure 1.23 illustrates a typical process tree for the Linux operating system, showing the name of each process and it's pid.

We use the term process rather loosely, as Linux prefers the term task instead. The init process (which always has a pid of 1) serves as the root parent process for all user processes. Once the system has booted, the init process can also create various user processes.

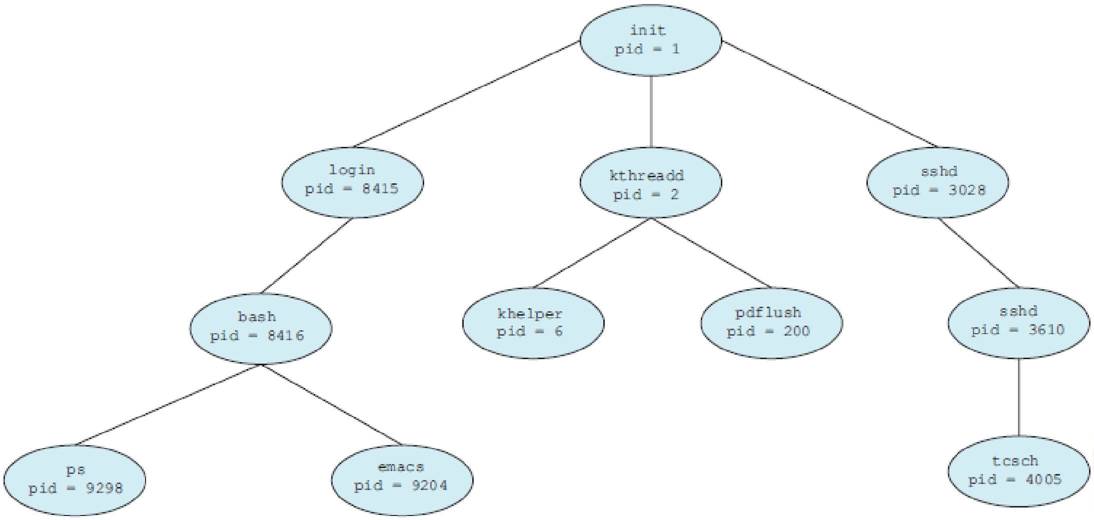


FIGURE 1.23: A TREE OF PROCESSES ON A TYPICAL LINUX SYSTEM

In Figure 1.23, we see two children of init—**kthreadd** and **sshd**. The kthreadd process is responsible for creating additional processes that perform tasks on behalf of the kernel. The sshd process is responsible for managing clients that connect to the system by using ssh (which is short for secure shell).

The login process is responsible for managing clients that directly log onto the system. In this example, a client has logged on and is using the bash shell, which has been assigned pid 8416. Using the bash command-line interface, this user has created the process ***ps*** as well as the ***emacs*** editor.

In general, when a process creates a child process, that child process will need certain resources (CPU time, memory, files, I/O devices) to accomplish its task. A child process may be able to obtain its resources directly from the operating system, or it may be constrained to a subset of the resources of the parent process.

The parent may have to partition its resources among its children, or it may be able to share some resources (such as memory or files) among several of its children. Restricting a child process to a subset of the parent's resources prevents any process from overloading the system by creating too many child processes.

When a process creates a new process, two possibilities for execution exist:

1. The parent continues to execute concurrently with its children.
2. The parent waits until some or all of its children have terminated.

There are also two address-space possibilities for the new process:

1. The child process is a duplicate of the parent process (it has the same program and data as the parent).
2. The child process has a new program loaded into it.

A new process is created by the ***fork()*** system call. The new process consists of a copy of the address space of the original process. This mechanism allows the parent process to communicate easily with its child process.

Both processes (the parent and the child) continue execution at the instruction after the fork(), with one difference: the return code for the fork() is zero for the new (child) process, whereas the (nonzero) process identifier of the child is returned to the parent.

*After a fork() system call, one of the two processes typically uses the **exec()** system call to replace the process's memory space with a new program. The exec() system call loads a binary file into memory (destroying the memory image of the program containing the exec() system call) and starts its execution. In this manner, the two processes are able to communicate and then go their separate ways.* The C program given below illustrates the UNIX system calls previously described.

```
#include <sys/types.h>
#include <stdio.h>
#include <unistd.h>
int main()
{
pid t pid;
```

```
/* fork a child process */
pid = fork();
if (pid < 0) { /* error occurred */
fprintf(stderr, "Fork Failed");
return 1;
}
else if (pid == 0) { /* child process */
execlp("/bin/ls","ls",NULL);
}
else { /* parent process */
/* parent will wait for the child to complete */
wait(NULL);
printf("Child Complete");
}
return 0;
}
```

We now have two different processes running copies of the same program. The only difference is that the value of pid (the process identifier) for the child process is zero, while that for the parent is an integer value greater than zero (in fact, it is the actual pid of the child process). The child process inherits privileges and scheduling attributes from the parent, as well certain resources, such as open files.

