OPERATING SYSTEM – [OS] UNIT-I

Introduction:

Operating System is software that works as an interface between a user and the computer hardware. The primary objective of an operating system is to make computer system convenient to use and to utilize computer hardware in an efficient manner. The operating system performs the basic tasks such as receiving input from the keyboard, processing instructions and sending output to the screen.

The Software is the Non-Touchable Parts of the Computer, and Software's are those which are used for Performing an Operation So that Software's are just used for Making an Application but hardware's are those which are used for Performing an Operation. Operating system is software that is required in order to run application programs and utilities. It works as a bridge to perform better interaction between application programs and hardware of the computer. Various types of operating systems' are UNIX, MS-DOS, MS-Windows - 98/XP/Vista, WindowsNT/2000, OS/2 and Mac OS.

Operating system manages overall activities of a computer and the input/output devices attached to the computer. It is the first software you see when you turn on the computer, and the last software you see when the computer is turned off. It is the software that enables all the programs you use. At the simplest level, an operating system does two things:

The first, it manages the hardware and software resources of the computer system. These resources include the processor, memory, disk space, etc. The second, it provides a stable, consistent way for applications to deal with the hardware without having-to know all the details of the hardware.

The first task is very important i.e. managing the hardware and software resources, as various processes compete to each other for getting the CPU time and memory space to complete the task. In this regard; the operating system acts as a manager to allocate the available resources to 'satisfy the requirements of each process.

The second task i.e. providing a consistent application interface is especially important. A consistent application program interface (API) allows a user (or S/W developer) to write an application program on any computer and to run this program on another computer, even if the hardware configuration is different like as amount of memory, type of CPU or storage disk. It shields the user of the machine from the low-level details of the machine's operation and provides frequently needed facilities. When you turn on the computer, the operating system program is loaded into the main memory. This program is called the kernel. Once initialized, the system program is prepared to run the user programs and permits them to use the hardware efficiently. Windows 98/XP is an excellent example that supports different types of hardware configurations from thousands of vendors and accommodates thousands of different I/O devices like printers, disk drives, scanners and cameras.

Operating systems may be classified based on if multiple tasks can be performed simultaneously, and if the system can be used by multiple users. It can be termed as single-user or multiuser OS, and single-tasking or multi-tasking OS.A multi-user system must be multi-tasking. MS-DOS and Windows 3x are examples of single user operating system. Whereas UNIX is an example of multi-user and multitasking operating system.

Classification of Operating systems:

- **Multi-user** : Allows two or more users to run programs at the same time. Some operating systems permit hundreds or even thousands of concurrent users.
- **Multiprocessing** : Supports running a program on more than one CPU.
- **Multitasking** : Allows more than one program to run concurrently.
- **Multithreading** : Allows different parts of a single program to run concurrently.
- **Real time** : Responds to input instantly. General-purpose operating systems, such as DOS and UNIX, are not real-time.

For Example if we want to Perform Some Paintings on the Screen, then we must use the Application Software as Paint and Hardware as a Mouse for Drawing an Object. But how the System knows what to do when Mouse Moves on the Screen and When the Mouse Draws a Line on the System so that Operating System is Necessary which Interact between or which Communicates with the Hardware and the Software.

Characteristics or Functions of OS:

For Better understanding you can see the Working of the Operating System. So we can say that the Operating System has the Following **Characteristics**:

- It boots the computer.
- Operating System is a Collection of Programs those are Responsible for the Execution of other Programs.
- Operating System is that which Responsible is for Controlling all the Input and Output Devices those are connected to the System.
- Operating System is that which Responsible is for Running all the Application Software's.
- Operating System is that which Provides Scheduling to the Various Processes Means Allocates the Memory to various Process those Wants to Execute.
- Operating System is that which provides the Communication between the user and the System.
- Operating System is Stored into the BIOS Means in the Basic Input and Output System means when a user Starts his System then this will Read all the instructions those are Necessary for Executing the System Means for Running the Operating System, Operating System Must be

Loaded into the Computer For this, this will use the Floppy or Hard Disks Which Stores the Operating System.

- It provides file management which refers to the way that the operating system manipulates, stores, retrieves and saves data.
- Error Handling is done by the operating system. It takes preventive measures whenever required to avoid errors.

Most Popular Desktop Operating Systems:

The three most popular types of operating systems for personal and business computing include Linux, Windows and Mac.

Windows - Microsoft Windows is a family of operating systems for personal and business computers. Windows dominates the personal computer world, offering a graphical user interface (GUI), virtual memory management, multitasking, and support for many peripheral devices.

Mac - Mac OS is the official name of the Apple Macintosh operating system. Mac OS features a graphical user interface (GUI) that utilizes windows, icons, and all applications that run on a Macintosh computer have a similar user interface.

Linux - Linux is a freely distributed open source operating system that runs on a number of hardware platforms. The Linux kernel was developed mainly by Linus Torvalds and it is based on Unix.

In the same way that a desktop OS controls your desktop or laptop computer, a mobile operating system is the software platform on top of which other programs can run on mobile devices, however, these systems are designed specifically to run on mobile devices such as mobile phones, smartphones, PDAs, tablet computers and other handhelds. The mobile OS is responsible for determining the functions and features available on your device, such as thumb wheel, keyboards, WAP, synchronization with applications, email, text messaging and more. The mobile OS will also determine which third-party applications (mobile apps) can be used on your device.

What is an Operating System?

A program that acts as an intermediary between a user of a computer and the computer hardware Operating system goals:

- Execute user programs and make solving user problems easier
- Make the computer system convenient to use
- Use the computer hardware in an efficient manner

Operating System Definition

- OS is a resource allocator
- Manages all resources
- Decides between conflicting requests for efficient and fair resource use

- OS is a control program
- Controls execution of programs to prevent errors and improper use of the computer
- No universally accepted definition
- Everything a vendor ships when you order an operating system" is good approximation. But varies wildly

===

Definition:

The operating system is the "one program running at all times on the computer"- called the **kernel**. (Along with the kernel, there are two other types of programs:

- **System programs-** which are associated with the operating system but are not necessarily part of the kernel, and
- **Application programs-** which include all programs not associated with the operation of the system.)

===

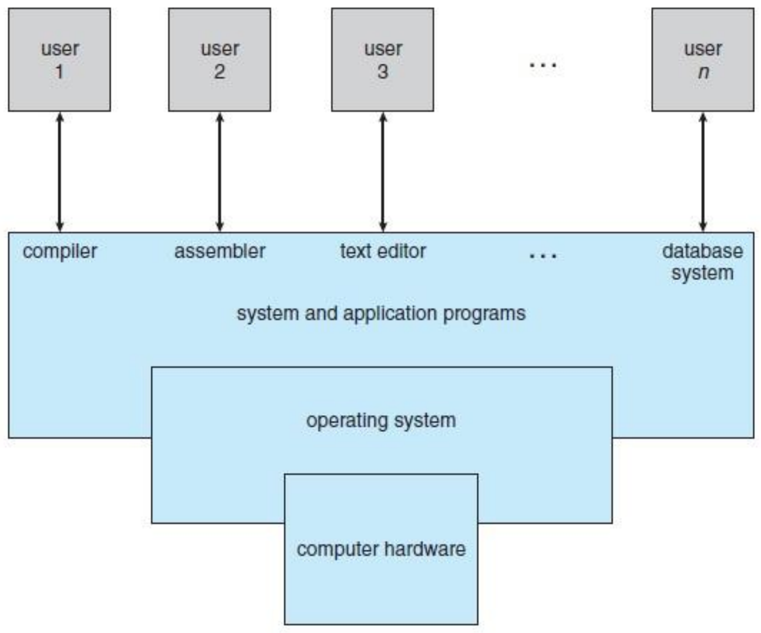
Why should I study Operating Systems?

- Need to understand interaction between the hardware and software
- Need to understand basic principles in the design of computer systems
- Efficient resource management, security, flexibility
- Because it enables you to do things that are difficult/impossible otherwise.

