



# UNIT-1

## Introduction

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

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

Internally, operating systems vary greatly in their makeup, since they are organized along many different lines. The design of a new operating system is a major task. It is important that the goals of the system be well defined before the design begins. These goals form the basis for choices among various algorithms and strategies. Because an operating system is large and complex, it must be created piece by piece. Each of these pieces should be a well-delineated portion of the system, with carefully defined inputs, outputs, and functions.

## OBJECTIVES

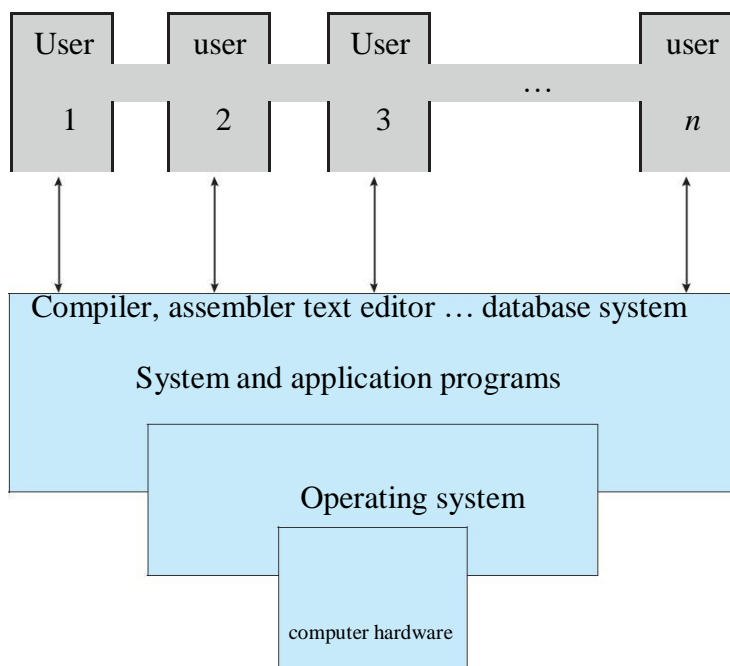
1. To describe the basic organization of computersystems.
2. To provide a grand tour of the major components of operating systems.
3. To give an overview of the many types of computing environments.
4. To explore several open-source operating systems.
5. To describe the services an operating system provides to users, processes, and others ystems.
6. To discuss the various ways of structuring an operatingsystem.
7. To explain how operating systems are installed and customized and how they boot.

## 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.

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.



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

## What Operating SystemsDo

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

The **hardware** — the **central processing unit (CPU)**, the **memory**, and the **input/output (I/O) devices** — provides the basic computing resources for the system. The **application programs** — such as word processors, spreadsheets, compilers, and Web browsers — define the ways in which these resources are used to solve users' computing problems. The operating system controls the hardware and coordinates its use among the various application programs for the various users.

We can also view a computer system as consisting of hardware, software, and data. The operating system provides the means for proper use of these resources in the operation of the computer system. An operating system is similar

to a government. Like a government, it performs no useful function by itself. It

simply provides an **environment** within which other programs can do useful work.

To understand more fully the operating system's role, we next explore operating systems from two viewpoints: that of the user and that of the system.

## UserView

The user's view of the computer varies according to the interface being used. Most computer users sit in front of a PC, consisting of a monitor, keyboard, mouse, and system unit. Such a system is designed for one user to monopolize its resources. The goal is to maximize the work (or play) that the user is performing. In this case, the operating system is designed mostly for **ease of use**, with some attention paid to performance and none paid to **resource utilization**, how various hardware and software resources are shared. Performance is, important to the user; but such systems are optimized for the single-user experience rather than the requirements of multiple users.

In other cases, a user sits at a terminal connected to a **mainframe** or a **minicomputer**. Other users are accessing the same computer through other terminals. These users share resources and may exchange information. The operating system in such cases is designed to maximize resource utilization — to assure that all available CPU time, memory, and I/O are used efficiently and that no individual user takes more than her fair share.

In still other cases, users sit at **workstations** connected to networks of other workstations and **servers**. These users have dedicated resources at their disposal, but they also share resources such as networking and servers, including file, compute, and print servers. Therefore, their operating system is designed to compromise between individual usability and resource utilization.

Recently, many varieties of mobile computers, such as smartphones and tablets, have come into fashion. Most mobile computers are standalone units for individual users. Quite often, they are connected to networks through cellular or other wireless technologies. Increasingly, these mobile devices are replacing desktop and laptop computers for people who are primarily interested in using computers for e-mail and web browsing. The user interface for mobile computers generally features a **touch screen**, where the user interacts with the system by pressing and swiping fingers across the screen rather than using a physical keyboard and mouse.

### 1.1.2. System View

From the computer's point of view, the operating system is the program most intimately involved with the hardware. In this context, we can view an operating system as a **resource allocator**. A computer system has many resources that may be required to solve a problem: CPU time, memory space, file-storage space, I/O devices, and so on. The operating system acts as the manager of these resources. Facing numerous and possibly conflicting requests for resources, the operating system must decide how to allocate them to specific programs and users so that it can operate the computer system efficiently and fairly.

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

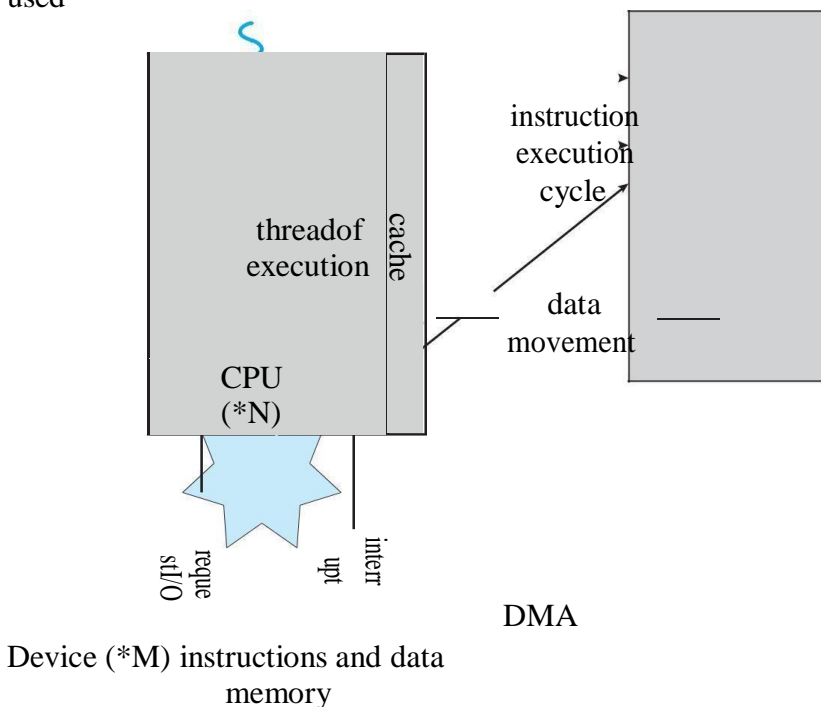
## I/O Structure

Storage is only one of many types of I/O devices within a computer. A large portion of operating system code is dedicated to managing I/O, both because of its importance to the reliability and performance of a system and because of the varying nature of the devices. Next, we provide an overview of I/O.

To start an I/O operation, the device driver loads the appropriate registers within the device controller. The device controller, in turn, examines the contents of these registers to determine what action to take (such as read a character from the keyboard). The controller starts the transfer of data from the device to its local buffer. Once the transfer of data is complete, the device controller informs the device driver via an interrupt that it has finished its operation. The device driver then returns control to the operating system, possibly returning the data or a pointer to the data if the operation was a read. For other operations, the device driver returns status information.