The child process then overlays its address space with the UNIX command /bin/ls (used to get a directory listing) using the execlp() system call (execlp() is a version of the exec() system call). The parent waits for the child process to complete with the wait() system call. When the child process completes (by either implicitly or explicitly invoking exit()), the parent process resumes from the call to wait(), where it completes using the exit() system call. This is also illustrated in Figure 1.24.

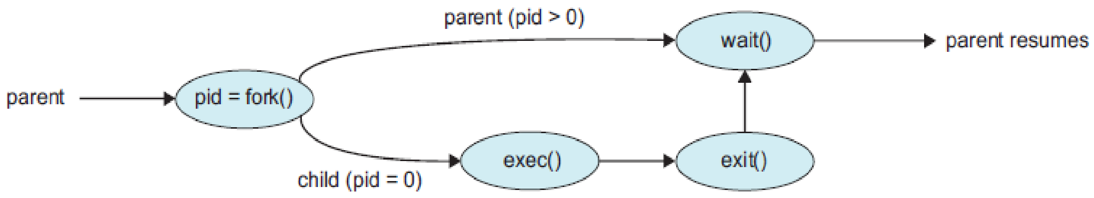


FIGURE 1.24: PROCESS CREATION USING THE fork() SYSTEM CALL

PROCESS TERMINATION:

A process terminates when it finishes executing its final statement and asks the operating system to delete it by using the exit() system call. At that point, the process may return a status value (typically an integer) to its parent process (via the wait() system call).

All the resources of the process—including physical and virtual memory, open files, and I/O buffers—are deallocated by the operating system. Termination can occur in other circumstances as well. A process can cause the termination of another process via an appropriate system call.

Usually, such a system call can be invoked only by the parent of the process that is to be terminated. Otherwise, users could arbitrarily kill each other's jobs. Note that a parent needs to know the identities of its children if it is to terminate them. Thus, when one process creates a new process, the identity of the newly created process is passed to the parent.

A parent may terminate the execution of one of its children for a variety of reasons, such as these:

- The child has exceeded its usage of some of the resources that it has been allocated. (To determine whether this has occurred, the parent must have a mechanism to inspect the state of its children.)
- The task assigned to the child is no longer required.
- The parent is exiting, and the operating system does not allow a child to continue if its parent terminates.

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

INTERPROCESS COMMUNICATION:

Processes executing concurrently in the operating system may be either ***independent*** processes or ***cooperating*** processes. A process is *independent* if it cannot affect or be affected by the other processes executing in the system. Any process that does not share data with any other process is independent.

A process is *cooperating* if it can affect or be affected by the other processes executing in the system. Clearly, any process that shares data with other processes is a cooperating process.

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

- **Information sharing**. Since several users may be interested in the same piece of information (for instance, a shared file), we must provide an environment to allow concurrent access to such information.
- **Computation speedup**. If we want a particular task to run faster, we must break it into subtasks, each of which will be executing in parallel with the others.
    - Notice that such a speedup can be achieved only if the computer has multiple processing cores.
- **Modularity**. We may want to construct the system in a modular fashion, dividing the system functions into separate processes or threads.
- **Convenience**. Even an individual user may work on many tasks at the same time. For instance, a user may be editing, listening to music, and compiling in parallel.

Cooperating processes require an interprocess communication (IPC) mechanism that will allow them to exchange data and information. There are two fundamental models of interprocess communication: ***shared memory*** and ***message passing***.

In the shared-memory model, a region of memory that is shared by cooperating processes is established. Processes can then exchange information by reading and writing data to the shared region.

In the message-passing model, communication takes place by means of messages exchanged between the cooperating processes. The two communications models are contrasted in Figure 1.25.

Both of the models just mentioned are common in operating systems, and many systems implement both. Message passing is useful for exchanging smaller amounts of data, because no conflicts need be avoided. Message passing is also easier to implement in a distributed system than shared memory.