OS challenges

•Reliability
-Does the system do what it was designed to do?
•Availability
-What portion of the time is the system working?
-Mean Time To Failure (MTTF), Mean Time to Repair
•Security
-Can the system be compromised by an attacker?
•Privacy
-Data is accessible only to authorized users
•Performance
•Latency/response time -How long does an operation take to complete?
•Throughput -How many operations can be done per unit of time?
•Overhead -How much extra work is done by the OS?
•Fairness -How equal is the performance received by different users?
•Predictability -How consistent is the performance over time?

WHAT OPERATING SYSTEMS DO



Abstract view of the components of a computer system.

A computer system can be divided roughly into four components: the ***hardware,*** the ***operating system,*** the ***application programs,*** and the ***users.***

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.

Two Views of Operating System

1. User's View
2. System View

User View:

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, of course, 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 other cases, users sit at **workstations** connected to networks of other workstations and **servers**.

Recently, many varieties of mobile computers, such as smart phones and tablets, have come into fashion. 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.

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**.

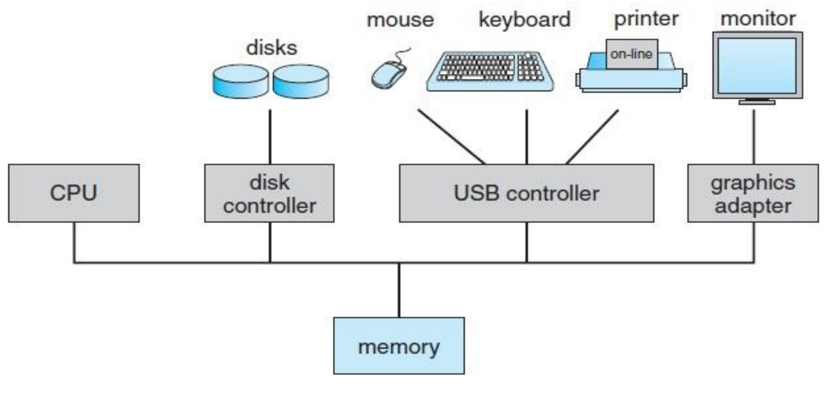
A computer system has many resources that may be required to solve a problem: CPU time, memory space, file-storage space, I/O devices, and so on. The operating system acts as the manager of these resources.

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.

COMPUTER SYSTEM ORGANIZATION

Before we can explore the details of how computer systems operate, we need general knowledge of the structure of a computer system. In this section, we look at several parts of this structure. The section is mostly concerned with computer-system organization, so you can skim or skip it if you already understand the concepts.

Computer-System Operation:



A modern computer system.

A modern general-purpose computer system consists of one or more CPUs and a number of device controllers connected through a common bus that provides access to shared memory (Figure). Each device controller is in charge of a specific type of device (for example, disk drives, audio devices, or video displays). The CPU and the device controllers can execute in parallel, competing for memory cycles. To ensure orderly access to the shared memory, a memory controller synchronizes access to the memory.

For a computer to start running—for instance, when it is powered up or rebooted—it needs to have an initial program to run.

This initial program, or **bootstrap program**, tends to be simple.

Typically, it is stored within the computer hardware in read-only memory (**ROM**) or electrically erasable programmable read-only memory (**EEPROM**), known by the general term **firmware**.

It initializes all aspects of the system, from CPU registers to device controllers to memory contents.

The bootstrap program must know how to load the operating system and how to start executing that system.

To accomplish this goal, the bootstrap program must locate the operating-system kernel and load it into memory.

Once the kernel is loaded and executing, it can start providing services to the system and its users. Some services are provided outside of the kernel, by system programs that are loaded into memory at boot time to become **system processes**, or **system daemons** that run the entire time the kernel is running. On UNIX, the first system process is "init," and it starts many other daemons. Once this phase is complete, the system is fully booted, and the system waits for some event to occur.

The occurrence of an event is usually signaled by an **interrupt** from either the hardware or the software. Hardware may trigger an interrupt at any time by sending a signal to the CPU, usually by way of the system bus. Software may trigger an interrupt by executing a special operation called a **system call** (also called a **monitor call**).

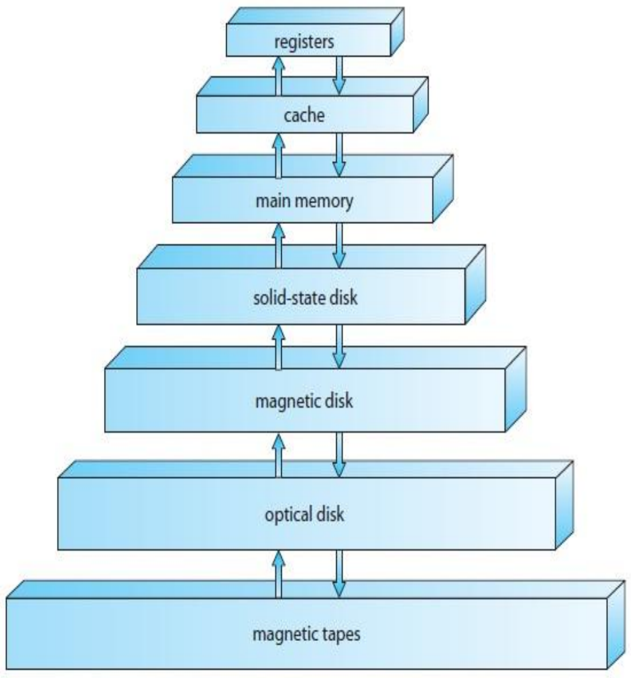
Storage Structure:

The CPU can load instructions only from memory, so any programs to run must be stored there. General-purpose computers run most of their programs from rewritable memory, called main memory (also called **random-access memory**, or **RAM**). Main memory commonly is implemented in a semiconductor technology called **dynamic random-access memory (DRAM)**. Ideally, we want the programs and data to reside in main memory permanently.

This arrangement usually is not possible for the following two reasons:

1. Main memory is usually too small to store all needed programs and data permanently.

2. Main memory is a **volatile** storage device that loses its contents when power is turned off or otherwise lost.



Storage-device hierarchy

Thus, most computer systems provide **secondary storage** as an extension of main memory. The main requirement for secondary storage is that it be able to hold large quantities of data permanently. The most common secondary-storage device is a **magnetic disk**, which provides storage for both programs and data.

The wide variety of storage systems can be organized in a hierarchy (Figure) according to speed and cost. The higher levels are expensive, but they are fast. As we move down the hierarchy, the cost per bit generally decreases, whereas the access time generally increases.

Storage Definition and notation

- The basic unit of computer storage is the **bit**. A bit can contain one of two values, 0 and 1.
- All other storage in a computer is based on collections of bits.
- Given enough bits, it is amazing how many things a computer can represent: numbers, letters, images, movies, sounds, documents, and programs, to name a few.
- A **byte** is 8 bits, and on most computers it is the smallest convenient chunk of storage.
- For example, most computers don't have an instruction to move a bit but do have one to move a byte. A less common term is **word**, which is a given computer architecture's native unit of data. A word is made up of one or more bytes.
- For example, a computer that has 64-bit registers and 64-bit memory addressing typically has 64-bit (8-byte) words.
- A computer executes many operations in its native word size rather than a byte at a time.
- Computer storage, along with most computer throughput, is generally measured and manipulated in bytes and collections of bytes.

a **kilobyte**, or **KB**, is 1,024 bytes;
a **megabyte**, or **MB**, is 1,0242 bytes;
a **gigabyte**, or **GB**, is 1,0243 bytes;
a **terabyte**, or **TB**, is 1,0244 bytes; and
a **petabyte**, or **PB**, is 1,0245 bytes.

- Computer manufacturers often round off these numbers and say that a megabyte is 1 million bytes and a gigabyte is 1 billion bytes. Networking measurements are an exception to this general rule; they are given in bits (because networks move data a bit at a time).

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.