## Computer-System Architecture

A computer system can be organized in a number of different ways, which we can categorize roughly according to the number of general-purpose processors used



**Figure 1.5** How a modern computer system works.

### 1.3.1. Single-Processor Systems

On a single-processor system, there is one main CPU capable of executing a general-purpose instruction set, including instructions from user processes. Almost all single-processor systems have other special-purpose processors as well. They may come in the form of device-specific processors, such as disk, keyboard, and graphics controllers; or, on mainframes, they may come in the form of more general-purpose processors, such as I/O processors that move data rapidly among the components of the system.

All of these special-purpose processors run a limited instruction set and do not run user processes. Sometimes, they are managed by the operating system, in that the operating system sends them information about their next task and monitors their status. For example, a disk-controller microprocessor receives a sequence of requests from the main CPU and implements its own disk queue and scheduling algorithm. This arrangement relieves the main CPU of the overhead of disk scheduling. PCs contain a microprocessor in the keyboard to convert the keystrokes into codes to be sent to the CPU. In other systems or circumstances, special-purpose processors are low-level components built into the hardware. The operating system cannot communicate with these processors; they do their jobs autonomously. The use of special-purpose microprocessors is common and does not turn a single-processor system into a multiprocessor. If there is only one general-purpose CPU, then the system is a single-processor system.

### 1.3.2 Multiprocessor Systems

Within the past several years, **multiprocessor systems** (also known as **parallel systems** or **multicore systems**) have begun to dominate the landscape of computing. Such systems have two or more processors in close communication, sharing the computer bus and sometimes the clock, memory, and peripheral devices. Multiprocessor systems first appeared prominently in servers and have since migrated to desktop and laptop systems. Recently, multiple processors have appeared on mobile devices such as smartphones and tablet computers.

Multiprocessor systems have three main advantages:

**Increased throughput.** By increasing the number of processors, we expect to get more work done in less time. The speed-up ratio with  $N$  processors is not  $N$ , however; rather, it is less than  $N$ . When multiple processors cooperate on a task, a certain amount of overhead is incurred in keeping all the parts working correctly. This overhead, plus contention for shared resources, lowers the expected gain from additional processors. Similarly,  $N$  programmers working closely together do not produce  $N$  times the amount of work a single programmer would produce.

**Economy of scale.** Multiprocessor systems can cost less than equivalent multiple single-processor systems, because they can share peripherals, mass storage, and power supplies.

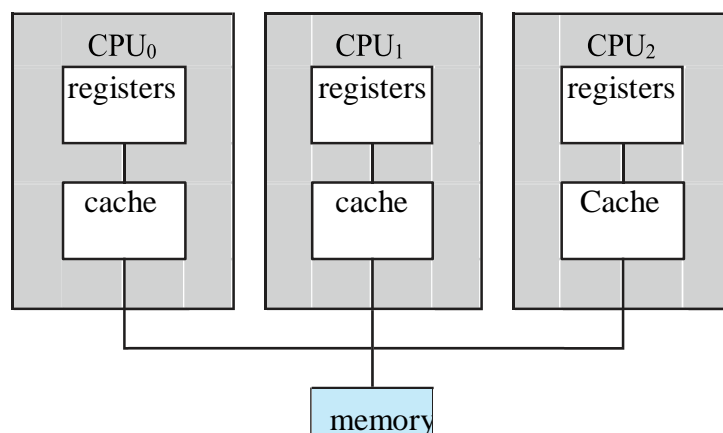
**Increased reliability.** If functions can be distributed properly among several processors, then the failure of one processor will not halt the system, only slow it down. If we have ten processors and one fails, then each of the remaining nine processors can pick up a share of the work of the failed processor. Thus, the entire system runs only 10 percent slower, rather than failing altogether.

The multiple-processor systems in use today are of two types. Some systems use **asymmetric multiprocessing**, in which each processor is assigned a specific task. A **boss** processor controls the system; the other processors either look to the boss for instruction or have predefined tasks. This scheme defines a boss – worker relationship. The boss processor schedules and allocates work to the worker processors.

The most common systems use **symmetric multiprocessing (SMP)**, in which each processor performs all tasks within the operating system. SMP means that all processors are peers; no boss – worker relationship exists between processors. Figure 1.6 illustrates a typical SMP architecture. Notice that each processor has its own set of registers, as well as a private — or local — cache. A multiprocessor system of this form will allow processes and resources — such as memory — to be shared dynamically among the various processors and can lower the variance among the processors.

The difference between symmetric and asymmetric multiprocessing may result from either hardware or software. Special hardware can differentiate the multiple processors, or the software can be written to allow only one boss and multiple workers. For instance, Sun Microsystems' operating system SunOS Version 4 provided asymmetric multiprocessing, whereas Version 5 (Solaris) is symmetric on the same hardware.

Multiprocessing adds CPUs to increase computing power. If the CPU has an integrated memory controller, then adding CPUs can also increase the amount of memory addressable in the system.



**Figure 1.6** Symmetric multiprocessing architecture.

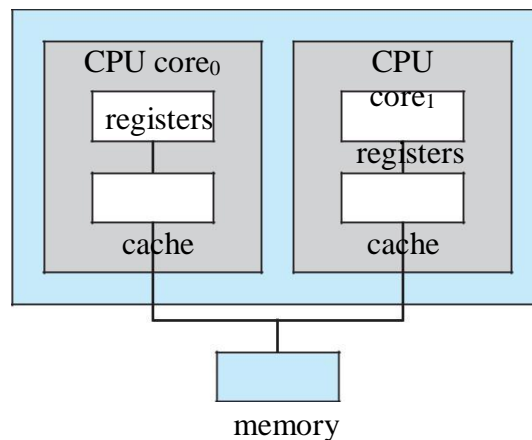
Either way, multiprocessing can cause a system to change its memory access model from uniform memory access (UMA) to non-uniform memory



access (**NUMA**). UMA is defined as the situation in which access to any RAM from any CPU takes the same amount of time. With NUMA, some parts of memory may take longer to access than other parts, creating a performance penalty.

A recent trend in CPU design is to include multiple computing **cores** on a single chip. Such multiprocessor systems are termed **multicore**. They can be more efficient than multiple chips with single cores because on-chip communication is faster than between-chip communication. In addition, one chip with multiple cores uses significantly less power than multiple single-core chips.

Finally, **blade servers** are a relatively recent development in which multiple processor boards, I/O boards, and networking boards are placed in the same chassis.



**Figure 1.7A** dual-core design with two cores placed on the same chip.

### 1.3.3. Clustered Systems

Another type of multiprocessor system is a **clustered system**, which gathers together multiple CPUs. Clustered systems differ from the multiprocessor systems, in that they are composed of two or more individual systems — or nodes — joined together. Such systems are considered **loosely coupled**. Each node may be a single processor system or a multicore system.

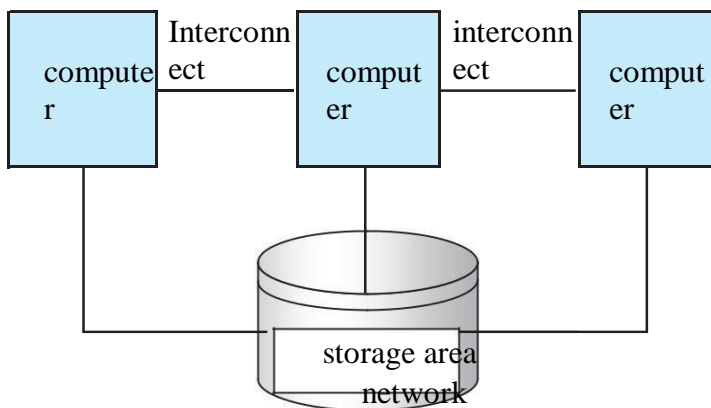
Clustering is usually used to provide **high-availability** service — that is, service will continue even if one or more systems in the cluster fail. Generally, we obtain high availability by adding a level of redundancy in the system. A layer of cluster software runs on the cluster nodes. Each node can monitor one or more of the others (over the LAN). If the monitored machine fails, the monitoring machine can take ownership of its storage and restart the applications that were running on the failed machine. The users and clients of the applications see only a brief interruption of service.