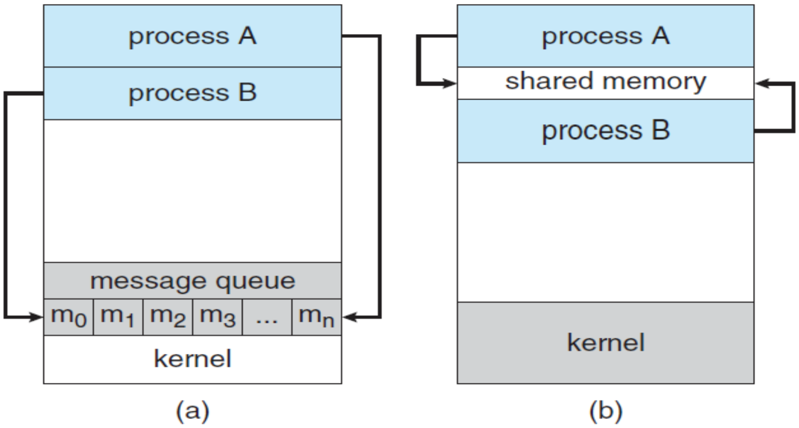


FIGURE 1.25: COMMUNICATIONS MODELS. (A) MESSAGE PASSING. (B) SHARED MEMORY

Shared memory can be faster than message passing, since message-passing systems are typically implemented using system calls and thus require the more time-consuming task of kernel intervention. In shared-memory systems, system calls are required only to establish shared memory regions. Once shared memory is established, all accesses are treated as routine memory accesses, and no assistance from the kernel is required.

SHARED-MEMORY SYSTEMS:

Interprocess communication using shared memory requires communicating processes to establish a region of shared memory. Typically, a shared-memory region resides in the address space of the process creating the shared-memory segment. Other processes that wish to communicate using this shared-memory segment must attach it to their address space.

Normally, the operating system tries to prevent one process from accessing another process’s memory. Shared memory requires that two or more processes agree to remove this restriction. They can then exchange information by reading and writing data in the shared areas. The form of the data and the location are determined by these processes and are not under the operating system’s control. The processes are also responsible for ensuring that they are not writing to the same location simultaneously.

To illustrate the concept of cooperating processes, let’s consider the producer–consumer problem (Figure 1.26), which is a common paradigm for cooperating processes. A producer process produces information that is consumed by a consumer process. For example, a compiler may produce assembly code that is consumed by an assembler. The assembler, in turn, may produce object modules that are consumed by the loader.

```
item next produced;

while (true) {

/* produce an item in next produced */

while (((in + 1) % BUFFER SIZE) == out)

; /* do nothing */

buffer[in] = next produced;

in = (in + 1) % BUFFER SIZE;

}
```

FIGURE 1.26: THE PRODUCER PROCESS USING SHARED MEMORY

One solution to the producer–consumer problem uses shared memory. To allow producer and consumer processes to run concurrently, we must have available a buffer of items that can be filled by the producer and emptied by the consumer. This buffer will reside in a region of memory that is shared by the producer and consumer processes.

A producer can produce one item while the consumer is consuming another item. The producer and consumer must be synchronized, so that the consumer does not try to consume an item that has not yet been produced.

Two types of buffers can be used. The ***unbounded buffer*** places no practical limit on the size of the buffer. The consumer may have to wait for new items, but the producer can always produce new items. The ***bounded buffer*** assumes a fixed buffer size. In this case, the consumer must wait if the buffer is empty, and the producer must wait if the buffer is full.

MESSAGE-PASSING SYSTEMS:

Message passing provides a mechanism to allow processes to communicate and to synchronize their actions without sharing the same address space. It is particularly useful in a distributed environment, where the communicating processes may reside on different computers connected by a network. For example, an Internet chat program could be designed so that chat participants communicate with one another by exchanging messages.

A message-passing facility provides at least two operations: ***send(message) receive(message)***

Messages sent by a process can be either fixed or variable in size. If only fixed-sized messages can be sent, the system-level implementation is straightforward. This restriction, however, makes the task of programming more difficult. Conversely, variable-sized messages require a more complex system level implementation, but the programming task becomes simpler. This is a common kind of tradeoff seen throughout operating-system design.

If processes P and Q want to communicate, they must send messages to and receive messages from each other: a communication link must exist between them. This link can be implemented in a variety of ways. We are concerned here not with the link's physical implementation (such as shared memory, hardware bus, or network) but rather with its logical implementation. Here are several methods for logically implementing a link and the send()/receive() operations:

- Direct or indirect communication
- Synchronous or asynchronous communication
- Automatic or explicit buffering

NAMING:

Processes that want to communicate must have a way to refer to each other. They can use either direct or indirect communication. Under **direct communication**, each process that wants to communicate must explicitly name the recipient or sender of the communication. In this scheme, the send() and receive() primitives are defined as:

- send(P, message)—Send a message to process P.
- receive(Q, message)—Receive a message from process Q.