A general-purpose computer system consists of CPUs and multiple device controllers that are connected through a common bus. Each device controller is in charge of a specific type of device. Depending on the controller, more than one device may be attached. For instance, seven or more devices can be attached to the **small computer-systems interface (SCSI)** controller.

A device controller maintains some local buffer storage and a set of special-purpose registers. The device controller is responsible for moving the data between the peripheral devices that it controls and its local buffer storage. Typically, operating systems have a **device driver** for each device controller. This device driver understands the device controller and provides the rest of the operating system with a uniform interface to the device.

This form of interrupt-driven I/O is fine for moving small amounts of data but can produce high overhead when used for bulk data movement such as disk I/O. To solve this problem, **direct memory access (DMA)** is used.

<u>COMPUTER-SYSTEM ARCHITECTURE</u>

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.

Single-Processor Systems

Until recently, most computer systems used a single processor. On a single processor system, there is one main CPU capable of executing a general-purpose instruction set, including instructions from user processes.

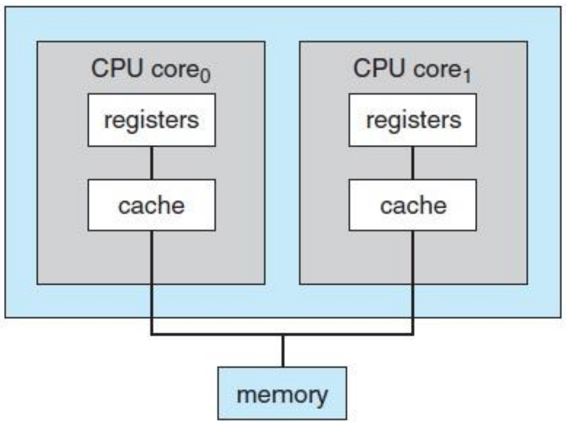
Almost all single processor systems have other special-purpose processors as well. They may come in the form of device-specific processors, such as disk, keyboard, and graphics controllers; or, on mainframes, they may come in the form of more general-purpose processors, such as I/O processors that move 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.

Multiprocessor Systems



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

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 appeared 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.

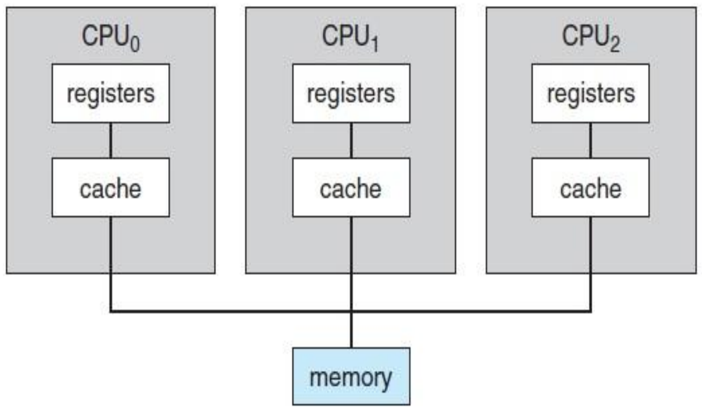
Multiprocessor systems have three main advantages:

1. 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.

2. Economy of scale. Multiprocessor systems can cost less than equivalent multiple single-processor systems, because they can share peripherals, mass storage, and power supplies. If several programs operate on the same set of data, it is cheaper to store those data on one disk and to have all the processors share them than to have many computers with local disks and many copies of the data.

3. 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.



Symmetric multiprocessing architecture.

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.

Clustered Systems:

- Like parallel systems, clustered systems gather together multiple CPUs to accomplish computational work.
- Clustered systems differ from parallel systems, however, in that they are composed of two or more individual systems coupled together.
- The definition of the term clustered is **not concrete;** the general accepted definition is that clustered computers share storage and is closely linked via LAN networking.
- Clustering is usually performed to provide **high availability**.
- A layer of cluster software runs on the cluster nodes. Each node can monitor one or more of the others. If the monitored machine fails, the monitoring machine can take ownership of its storage, and restart the application(s) that were running on the failed machine. The failed machine can remain down, but the users and clients of the application would only see a brief interruption of service.

- **Asymmetric Clustering -** In this, 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.
- **Symmetric Clustering -** In this, two or more hosts are running applications, and they are monitoring each other. This mode is obviously more efficient, as it uses all of the available hardware.
- **Parallel Clustering -** Parallel clusters allow multiple hosts to access the same data on the shared storage. Because most operating systems lack support for this simultaneous data access by multiple hosts, parallel clusters are usually accomplished by special versions of software and special releases of applications.

Clustered technology is rapidly changing. Clustered system use and features should expand greatly as **Storage Area Networks (SANs)**. SANs allow easy attachment of multiple hosts to multiple storage units. Current clusters are usually limited to two or four hosts due to the complexity of connecting the hosts to shared storage.

OPERATING-SYSTEM STRUCTURE

Now that we have discussed basic computer-system organization and architecture, we are ready to talk about operating systems.

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. There are, however, many commonalities, which we consider in this section.

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.

This idea is common in other life situations. A lawyer does not work for only one client at a time, for example. While one case is waiting to go to trial or have papers typed, the lawyer can work on another case. If he has enough clients, the lawyer will never be idle for lack of work. (Idle lawyers tend to become politicians, so there is a certain social value in keeping lawyers busy.) 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.

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.

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

In 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.

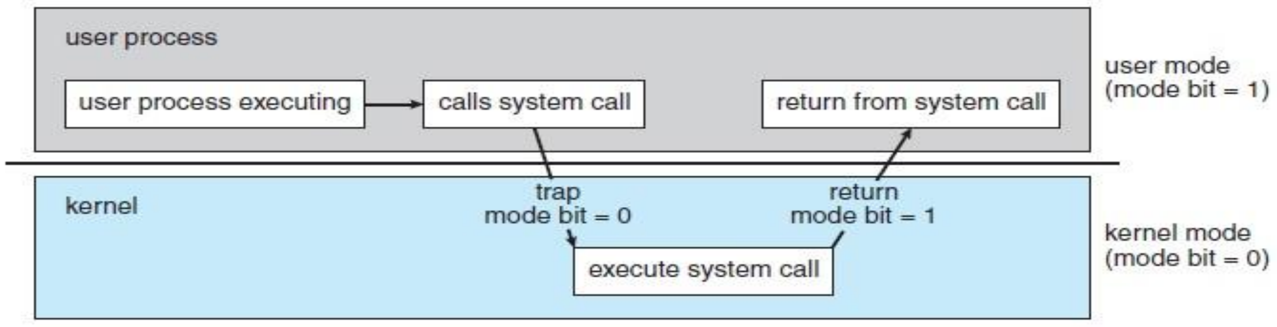
A **trap** (or an **exception**) is a software-generated interrupt caused either by an error (for example, division by zero or invalid memory access) or by a specific request from a user program that an operating-system service be performed. The interrupt-driven nature of an operating system defines 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.



Transition from user to kernel mode.

At the very least, we need two separate ***modes*** of operation:

1. **User mode** and
2. **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 above Figure. 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.

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 MANAGEMENT TASKS

1. **Processor management** which involves putting the tasks into order and pairing them into manageable size before they go to the CPU.
2. **Memory management** which coordinates data to and from RAM (random-access memory) and determines the necessity for virtual memory.
3. **Device management** which provides interface between connected devices.
4. **Storage management** which directs permanent data storage.
5. **Production and Security.**
6. **Application** which allows standard communication between software and your computer.
7. **User interface** which allows you to communicate with your computer.

Process Management:

A program does nothing unless its instructions are executed by a CPU.

A program in execution, as mentioned, is a process.

A time-shared user program such as a compiler is a process.

A word-processing program being run by an individual user on a PC is a process.

A system task, such as sending output to a printer, can also be a process .