Clustering can be structured asymmetrically or symmetrically. In **asymmetric clustering**, one machine is in **hot-standby mode** while the other is running the applications. The hot-standby host machine does nothing but monitor the active server. If that server fails, the hot-standby host becomes the active server. In **symmetric clustering**, two or more hosts are running applications and are monitoring each other. This structure is obviously more efficient, as it uses all of the available hardware. However it does require that more than one application be available to run.

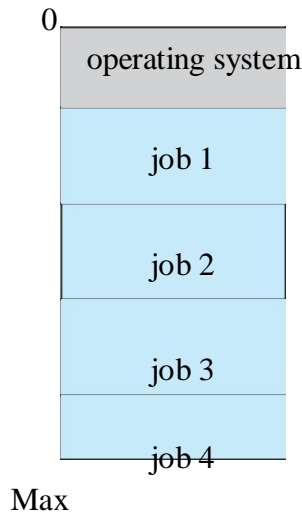
Since a cluster consists of several computer systems connected via a network, clusters can also be used to provide **high-performance computing** environments. Such systems can supply significantly greater computational power than single-processor or even SMP systems because they can run an application concurrently on all computers in the cluster. The application must have been written specifically to take advantage of the cluster, however. This involves a technique known as **parallelization**, which divides a program into separate components that run in parallel on individual computers in the cluster.

Parallel clusters allow multiple hosts to access the same data on shared storage. Because most operating systems lack support for simultaneous data access by multiple hosts, parallel clusters usually require the use of special versions of software and special releases of applications. For example, Oracle Real Application Cluster is a version of Oracle's database that has been designed to run on a parallel cluster. Each machine runs Oracle, and a layer of software tracks access to the shared disk. Each machine has full access to all data in the database.

Cluster technology is changing rapidly. Some cluster products support dozens of systems in a cluster, as well as clustered nodes that are separated by miles. Many of these improvements are made possible by **storage-area networks (SANs)**, which allow many systems to attach to a pool of storage. If the applications and their data are stored on the SAN, then the cluster software can assign the application to run on any host that is attached to the SAN. If the host fails, then any other host can takeover.



**Figure 1.8** General structure of a clustered system



**Figure 1.9** Memory layout for a multiprogramming system.

## 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 (Figure 1.9). 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.

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. When the operating system selects a job from the job pool, it loads that job into memory for execution. 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.

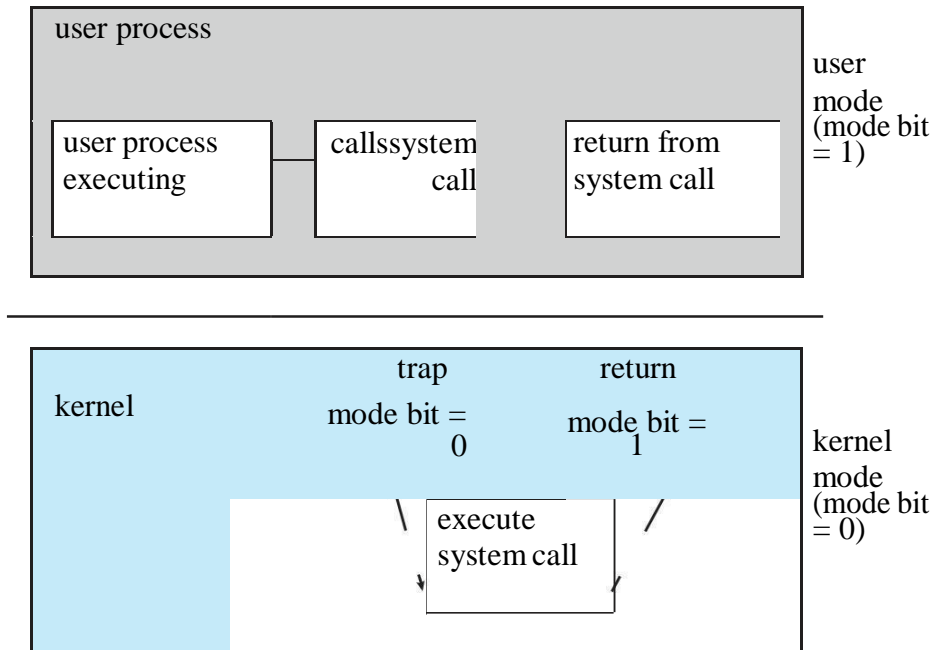
## Operating-System Operations

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.

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.

## 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.



**Figure 1.10** Transition from user to kernel mode.

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 1.10. As we shall see, this architectural enhancement is useful for many other aspects of system operation as well.

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). 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.

System calls provide the means for a user program to ask the operating system to perform tasks reserved for the operating system on the user program's behalf. A system call is invoked in a variety of ways, depending on the functionality provided by the underlying processor. In all forms, it is the method used by a process to request action by the operating system. A system call usually takes the form of a trap to a specific location in the interrupt vector. This trap can be executed by a generic trap instruction, although some systems (such as MIPS) have a specific syscall instruction to invoke a system call.

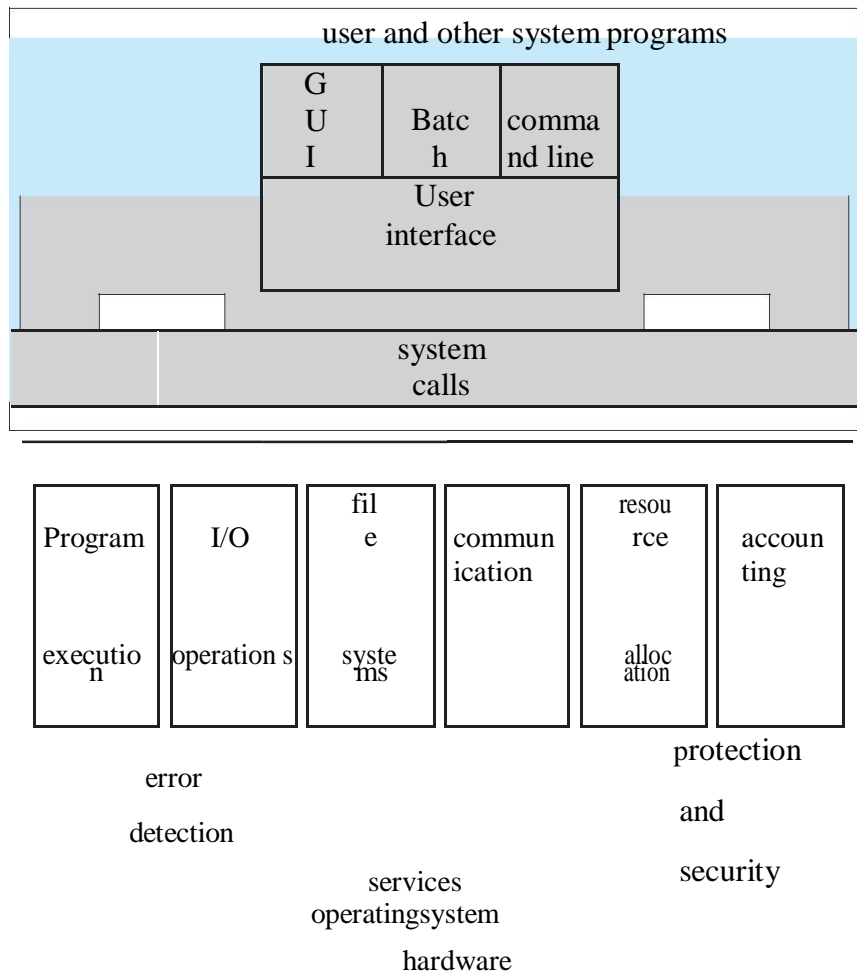
## Timer

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.