A communication link in this scheme has the following properties:

- A link is established automatically between every pair of processes that want to communicate. The processes need to know only each other's identity to communicate.

- A link is associated with exactly two processes.
- Between each pair of processes, there exists exactly one link.

This scheme exhibits symmetry in addressing; that is, both the sender process and the receiver process must name the other to communicate. A variant of this scheme employs asymmetry in addressing. Here, only the sender names the recipient; the recipient is not required to name the sender. In this scheme, the send() and receive() primitives are defined as follows:

- send(P, message)—Send a message to process P.
- receive(id, message)—Receive a message from any process. The variable id is set to the name of the process with which communication has taken place.

The disadvantage in both of these schemes (symmetric and asymmetric) is the limited modularity of the resulting process definitions. With indirect communication, the messages are sent to and received from mailboxes, or ports. A mailbox can be viewed abstractly as an object into which messages can be placed by processes and from which messages can be removed. Each mailbox has a unique identification.

For example, POSIX message queues use an integer value to identify a mailbox. A process can communicate with another process via a number of different mailboxes, but two processes can communicate only if they have a shared mailbox. The send() and receive() primitives are defined as follows:

- send(A, message)—Send a message to mailbox A.
- receive(A, message)—Receive a message from mailbox A.

In this scheme, a communication link has the following properties:

- A link is established between a pair of processes only if both members of the pair have a shared mailbox.
- A link may be associated with more than two processes.
- Between each pair of communicating processes, a number of different links may exist, with each link corresponding to one mailbox.

A mailbox may be owned either by a process or by the operating system. If the mailbox is owned by a process (that is, the mailbox is part of the address space of the process), then we distinguish between the owner (which can only receive messages through this mailbox) and the user (which can only send messages to the mailbox).

Since each mailbox has a unique owner, there can be no confusion about which process should receive a message sent to this mailbox. When a process that owns a mailbox terminates, the mailbox disappears. Any process that subsequently sends a message to this mailbox must be notified that the mailbox no longer exists.

In contrast, a mailbox that is owned by the operating system has an existence of its own. It is independent and is not attached to any particular process. The operating system then must provide a mechanism that allows a process to do the following:

- Create a new mailbox.
- Send and receive messages through the mailbox.
- Delete a mailbox.

The process that creates a new mailbox is that mailbox's owner by default. Initially, the owner is the only process that can receive messages through this mailbox. However, the ownership and receiving privilege may be passed to other processes through appropriate system calls. Of course, this provision could result in multiple receivers for each mailbox.

SYNCHRONIZATION:

Communication between processes takes place through calls to send() and receive() primitives. There are different design options for implementing each primitive. Message passing may be **blocking** or **nonblocking**— also known as synchronous and asynchronous.

- **Blocking send**. The sending process is blocked until the message is received by the receiving process or by the mailbox.
- **Nonblocking send**. The sending process sends the message and resumes operation.
- **Blocking receive**. The receiver blocks until a message is available.

- **Nonblocking receive**. The receiver retrieves either a valid message or a null.

Different combinations of send() and receive() are possible. When both send() and receive() are blocking, we have a rendezvous between the sender and the receiver. The solution to the producer–consumer problem becomes trivial when we use blocking send() and receive() statements. The producer merely invokes the blocking send() call and waits until the message is delivered to either the receiver or the mailbox. Likewise, when the consumer invokes receive(), it blocks until a message is available. This is illustrated in Figures 1.27 and 1.28.

```
message next produced;
while (true) {
/* produce an item in next produced */
send(next produced);
}
```

FIGURE 1.27 THE PRODUCER PROCESS USING MESSAGE PASSING

```
message next consumed;
while (true) {
receive(next consumed);
/* consume the item in next consumed */
}
```

FIGURE 1.28 THE CONSUMER PROCESS USING MESSAGE PASSING

BUFFERING:

Whether communication is direct or indirect, messages exchanged by communicating processes reside in a temporary queue. Basically, such queues can be implemented in three ways:

- Zero capacity. The queue has a maximum length of zero; thus, the link cannot have any messages waiting in it. In this case, the sender must block until the recipient receives the message.

- Bounded capacity. The queue has finite length n; thus, at most n messages can reside in it. If the queue is not full when a new message is sent, the message is placed in the queue (either the message is copied or a pointer to the message is kept), and the sender can continue execution without waiting. The link's capacity is finite, however. If the link is full, the sender must block until space is available in the queue.
- Unbounded capacity. The queue's length is potentially infinite; thus, any number of messages can wait in it. The sender never blocks.

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