A process needs certain resources-including CPU time, memory, files, and I/O devices-to accomplish its task

A process is the unit of work in a system. A system consists of a collection of processes, some of which are operating-system processes (those that execute system code) and the rest of which are user processes (those that execute user code). All these processes can potentially execute concurrently—by multiplexing on a single CPU, for example.

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

- Scheduling processes and threads on the CPUs
- Creating and deleting both user and system processes
- Suspending and resuming processes
- Providing mechanisms for process synchronization
- Providing mechanisms for process communication

Memory Management:

The main memory is generally the only large storage device that the CPU is able to address and access directly. For example, for the CPU to process data from disk, those data must first be transferred to main memory by CPU-generated I/O calls. In the same way, instructions must be in memory for the

CPU to execute them. For a program to be executed, it must be mapped to absolute addresses and loaded into memory. As the program executes, it accesses program instructions and data from memory by generating these absolute addresses. Eventually, the program terminates, its memory space is declared available, and the next program can be loaded and executed.

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

- Keeping track of which parts of memory are currently being used and who is using them
- Deciding which processes (or parts of processes) and data to move into and out of memory
- Allocating and deallocating memory space as needed

Storage Management:

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

File-System Management:

File management is one of the most visible components of an operating system. Computers can store information on several different types of physical media. Magnetic disk, optical disk, and magnetic tape are the most common. Each of these media has its own characteristics and physical organization. Each medium is controlled by a device, such as a disk drive or tape drive, that also has its own unique characteristics. These properties include access speed, capacity, data-transfer rate, and access method (sequential or random).

A file is a collection of related information defined by its creator. Commonly, files represent programs (both source and object forms) and data. Data files may be numeric, alphabetic, alphanumeric, or binary. Files may be free-form (for example, text files), or they may be formatted rigidly (for example, fixed fields).

Clearly, the concept of a file is an extremely general one. The operating system implements the abstract concept of a file by managing mass-storage media, such as tapes and disks, and the devices that control them. In addition, files are normally organized into directories to make them easier to use. Finally, when multiple users have access to files, it may be desirable to control which user may access a file and how that user may access it (for example, read, write, append).

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

- Creating and deleting files
- Creating and deleting directories to organize files
- Supporting primitives for manipulating files and directories
- Mapping files onto secondary storage
- Backing up files on stable (nonvolatile) storage media

Mass-Storage Management:

The main memory is too small to accommodate all data and programs, and because the data that it holds are lost when power is lost, the computer system must provide secondary storage to back up main memory. Most modern computer systems use disks as the principal on-line storage medium for both programs and data. Most programs—including compilers, assemblers, word processors, editors, and formatters—are stored on a disk until loaded into memory. They then use the disk as both the source and destination of their processing. Hence, the proper management of disk storage is of central importance to a computer system.

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

- Free-space management
- Storage allocation
- Disk scheduling

Because secondary storage is used frequently, it must be used efficiently. The entire speed of operation of a computer may hinge on the speeds of the disk subsystem and the algorithms that manipulate that subsystem.

Caching:

Caching is an important principle of computer systems. Here’s how it works. Information is normally kept in some storage system (such as main memory). As it is used, it is copied into a faster storage system—the cache—on a temporary basis. When we need a particular piece of information, we first check whether it is in the cache. If it is, we use the information directly from the cache. If it is not, we use the information from the source, putting a copy in the cache under the assumption that we will need it again soon In a hierarchical storage structure, the same data may appear in different levels of the storage system. For example, suppose that an integer A that is to be incremented by 1 is located in file B, and file B resides on magnetic disk.

The increment operation proceeds by first issuing an I/O operation to copy the disk block on which A resides to main memory. This operation is followed by copying A to the cache and to an internal register. Thus, the copy of A appears in several places: on the magnetic disk, in main memory, in the cache, and in an internal register. Once the increment takes place in the internal register, the value of A differs in the various storage systems. The value of A becomes the same only after the new value of A is written from the internal register back to the magnetic disk.

I/O Systems:

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

The I/O subsystem consists of several components:

• A memory-management component that includes buffering, caching, and spooling

• A general device-driver interface

• Drivers for specific hardware devices

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

Protection and Security:

Protection – any mechanism for controlling access of processes or users to resources defined by the OS

Security – defense of the system against internal and external attacks

- Huge range, including denial-of-service, worms, viruses, identity theft, theft of service
- Systems generally first distinguish among users, to determine who can do what
- User identities (**user IDs**, security IDs) include name and associated number, one per user
- User ID then associated with all files, processes of that user to determine access control
- Group identifier (**group ID**) allows set of users to be defined and controls managed, then also associated with each process, file
- **Privilege escalation** allows user to change to effective ID with more rights

COMPUTING ENVIRONMENTS

In computers, the term **environment** when unqualified usually refers to the combination of hardware and software in a **computer**. In this usage, the term platform is a synonym.

Traditional Computing:

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 are stretching the boundaries of traditional computing. Companies establish portals, which provide web accessibility to their internal servers.

Network computers are essentially terminals that understand web-based computing. Handheld computers can synchronize with PCs to allow very portable use of company information. Handheld PDAs can also connect to wireless networks to use the company's web portal (as well as the myriad other web resources).

At home, most users 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, 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. Some homes even have firewalls to protect their networks from security breaches. Those firewalls cost thousands of rupees a few years ago and did not even exist a decade ago. In the latter half of the previous century, computing resources were scarce.

Mobile Computing:

Mobile computing refers to computing on handheld smartphones 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.

Today, mobile systems are used not only for e-mail and web browsing but also for playing music and video, reading digital books, taking photos, and recording high-definition video. Accordingly, tremendous growth continues in the wide range of applications that run on such devices.

Many developers are now designing applications that take advantage of the unique features of mobile devices, such as

global positioning system (GPS) chips,

accelerometers, and

gyroscopes.

The memory capacity and processing speed of mobile devices, however, are more limited than those of PCs. Whereas a smartphone or tablet may have 64 GB in storage, it is not uncommon to find 1

TB in storage on a desktop computer. Similarly, becauseare smaller, are slower, and offer fewer processing cores than processors found on traditional desktop and laptop computers.

Two operating systems currently dominate mobile computing:

Apple iOS and Google Android. iOS was designed to run on Apple iPhone and iPad mobile devices. Android powers smartphones and tablet computers available from many manufacturers.

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.

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.

A **network**, in the simplest terms, is a communication path between two or more systems. Distributed systems depend on networking for their functionality.

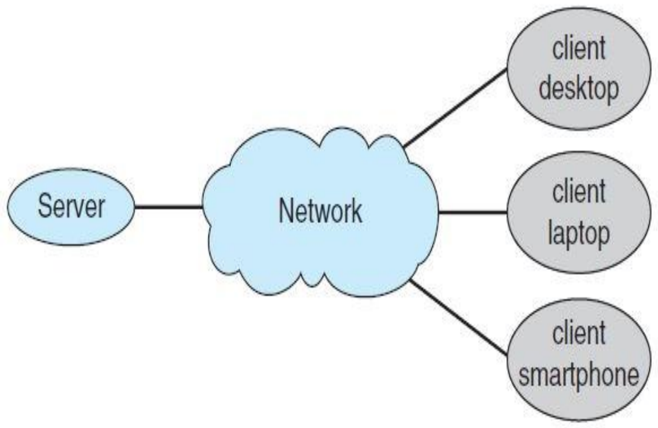
Networks vary by the protocols used, the distances between nodes, and the transport media. **TCP/IP** is the most common network protocol, and it provides the fundamental architecture of the Internet.

Networks are characterized based on the distances between their nodes.

- A **local-area network (LAN)** connects computers within a room, a building, or a campus.
- A **wide-area network (WAN)** usually links buildings, cities, or countries. A global company may have a WAN to connect its offices worldwide, for example. These networks may run one protocol or several protocols.
- The continuing advent of new technologies brings about new forms of networks. For example, a **metropolitan-area network (MAN)** could link buildings within a city.
- BlueTooth device use wireless technology to communicate over a distance of several feet, in essence creating a **personal-area network (PAN)** between a phone and a headset or a smartphone and a desktop computer.