## 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



**Figure2.1** A view of operating systemservices.

task easier. Figure 2.1 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.

**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 windows 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. Many operating systems provide a variety of file systems, sometimes to allow personal choice and sometimes to provide specific features or performance characteristics.

**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. Sometimes, it has no choice but to halt the system. At other times, it might terminate an error-causing process or return an error code to a process for the process to detect and possibly correct.

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 himself or herself 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. If a system is to be protected and secure, precautions must be instituted throughout it. A chain is only as strong as its weakest link.

## User and Operating-System Interface

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.

## SystemCalls

**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 systemcalls.

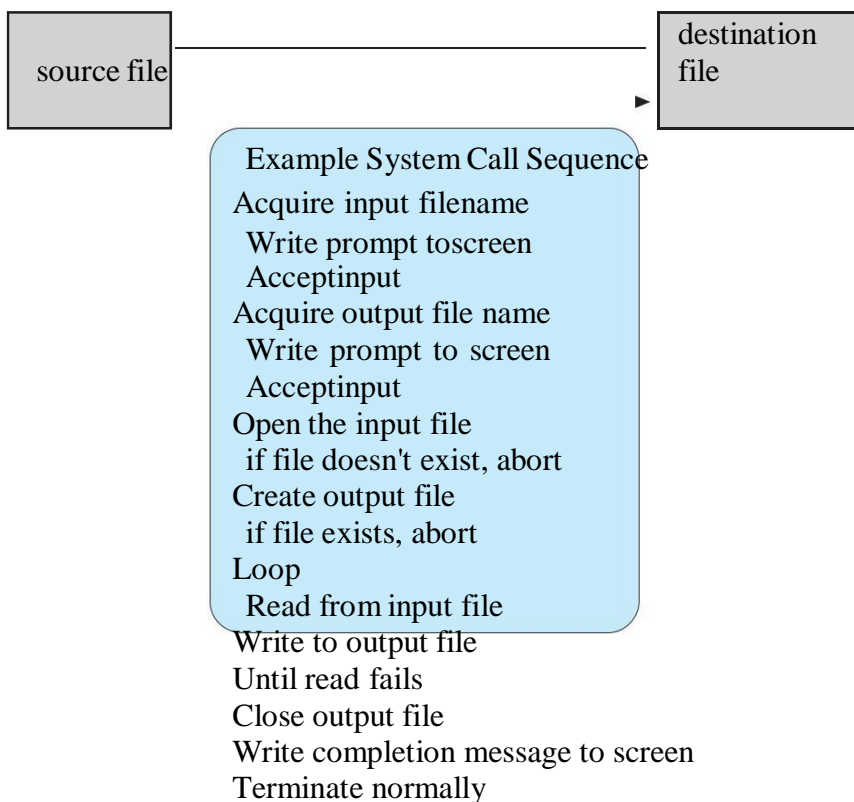
Once the two file names have been obtained, the program must open the input file and create the output file. Each of these operations requires another system call. Possible error conditions for each operation can require additional system calls. When the program tries to open the input file, for example, 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). Another option, in an interactive system, is to ask the user (via a sequence of system calls to output the prompting message and to read the response from the terminal) whether to replace the existing file or to abort the program.

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

Finally, after the entire file is copied, the program may close both files (another system call), write a message to the console or window (more system calls), and finally terminate normally (the final system call). This system-call sequence is shown in Figure 2.5.

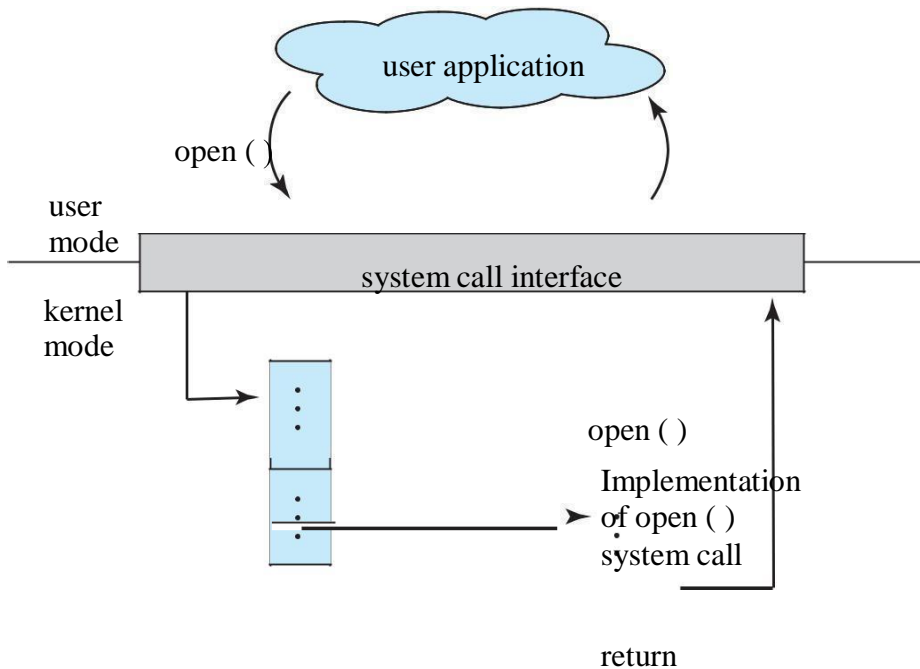
Typically, application developers design programs according to an **application programming interface (API)**. The API specifies a set of functions that are available to an application programmer, including the parameters that are passed to each function and the return values the programmer can expect. 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 OS X), 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.

There are several reasons for doing so. 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.



**Figure 2.5** Example of how system calls are used.

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.



**Figure 2.6** The handling of a user application invoking the `open()` system call.

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

## 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 2.8 summarizes the types of system calls normally provided by an operating system.

## ProcessControl

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. 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 errorparameter.

### 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

**Figure2.8** Types of systemcalls.

It is then possible to combine normal and abnormal termination by defining a normal termination as an error at level 0. The command interpreter or a following program can use this error level to determine the next action automatically.

A process or job executing one program may want to load() and execute() another program.

### ***EXAMPLES OF WINDOWS AND UNIX SYSTEM CALLS***

|                                | <b>Windows</b>   | <b>Unix</b>                            |
|--------------------------------|--|--|
| <b>Process Control</b>         | CreateProcess()<br>ExitProcess()<br>WaitForSingleObject()  | fork()<br>exit()<br>wait()             |
| <b>File Manipulation</b>       | CreateFile()<br>ReadFile()<br>WriteFile()<br>CloseHandle() | open()<br>read()<br>write()<br>close() |
| <b>Device Manipulation</b>     | SetConsoleMode()<br>ReadConsole()<br>WriteConsole()        | ioctl()<br>read()<br>write()           |
| <b>Information Maintenance</b> | GetCurrentProcessID()<br>SetTimer()<br>Sleep()             | getpid()<br>alarm()<br>sleep()         |
| <b>Communication</b>           | CreatePipe()<br>CreateFileMapping()<br>MapViewOfFile()     | pipe()<br>shm open()<br>mmap()         |



|                   |                                |         |
|-------------------|--------------------------------|---------|
| <b>Protection</b> | SetFileSecurity()              | chmod() |
|                   | InitializeSecurityDescriptor() | umask() |
|                   | SetSecurityDescriptorGroup()   | chown() |

An interesting question is where to return control when the loaded program terminates. This question is related to whether the existing program is allowed to continue execution concurrently with the new program.

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. 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()).

### **FileManagement**

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().

## DeviceManagement

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.