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.



General structure of a client–server system.

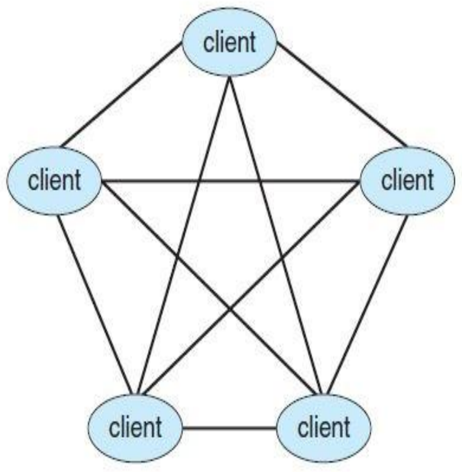
Correspondingly, user-interface functionality once handled directly by centralized systems is increasingly being handled by PCs, quite often through a web interface. As a result, many of today's systems act as **server systems** to satisfy requests generated by **client systems**.

This form of specialized distributed system, called a **client–server** system, has the general structure depicted in Figure

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:



Peer-to-peer system with no centralized service.

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. To participate in a peer-to-peer system, a node must first join the network of peers. Once a node has joined the network, it can begin providing services to—and requesting services from—other nodes in the network. Determining what services are available is accomplished in one of two general ways:

• When a node joins a network, it registers its service with a centralized lookup service on the network. Any node desiring a specific service first contacts this centralized lookup service to determine which node provides the service. The remainder of the communication takes place between the client and the service provider.

• A peer acting as a client must first discover what node provides a desired service by broadcasting a request for the service to all other nodes in the network. The node (or nodes) providing that service responds to the peer making the request. To support this approach, a discovery protocol must be provided that allows peers to discover services provided by other peers in the network.

Peer-to-peer networks gained widespread popularity in the late 1990s with several file-sharing services, such as Napster and Gnutella that enable peers to exchange files with one another.

Web-Based Computing

The Web has become ubiquitous, leading to more access by a wider variety of devices than was dreamt of a few years ago. PCs are still the most prevalent access devices, with workstations, handheld PDAs, and even cell phones also providing access.

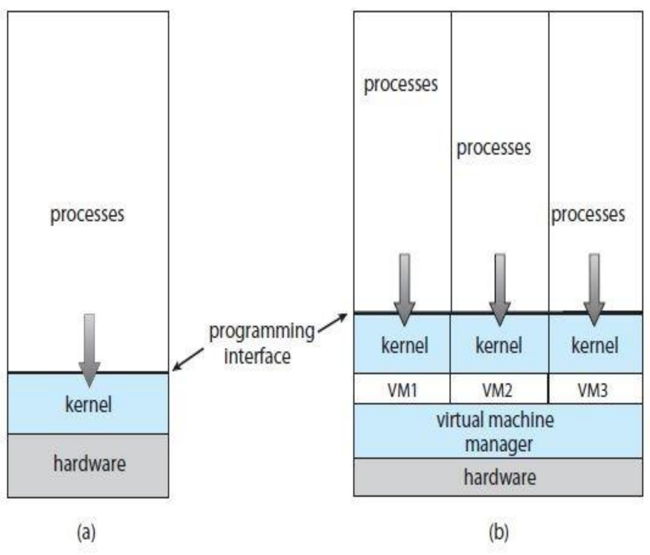
Web computing has increased the emphasis on networking. Devices that were not previously networked now include wired or wireless access. Devices that were networked now have faster network connectivity, provided by either improved networking technology, optimized network implementation code, or both.

The implementation of web-based computing has given rise to new categories of devices, such as load balancers, which distribute network connections among a pool of similar servers.

Operating systems like Windows 95, which acted as web clients, have evolved into Linux and Windows XP, which can act as web servers as well as clients. Generally, the Web has increased the complexity of devices, because their users require them to be web-enabled.

Virtualization:

Virtualization is a technology that allows operating systems to run as applications within other operating systems. But the virtualization industry is vast and growing, which is a testament to its utility and importance. Broadly speaking,



VMware.

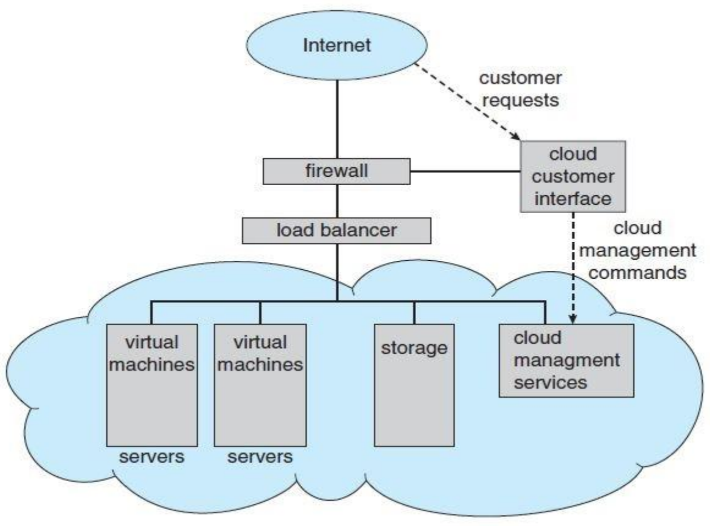
It is also called OS-level virtualization is a type of virtualization technology which work on OS layer. Here the kernel of an OS allows more than one isolated user-space instances to exist. Such instances are called containers/software containers or virtualization engines. In other words, OS kernel will run a single operating system & provide that operating system's functionality to replicate on each of the isolated partitions.

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. That same concept can be extended to allow an entire operating system written for one platform to runon another. Emulation comes at a heavy price, however. Every machine-level instruction that runs natively on the source system must be translated to the equivalent function on the target system, frequently resulting in several target instructions. If the source and target CPUs have similar performance levels, the emulated code can run much slower than the native code.

Cloud Computing:



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.

For example, the Amazon Elastic Compute Cloud **(EC2)** facility has thousands of servers, millions of virtual machines, and peta bytes of storage available for use by anyone on the Internet. Users pay per month based on how much of those resources they use.

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)

These cloud-computing types are not discrete, as a cloud computing environment may provide a combination of several types. For example, an organization may provide both SaaS and IaaS as a publicly available service.

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.

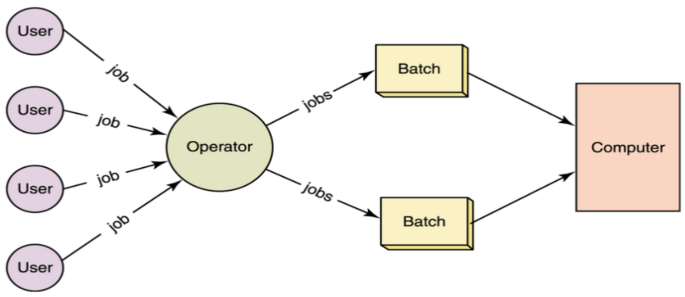
Embedded systems almost always run **real-time operating systems**. A real-time system is used when rigid time requirements have been placed on the operation of a processor or the flow of data; thus, it is often used as a control device in a dedicated application. Sensors bring data to the computer.

The computer must analyze the data and possibly adjust controls to modify the sensor inputs. Systems that control scientific experiments, medical imaging systems, industrial control systems, and certain display systems are real-time systems. Some automobile-engine fuel-injection systems, home-appliance controllers, and weapon systems are also real-time systems.

HISTORY OF OPERATING SYSTEM

- Earliest computers had no Operating System
    - Applications loaded manually
    - Users were experts on the hardware
- First *System Software* was libraries of code to manage devices.
- This grew to batch processing systems, where some focused on application programming and some on systems programming.

Batch Processing:



A typical computer in the 1960s and 70s was a large machine. Its processing was managed by a human operator. The operator would organize various jobs from multiple users into batches. Human operators would organize jobs into batches

Time Sharing:

A timesharing system allows multiple users to interact with a computer at the same time.