## 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 soon.

Many operating systems provide a time profile of a program to indicate the amount of time that the program executes at a particular location or set of locations. A time profile requires either a tracing facility or regular timer interrupts. At every occurrence of the timer interrupt, the value of the program counter is recorded. With sufficiently frequent timer interrupts, a statistical picture of the time spent on various parts of the program can be obtained.

In addition, the operating system keeps information about all its processes, and system calls are used to access this information. Generally, calls are also used to reset the process information (get process attributes() and set process attributes()).

## Communication

There are two common models of interprocess communication: the message-passing model and the shared-memory model. In the **message-passing model**, the communicating processes exchange messages with one another to transfer information. Messages can be exchanged between the processes either directly or indirectly through a common mailbox. Before communication can take place, a connection must be opened. The name of the other communicator must be known, be it another process on the same system or a process on another computer connected by a communications network. Each computer in a network has a **host name** by which it is commonly known. A host also has a network identifier, such as an IP address. Similarly, each process has a **process name**, and this name is translated into an identifier by which the operating system can refer to the process. The gethostid() and getprocessid() system calls do this translation. The identifiers are then passed to the general-purpose open() and

close() calls provided by the file system or to specific open connection() and close connection() system calls, depending on the system's model of communication. The recipient process usually must give its permission for communication to take place with an accept connection() call. Most processes that will be receiving connections are special-purpose **daemons**, which are system programs provided for that purpose. They execute a wait for connection() call and are awakened when a connection is made. The source of the communication, known as the **client**, and the receiving daemon, known as a **server**, then exchange messages by using read message() and write message() system calls. The close connection() call terminates the communication.

In the **shared-memory model**, processes use shared memory create() and shared memory attach() system calls to create and gain access to regions of memory owned by other processes. 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.

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

## 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

**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.

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.

## 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

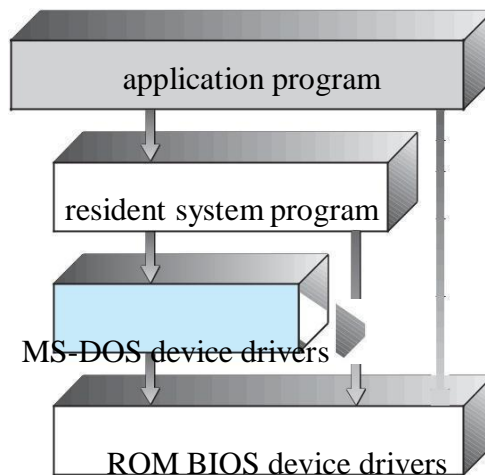
**monolithic** system. Each of these modules should be a well-defined portion of the system, with carefully defined inputs, outputs, and functions. In this section, we discuss how these components are interconnected and melded into a kernel.

### Simple Structure

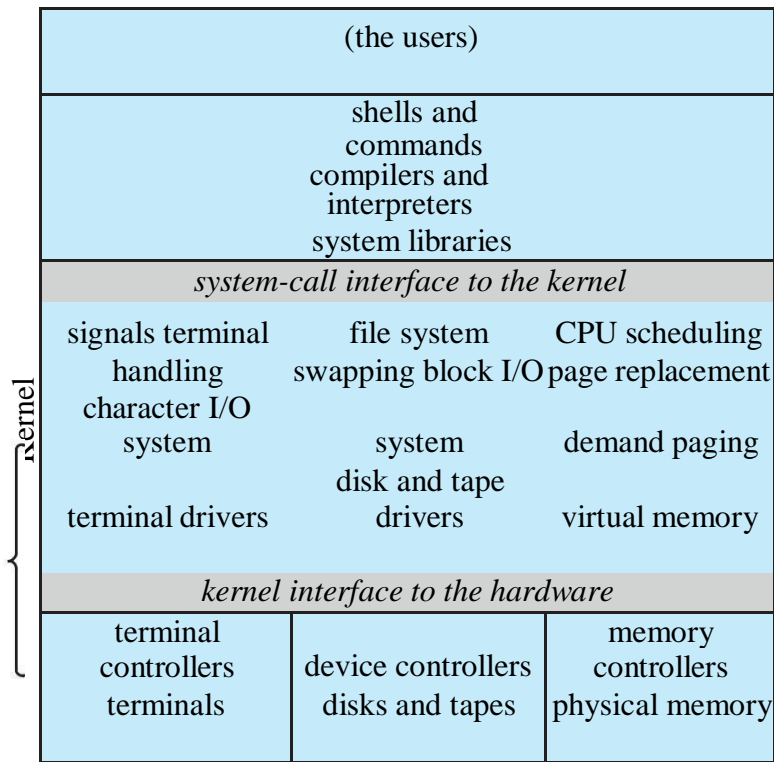
Many operating systems do not have well-defined structures. Frequently, such systems started as small, simple, and limited systems and then grew beyond their original scope. MS-DOS is an example of such a system. It was originally designed and implemented by a few people who had no idea that it would become so popular. It was written to provide the most functionality in the least space, so it was not carefully divided into modules. Figure 2.11 shows its 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



**Figure 2.11** MS-DOS layer structure.



**Figure2.12** Traditional UNIX system structure.

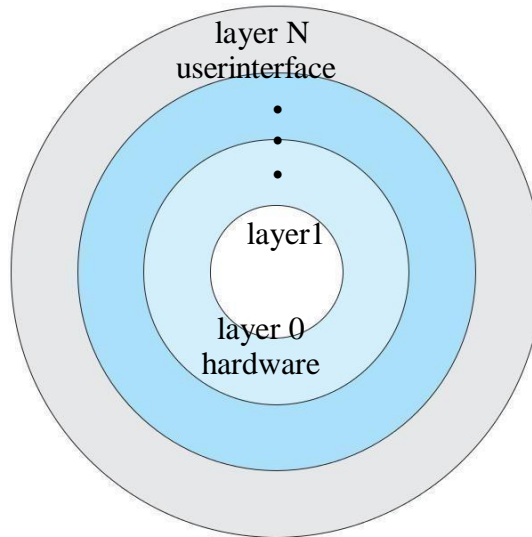
is further separated into a series of interfaces and device drivers, which have been added and expanded over the years as UNIX has evolved. We can view the traditional UNIX operating system as being layered to some extent, as shown in Figure 2.12. Everything below the system-call interface and above the physical hardware is the kernel. The kernel provides the file system, CPU scheduling, memory management, and other operating-system functions through systemcalls.

**LayeredApproach**

With proper hardware support, operating systems can be broken into pieces that are smaller and more appropriate than those allowed by the original MS-DOS and UNIX systems. The operating system can then retain much greater control over the computer and over the applications that make use of that computer. Implementers have more freedom in changing the inner workings of the system and in creating modular operating systems. Under a top-down approach, the overall functionality and features are determined and are separated into components. Information hiding is also important, because it leaves programmers free to implement the low-level routines as they see fit,

provided that the external interface of the routine stays unchanged and that the routine itself performs the advertised task.

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 2.13.



**Figure 2.13** 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. For example, the device driver for the backing store (disk space used by virtual-memory algorithms) must be at a lower level than the memory-management routines, because memory management requires the ability to use the backing store.

A final problem with layered implementations is that they tend to be less efficient than other types. For instance, when a user program executes an I/O operation, it executes a system call that is trapped to the I/O layer, which calls the memory-management layer, which in turn calls the CPU-scheduling layer, which is then passed to the hardware. At each layer, the parameters may be modified, data may need to be passed, and so on. Each layer adds overhead to the system call. The net result is a system call that takes longer than does one on a nonlayered system.