Multiprogramming allowed multiple processes to be active at once, which gave rise to the ability for programmers to interact with the computer system directly, while still sharing its resources. In a timesharing system, each user has his or her own virtual machine, in which all system resources are (in effect) available for use.

Fun Pictures

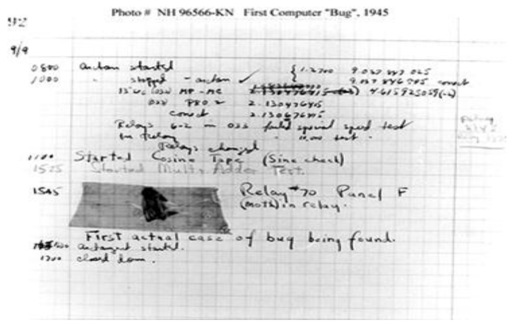


Photo # NH 96566-KN First Computer "Bug", 1945

92

9/9

0800 anctan started {1.2700 9.037 847 025
1000 " stopped - anctan ✓ 9.037 846 995 const
13" UC (033) MP - MC ~~2.130476415~~ 4.615925059(-2)
(033) PRO 2 2.130476415
const 2.130676415
Relays 6-2 in 033 failed special speed test
in relay " 11.000 test.
Relays changed
1100 Started Cosine Tape (Sine check)
1525 Started Multi Adder Test.
1545 Relay #70 Panel F
(moth) in relay.
First actual case of bug being found.
1630 antangent started.
1700 closed down.

first computer *bug*, a moth

The IBM 650 Magnetic Drum Data Processing System Machine

Cray I supercomputer, introduced in 1976

Current Operating System Research Topics

Symmetric multiprocessing

Allows for several CPUs to process multiple jobs at the same time. CPUs are independent of one another, but each has access to the operating system.

Asymmetric multiprocessing

Some operating systems functions are assigned to subordinate processors, which take their instructions from the main CPU.

Distributed processing

Processors are placed at remote locations and are connected to each other via telecom devices. Different from symmetric multiprocessing systems as they do not share memory. Computations can be dispersed among several processors.

EVOLUTION OF OPERATING SYSTEMS

The evolution of operating systems is directly dependent to the development of computer systems and how users use them. Here is a quick tour of computing systems through the past fifty years in the timeline.

Early Evolution

- 1945: ENIAC, Moore School of Engineering, University of Pennsylvania.
- 1949: EDSAC and EDVAC
- 1949 BINAC - a successor to the ENIAC
- 1951: UNIVAC by Remington
- 1952: IBM 701
- 1956: The interrupt
- 1954-1957: FORTRAN was developed

Operating Systems by the late 1950s

By the late 1950s Operating systems were well improved and started supporting following usages :

- It was able to Single stream batch processing
- It could use Common, standardized, input/output routines for device access
- Program transition capabilities to reduce the overhead of starting a new job was added
- Error recovery to clean up after a job terminated abnormally was added.
- Job control languages that allowed users to specify the job definition and resource requirements were made possible.

Operating Systems In 1960s

- 1961: The dawn of minicomputers
- 1962 Compatible Time-Sharing System (CTSS) from MIT
- 1963 Burroughs Master Control Program (MCP) for the B5000 system
- 1964: IBM System/360
- 1960s: Disks become mainstream
- 1966: Minicomputers get cheaper, more powerful, and really useful
- 1967-1968: The mouse
- 1964 and onward: Multics
- 1969: The UNIX Time-Sharing System from Bell Telephone Laboratories

Supported OS Features by 1970s

- Multi User and Multi tasking was introduced.
- Dynamic address translation hardware and Virtual machines came into picture.
- Modular architectures came into existence.
- Personal, interactive systems came into existence.

Accomplishments after 1970

- 1971: Intel announces the microprocessor
- 1972: IBM comes out with VM: the Virtual Machine Operating System
- 1973: UNIX 4th Edition is published
- 1973: Ethernet

- 1974 The Personal Computer Age begins
- 1974: Gates and Allen wrote BASIC for the Altair
- 1976: Apple II
- August 12, 1981: IBM introduces the IBM PC
- 1983 Microsoft begins work on MS-Windows
- 1984 Apple Macintosh comes out
- 1990 Microsoft Windows 3.0 comes out
- 1991 GNU/Linux
- 1992 The first Windows virus comes out
- 1993 Windows NT
- 2007: iOS
- 2008: Android OS

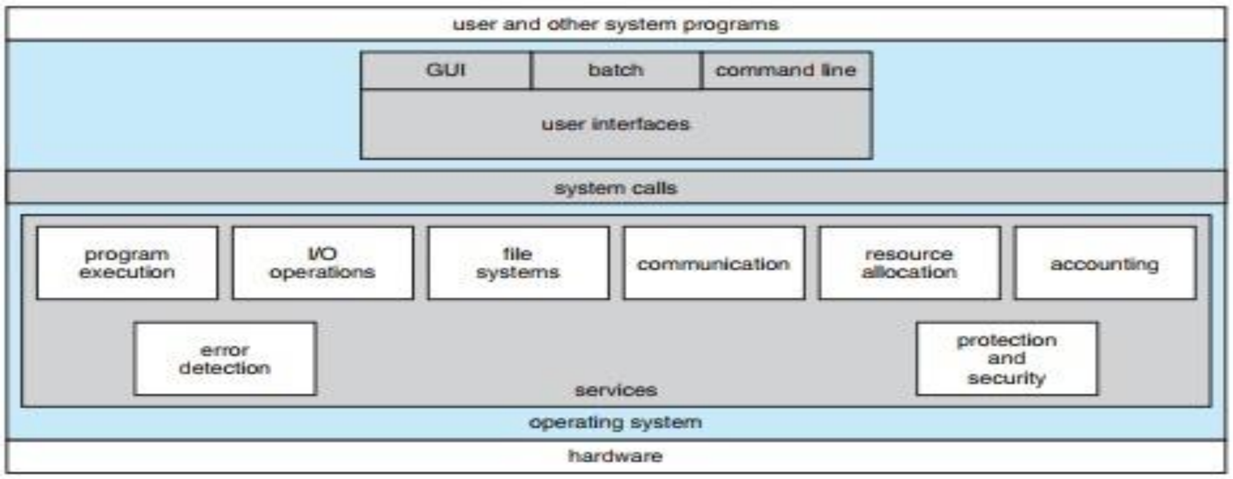
And as the research and development work still goes on, with new operating systems being developed and existing ones getting improved and modified to enhance the overall user experience, making operating systems fast and efficient like never before.

OPERATING SYSTEM SERVICES

An Operating System provides services to both the users and to the programs.

- It provides programs an environment to execute.
- It provides users the services to execute the programs in a convenient manner.

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.



A view of operating system services.

The above Figure shows one view of the various operating-system services and how they interrelate.

- User Interface
- Program Execution
- I/O operations
- File System manipulation
- Communication
- Error Detection
- Resource Allocation
- Accounting
- Protection and Security

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 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 Operation:

An I/O subsystem comprises of I/O devices and their corresponding driver software. Drivers hide the peculiarities of specific hardware devices from the users.

An Operating System manages the communication between user and device drivers.

- I/O operation means read or write operation with any file or any specific I/O device.
- Operating system provides the access to the required I/O device when required.

File system manipulation:

A file represents a collection of related information. Computers can store files on the disk (secondary storage), for long-term storage purpose. Examples of storage media include magnetic tape, magnetic disk and optical disk drives like CD, DVD. Each of these media has its own properties like speed, capacity, data transfer rate and data access methods.

A file system is normally organized into directories for easy navigation and usage. These directories may contain files and other directions.

Following are the major activities of an operating system with respect to file management −

- Program needs to read a file or write a file.
- The operating system gives the permission to the program for operation on file.
- Permission varies from read-only, read-write, denied and so on.
- Operating System provides an interface to the user to create/delete files.
- Operating System provides an interface to the user to create/delete directories.
- Operating System provides an interface to create the backup of file system.