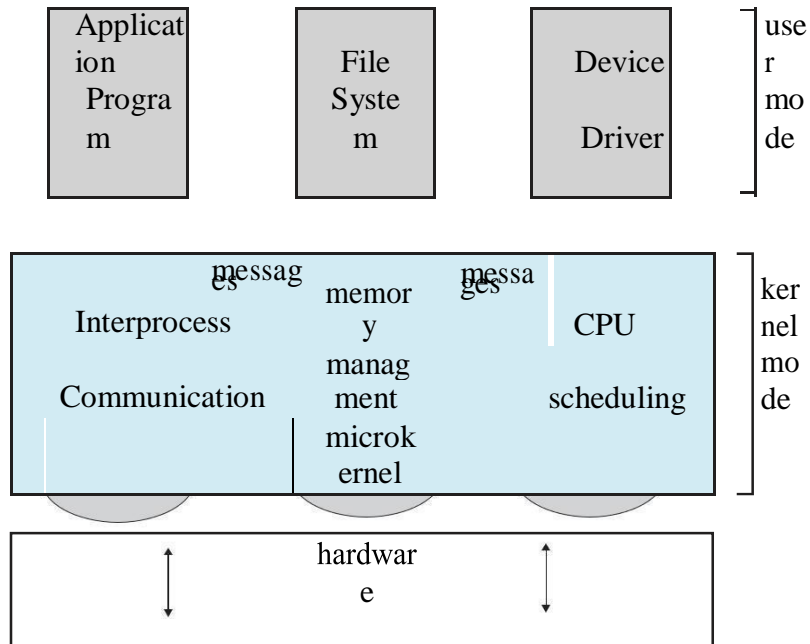
## Microkernels

We have already seen that as UNIX expanded, the kernel became large and difficult to manage. 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.. Figure 2.14 illustrates the 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**, which was described in Section 2.4.5. For example, if the client program wishes to access a file, it must interact with the file server. The client program and service never interact directly. Rather, they communicate indirectly by exchanging messages with the microkernel.

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.





**Figure2.14** Architecture of a typicalmicrokernel.

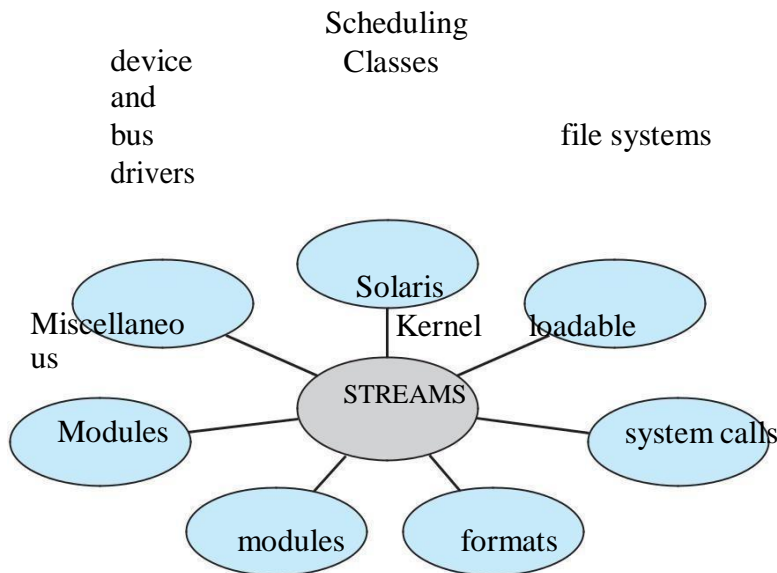
Unfortunately, the performance of microkernels can suffer due to increased system-function overhead. Consider the history of Windows NT. The first release had a layered microkernel organization. This version's performance was low compared with that of Windows 95. Windows NT 4.0 partially corrected the performance problem by moving layers from user space to kernel space and integrating them more closely. By the time Windows XP was designed, Windows architecture had become more monolithic than microkernel.

## 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.



**Figure 2.15** Solaris loadable modules.

The Solaris operating system structure, shown in Figure 2.15, is organized around a core kernel with seven types of loadable kernel modules:

Scheduling classes

File systems

Loadable system calls

Executable formats

STREAMS modules

Miscellaneous

Device and bus drivers

Linux also uses loadable kernel modules, primarily for supporting device drivers and file systems. We cover creating loadable kernel modules in Linux as a programming exercise at the end of this chapter.

## 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. In this, we explore the structure of three hybrid systems: the Apple Mac OS X operating system and the two most prominent mobile operating systems — iOS and Android.

### Mac OS X

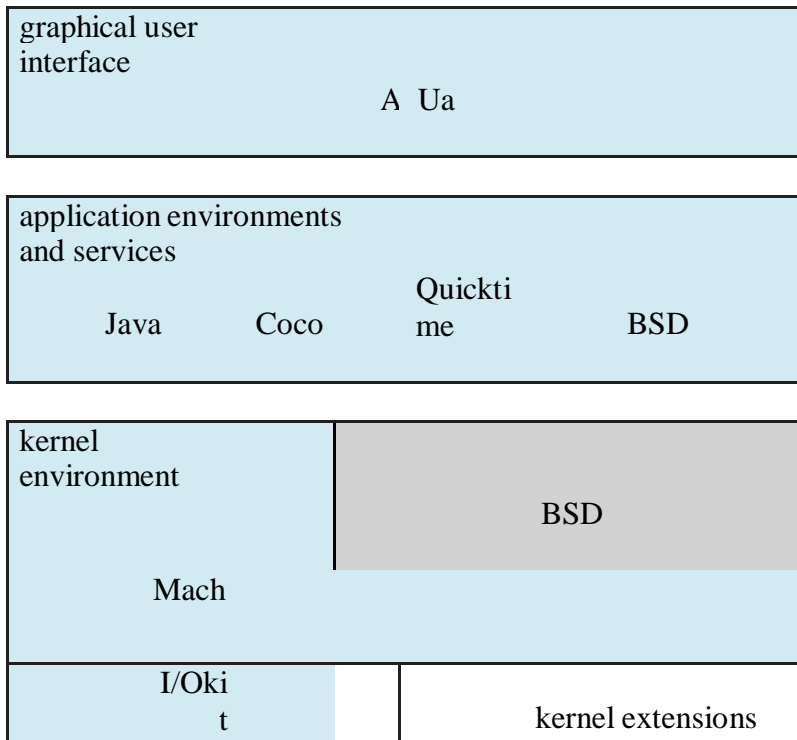
The Apple Mac OS X operating system uses a hybrid structure. As shown in Figure 2.16, it is a layered system. The top layers include the *Aqua* user interface (Figure 2.4) and a set of application environments and services. Notably, the **Cocoa** environment specifies an API for the Objective-C programming language, which is used for writing Mac OS X applications. Below these layers is the *kernel environment*, which consists primarily of the Mach microkernel and the BSD UNIX kernel. Mach provides memory management; support for remote procedure calls (RPCs) and interprocess communication (IPC) facilities, including message passing; and thread scheduling. The BSD component provides a BSD command-line interface, support for networking and file systems, and an implementation of POSIX APIs, including Pthreads. In addition to Mach and BSD, the kernel environment provides an I/O kit for development of device drivers and dynamically loadable modules (which Mac OS X refers to as **kernel extensions**). As shown in Figure 2.16, the BSD application environment can make use of BSD facilities directly.

### iOS

iOS is a mobile operating system designed by Apple to run its smartphone, the *iPhone*, as well as its tablet computer, the *iPad*. iOS is structured on the Mac OS X operating system, with added functionality pertinent to mobile devices, but does not directly run Mac OS X applications. The structure of iOS appears in Figure 2.17.

**Cocoa Touch** is an API for Objective-C that provides several frameworks for developing applications that run on iOS devices. The fundamental difference

between Cocoa, mentioned earlier, and Cocoa Touch is that the latter provides support for hardware features unique to mobile devices, such as touch screens. The **media services** layer provides services for graphics, audio, and video.



**Figure 2.16** The Mac OS X structure.

The **core services** layer provides a variety of features, including support for cloud computing and databases. The bottom layer represents the core operating system, which is based on the kernel environment shown in Figure 2.16.

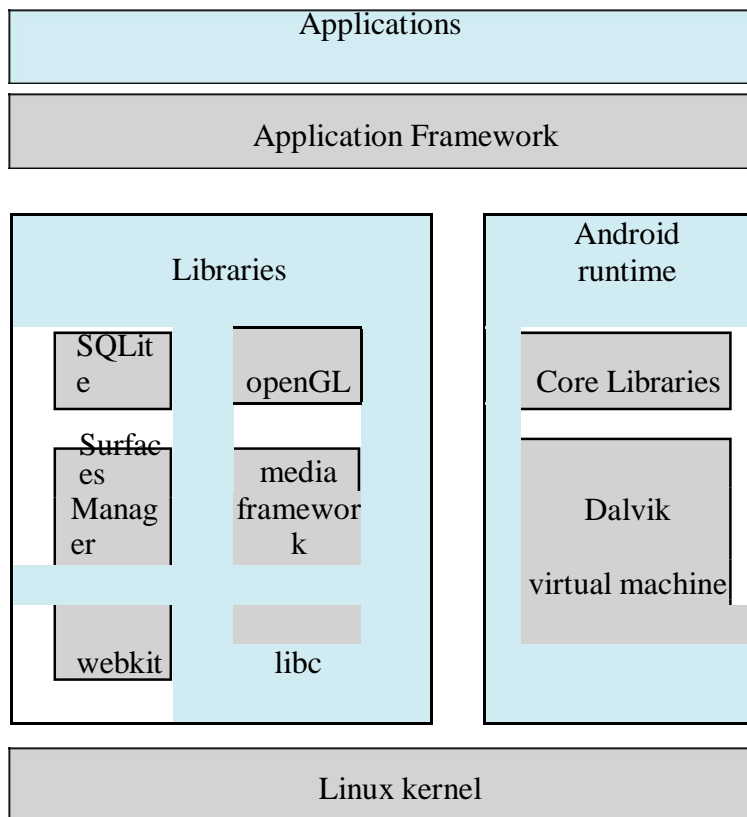


**Figure 2.17** Architecture of Apple's iOS.

## Android

The Android operating system was designed by the Open Handset Alliance (led primarily by Google) and was developed for Android smartphones and tablet computers. Whereas iOS is designed to run on Apple mobile devices and is close-sourced, Android runs on a variety of mobile platforms and is open-sourced, partly explaining its rapid rise in popularity. The structure of Android appears in Figure 2.18.

Android is similar to iOS in that it is a layered stack of software that provides a rich set of frameworks for developing mobile applications. At the bottom of this software stack is the Linux kernel, although it has been modified by Google and is currently outside the normal distribution of Linux releases.



**Figure 2.18** Architecture of Google's Android.

Linux is used primarily for process, memory, and device-driver support for hardware and has been expanded to include power management. The Android runtime environment includes a core set of libraries as well as the Dalvik virtual machine. Software designers for Android devices develop applications in the Java language. However, rather than using the standard Java API, Google has designed a separate Android API for Java development. The Java class files are first compiled to Java bytecode and then translated into an executable file that runs on the Dalvik virtual machine. The Dalvik virtual machine was designed for Android and is optimized for mobile devices with limited memory and CPU processing capabilities.

The set of libraries available for Android applications includes frameworks for developing web browsers (webkit), database support (SQLite), and multimedia. The libc library is similar to the standard C library but is much smaller and has been designed for the slower CPUs that characterize mobile devices.

## **Batch Systems:**

**Batch Operating system** is one of the important type of operating system.

The users who using a batch operating system do not interact with the computer directly. Each user prepares its job on an off-line device like punch cards and submits it to the computer operator. To speed up the processing, jobs with similar needs are batched together and run as a group. The programmers exit their programs with the operator and the operator then sorts the programs with similar requirements into batches.

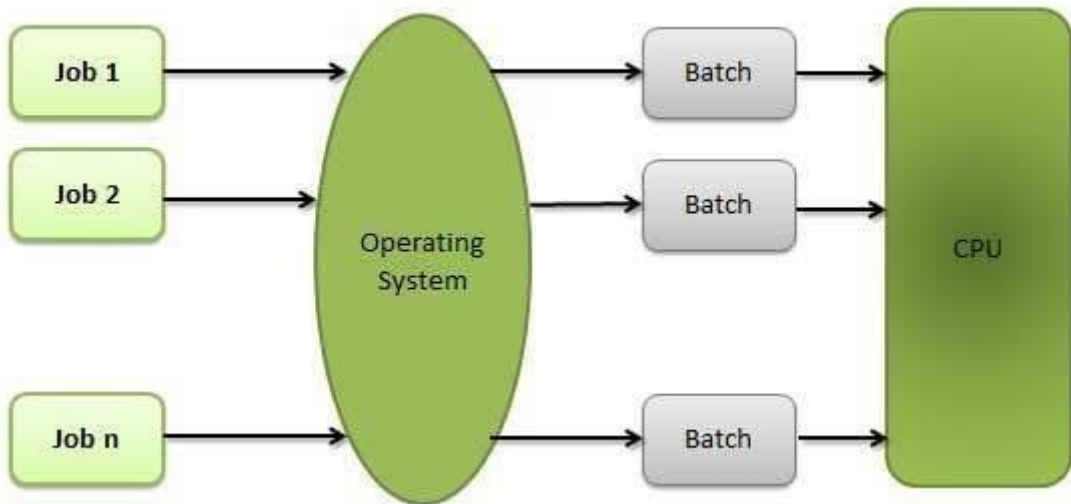
**The problems that occurs** with Batch Systems are as follows –

- There is a lack of interaction between the user and the job.
- CPU is being often idle, because the speed of the mechanical I/O devices is slower than the CPU.
- It is difficult to provide the desired priority.

## **Batch processing**

Batch processing is a technique in which an Operating System collects the programs and data together in a batch before processing starts. An operating system does the following activities related to batch processing .

- The OS defines a job which has predefined sequence of commands, programs and data as a single unit.
- The OS keeps a number a jobs in memory and executes them without any manual information.
- Jobs are processed in the order of submission, i.e., first come first served fashion.
- When a job completes its execution, its memory is released and the output for the job gets copied into an output spool for later printing or processing.



#### Advantages

- Batch processing takes much of the work of the operator to the computer.
- Increased performance as a new job get started as soon as the previous job is finished, without any manual intervention.

#### Disadvantages

- Difficult to debug program.
- A job could enter an infinite loop.
- Due to lack of protection scheme, one batch job can affect pending jobs.

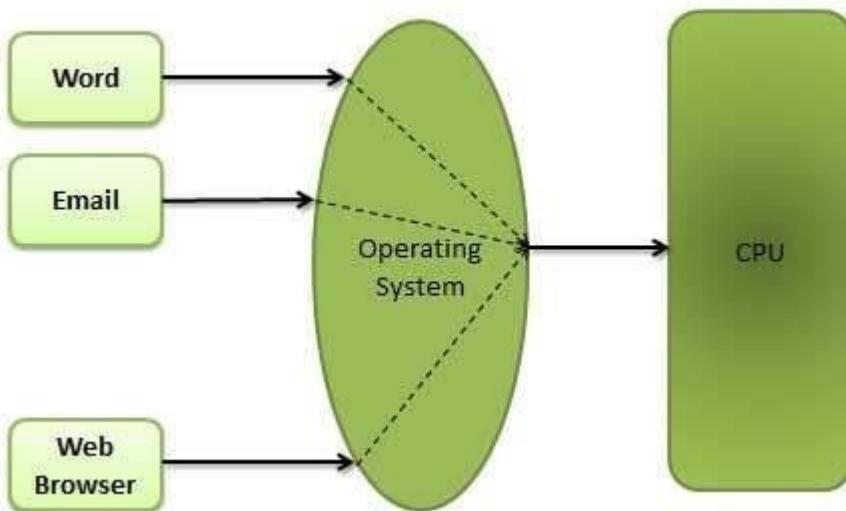
### Multitasking

Multitasking is when multiple jobs are executed by the CPU simultaneously by switching between them. Switches occur so frequently that the users may interact with each program while it is running. An OS does the following activities related to multitasking

- The user gives instructions to the operating system or to a program directly, and receives an immediate response.
- The OS handles multitasking in the way that it can handle multiple operations/executes multiple programs at a time.
- Multitasking Operating Systems are also known as Time-sharing systems.
- These Operating Systems were developed to provide interactive use of a computer system at a reasonable cost.
- A time-shared operating system uses the concept of CPU scheduling and multiprogramming to provide each user with a small portion of a time-shared CPU.