Communication:

In case of distributed systems which are a collection of processors that do not share memory, peripheral devices, or a clock, the operating system manages communications between all the processes. Multiple processes communicate with one another through communication lines in the network.

The OS handles routing and connection strategies, and the problems of contention and security. Following are the major activities of an operating system with respect to communication −

- Two processes often require data to be transferred between them
- Both the processes can be on one computer or on different computers, but are connected through a computer network.
- Communication may be implemented by two methods, either by Shared Memory or by Message Passing.

Error detection:

Errors can occur anytime and anywhere. An error may occur in CPU, in I/O devices or in the memory hardware.

Following are the major activities of an operating system with respect to error handling −

- The OS constantly checks for possible errors.
- The OS takes an 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

In case of multi-user or multi-tasking environment, resources such as main memory, CPU cycles and files storage are to be allocated to each user or job.

Following are the major activities of an operating system with respect to resource management −

- The OS manages all kinds of resources using schedulers.
- CPU scheduling algorithms are used for better utilization of CPU.

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:

Considering a computer system having multiple users and concurrent execution of multiple processes, the various processes must be protected from each other's activities.

Protection refers to a mechanism or a way to control the access of programs, processes, or users to the resources defined by a computer system.

Following are the major activities of an operating system with respect to protection −

- The OS ensures that all access to system resources is controlled.
- The OS ensures that external I/O devices are protected from invalid access attempts.
- The OS provides authentication features for each user by means of passwords.

USER AND OPERATING SYSTEM INTERFACE

We mentioned earlier that 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- a GUI -Gra**phical User Interfaces.**

Command Line Interpreter:

Some operating systems include the command interpreter in the kernel. Some, such as the popular Windows and Linux operating systems, use the command interpreter as a special program that runs when a user logs on or a job is initiated.

- In Windows, this is the MS-DOS prompt.
- Linux has more options. The command interpreter in Linux is known as a shell. The most commonly used shell is the Bash shell, but others such as the Korn shell, C shell, and Bourne shell exist. Most shells provide similar functionality, personal preference usually dictates which shell is best.
- The main function of the command utility is to receive and execute the next user generated command.
- Many commands are intended to manipulate files.
- Operating systems such as UNIX implements commands through system programs. Often these programs are stored as text files, which allow programmers to add additional functionality to the utility.

Thus, the UNIX command to delete a file

Root rm file.txt

would search for a file called rm, load the file into memory, and execute it with the parameter file.txt.

Graphical User Interfaces – 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.

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

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.

A GUI provides a mouse-based windows and menu system as an interface. Users of Windows are more likely to use the GUI rather than the command line interface of MS-DOS, while UNIX users generally prefer using the command line interface of the shell rather than the GUI.

SYSTEM CALLS

System calls provide an interface to the services made available by an operating system.

Application developers often do not have direct access to the system calls, but can access them through an application programming interface (API). The functions that are included in the API invoke the actual system calls. By using the API, certain benefits can be gained:

- Portability: as long a system supports an API, any program using that API can compile and run.
- Ease of Use: using the API can be significantly easier than using the actual system call.

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.

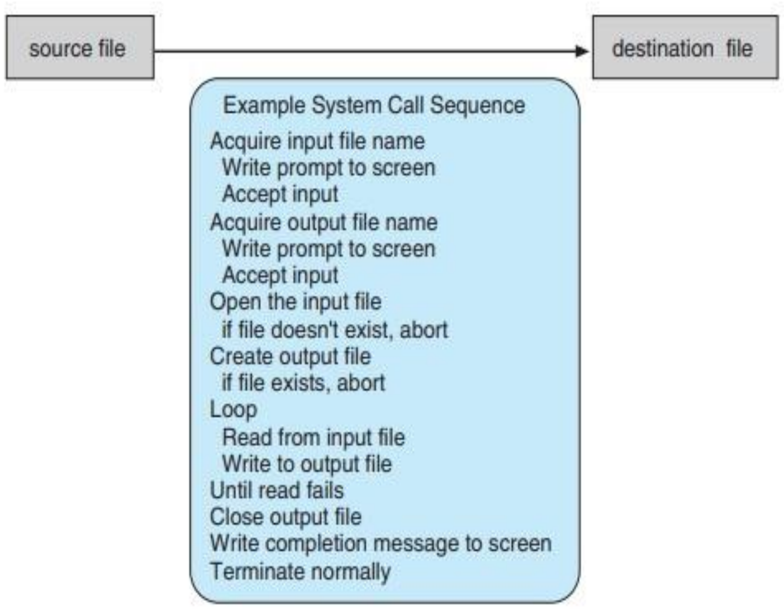
Before we discuss how an operating system makes system calls available, let's first use an example to illustrate how system calls are used: writing a simple program to read data from one file and copy them to another file.

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.



Example of how system calls are used.

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.

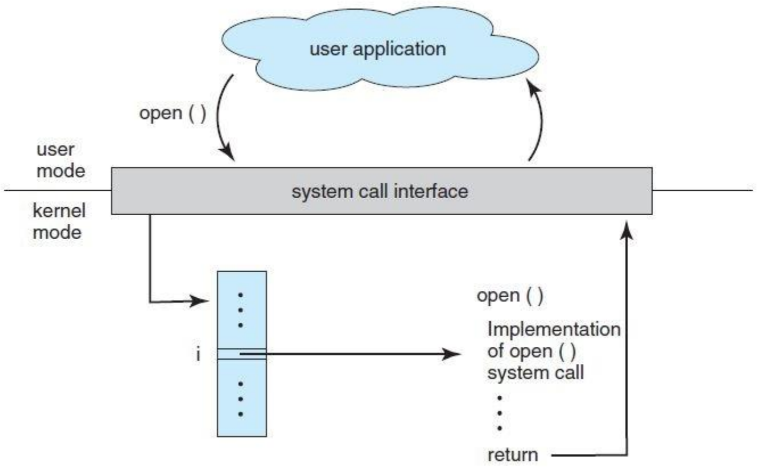
When both files are set up, we enter a loop that reads from the input file (a system call) and writes to the output file (another system call). Each read and write must return status information regarding various possible error conditions.

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 above Figure.

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)**.

- Programming interface to the services provided by the OS
- Typically written in a high-level language (C or C++)
- Mostly accessed by programs via a high-level Application Program Interface (API)
- Three most common APIs are
    1. Win32 API for Windows,
    2. POSIX API (all versions of UNIX, Linux, and Mac OS X), and
    3. Java API for the Java virtual machine (JVM)

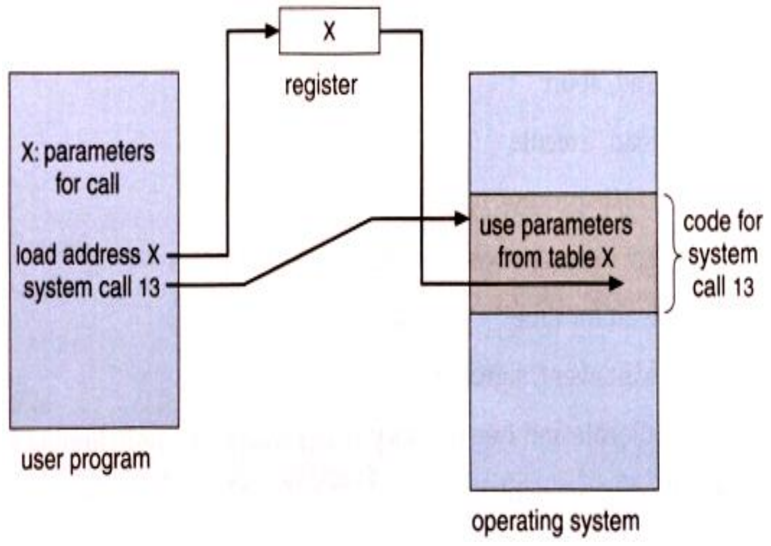


The handling of a user application invoking the open() system call.