- Each user has at least one separate program in memory.



- A program that is loaded into memory and is executing is commonly referred to as a **process**.
- When a process executes, it typically executes for only a very short time before it either finishes or needs to perform I/O.
- Since interactive I/O typically runs at slower speeds, it may take a long time to complete. During this time, a CPU can be utilized by another process.
- The operating system allows the 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 CPU rapidly from one user/program to the next, each user is given the impression that he/she has his/her own CPU, whereas actually one CPU is being shared among many users.

## Multiprogramming

Sharing the processor, when two or more programs reside in memory at the same time, is referred to as **multiprogramming**. Multiprogramming assumes a single shared processor. Multiprogramming increases CPU utilization by organizing jobs so that the CPU always has one to execute.

The following figure shows the memory layout for a multiprogramming system.



An OS does the following activities related to multiprogramming.

- The operating system keeps several jobs in memory at a time.
- This set of jobs is a subset of the jobs kept in the job pool.
- The operating system picks and begins to execute one of the jobs in the memory.
- Multiprogramming operating systems monitor the state of all active programs and system resources using memory management programs to ensure that the CPU is never idle, unless there are no jobs to process.

### **Advantages**

- High and efficient CPU utilization.
- User feels that many programs are allotted CPU almost simultaneously.

### **Disadvantages**

- CPU scheduling is required.
- To accommodate many jobs in memory, memory management is required.

### **Interactivity**

Interactivity refers to the ability of users to interact with a computer system. An Operating system does the following activities related to interactivity –

- Provides the user an interface to interact with the system.
- Manages input devices to take inputs from the user. For example, keyboard.
- Manages output devices to show outputs to the user. For example, Monitor.

The response time of the OS needs to be short, since the user submits and waits for the result.

## Real Time System

Real-time systems are usually dedicated, embedded systems. An operating system does the following activities related to real-time system activity.

- In such systems, Operating Systems typically read from and react to sensor data.
- The Operating system must guarantee response to events within fixed periods of time to ensure correct performance.

**Real time operating systems (RTOS)** are used in environments where a large number of events, mostly external to the computer system, must be accepted and processed in a short time or within certain deadlines. such applications are industrial control, telephone switching equipment, flight control, and real time simulations.

The real time operating systems can be of 2 types –

### 1. **Hard Real Time operating system:**

These operating systems guarantee that critical tasks be completed within a range of time.

For example, a robot is hired to weld a car body, if robot welds too early or too late, the car cannot be sold, so it is a hard real time system that require to complete car welding by robot hardly on the time.

### 2. **Soft real time operating system:**

This operating systems provides some relaxation in time limit.

**For example** – Multimedia systems, digital audio system etc. Explicit, programmer- defined and controlled processes are encountered in real time systems. A separate process is changed with handling a single external event. The process is activated upon occurrence of the related event signaled by an interrupt.

Multitasking operation is accomplished by scheduling processes for execution independently of each other. Each process is assigned a certain level of priority that corresponds to the relative importance of the event that it services. The processor is allocated to the highest priority processes. This type of scheduling, called, priority based preemptive scheduling is used by real time systems.

## **Advantages:**

The advantages of real time operating systems are as follows-

### 1. **Maximum consumption –**

Maximum utilization of devices and system. Thus more output from all the resources.

## 2. Task Shifting –

Time assigned for shifting tasks in these systems are very less. For example in older systems it takes about 10 micro seconds. In shifting one task to another and in latest systems it takes 3 micro seconds.

## 3. Focus On Application –

Focus on running applications and less importance to applications which are in queue.

## 4. Real Time Operating System In Embedded System –

Since size of programs are small, RTOS can also be embedded systems like in transport and others.

## 5. Error Free –

These types of systems are error free.

# Distributed Environment

A distributed environment refers to multiple independent CPUs or processors in a computer system. An operating system does the following activities related to distributed environment –

- The OS distributes computation logics among several physical processors.
- The processors do not share memory or a clock. Instead, each processor has its own local memory.
- The OS manages the communications between the processors. They communicate with each other through various communication lines.

An operating system (OS) is basically a collection of software that manages computer hardware resources and provides common services for computer programs. Operating system is a crucial component of the system software in a computer system.

**Distributed Operating System** is one of the important type of operating system.

Multiple central processors are used by Distributed systems to serve multiple real-time applications and multiple users. Accordingly, Data processing jobs are distributed among the processors.

Processors communicate with each other through various communication lines (like high-speed buses or telephone lines). These are known as **loosely coupled systems** or distributed systems. Processors in this system may vary in size and function. They are referred as sites, nodes, computers, and so on.

## Advantages

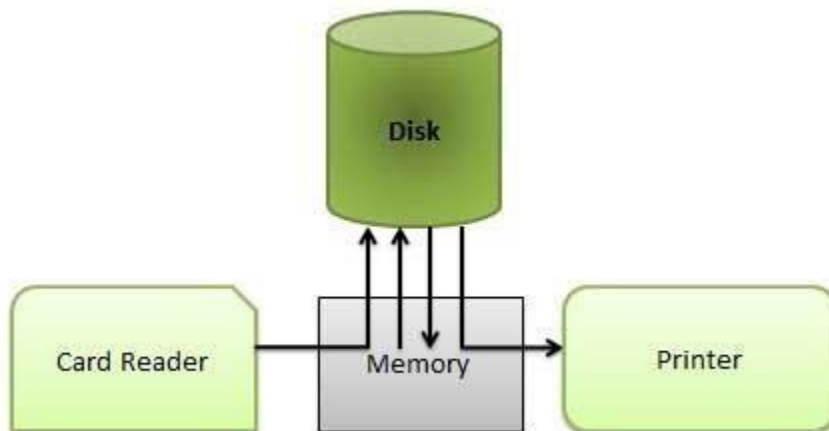
- With resource sharing facility, a user at one site may be able to use the resources available at another.
- Speedup the exchange of data with one another via electronic mail.
- Failure of one site in a distributed system doesn't affect the others, the remaining sites can potentially continue operating.
- Better service to the customers.
- Reduction of the load on the host computer.
- Reduction of delays in data processing.

## Spooling

Spooling is an acronym for simultaneous peripheral operations on line. Spooling refers to putting data of various I/O jobs in a buffer. This buffer is a special area in memory or hard disk which is accessible to I/O devices.

An operating system does the following activities related to distributed environment –

- Handles I/O device data spooling as devices have different data access rates.
- Maintains the spooling buffer which provides a waiting station where data can rest while the slower device catches up.
- Maintains parallel computation because of spooling process as a computer can perform I/O in parallel fashion. It becomes possible to have the computer read data from a tape, write data to disk and to write out to a tape printer while it is doing its computing task.



## Advantages

- The spooling operation uses a disk as a very large buffer.
- Spooling is capable of overlapping I/O operation for one job with processor operations for another job.