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, which illustrates how the operating system handles a user application invoking the open() system call.

System calls Parameters:

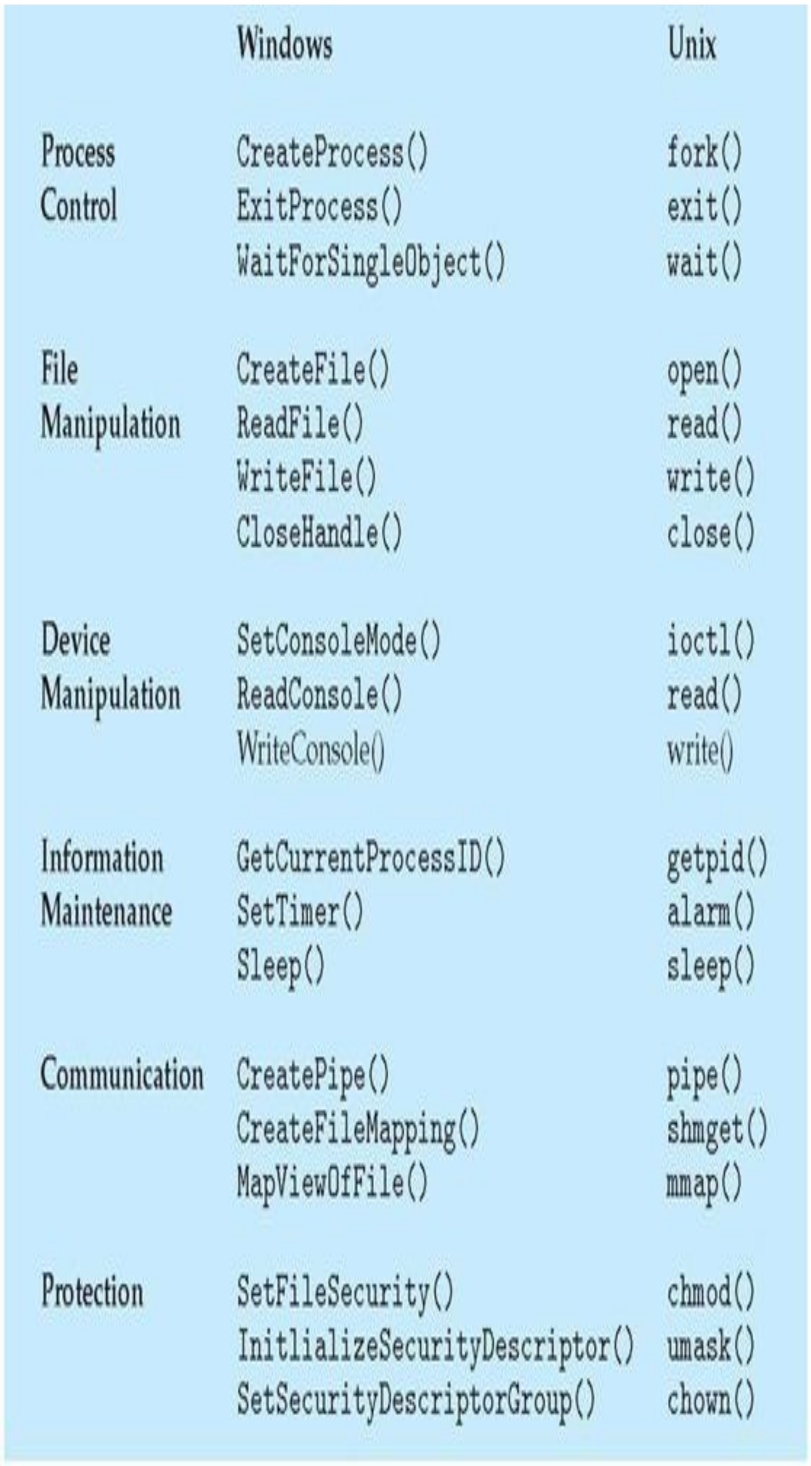
Three general methods exist for passing parameters to the OS:



1. Parameters can be passed in registers.
2. When there are more parameters than registers, parameters can be stored in a block and the block address can be passed as a parameter to a register.
3. Parameters can also be pushed on or popped off the stack by the operating system.

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.



	Windows	Unix
Process Control	CreateProcess()	fork()
	ExitProcess()	exit()
	WaitForSingleObject()	wait()
File Manipulation	CreateFile()	open()
	ReadFile()	read()
	WriteFile()	write()
	CloseHandle()	close()
Device Manipulation	SetConsoleMode()	ioctl()
	ReadConsole()	read()
	WriteConsole()	write()
Information Maintenance	GetCurrentProcessID()	getpid()
	SetTimer()	alarm()
	Sleep()	sleep()
Communication	CreatePipe()	pipe()
	CreateFileMapping()	shmget()
	MapViewOfFile()	mmap()
Protection	SetFileSecurity()	chmod()
	InitlializeSecurityDescriptor()	umask()
	SetSecurityDescriptorGroup()	chown()

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

- request device, release device
- read, write, reposition
- get device attributes,
- set device attributes
- logically attach or detach devices

Communication

- create, delete comm. connection
- send, receive messages
- transfer status information
- attach or detach remote devices

Process Control:

A running program needs to be able to stop execution either normally or abnormally. When execution is stopped abnormally, often a dump of memory is taken and can be examined with a debugger.

File Management:

Some common system calls are *create*, *delete*, *read*, *write*, *reposition*, or *close*. Also, there is a need to determine the file attributes – *get* and *set* file attribute. Many times the OS provides an API to make these system calls.

Device Management:

Process usually requires several resources to execute, if these resources are available, they will be granted and control returned to the user process. These resources are also thought of as devices. Some are physical, such as a video card, and others are abstract, such as a file.

User programs *request* the device, and when finished they *release* the device. Similar to files, we can *read*, *write*, and *reposition* the device.

Information Management:

Some system calls exist purely for transferring information between the user program and the operating system. An example of this is *time*, or *date*.

The OS also keeps information about all its processes and provides system calls to report this information.

Communication:

There are two models of interprocess communication, the message-passing model and the shared memory model.

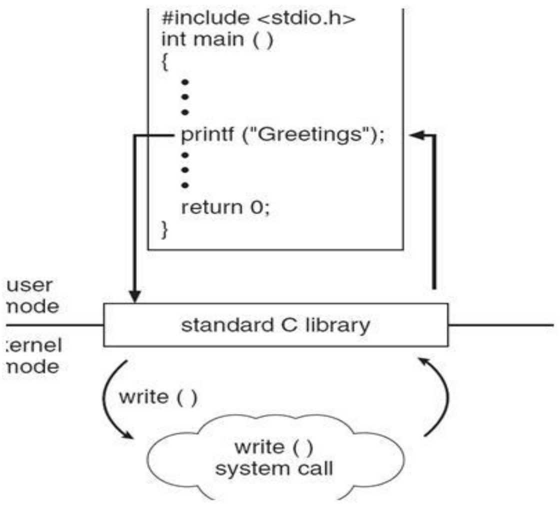
- Message-passing uses a common mailbox to pass messages between processes.
- Shared memory use certain system calls to create and gain access to create and gain access to regions of memory owned by other processes. The two processes exchange information by reading and writing in the shared data.

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.

Example of Standard C Library:



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

- The standard C library provides a portion of the system call interface for many version of UNIX and Linux.
- As an example, let's assume a C program invokes printf() statement.
- The C library intercepts this call and invokes the necessary system call(s) in the OS.
- The C library takes the value returned by write() and passes it back to the user program

OPERATING-SYSTEM DESIGN AND IMPLEMENTATION

In this section, we discuss problems we face in designing and implementing an operating system. There are, of course, no complete solutions to such problems, but there are approaches that have proved successful.

Design Goals

The first problem in designing a system is to define goals and specifications. At the highest level, the design of the system will be affected by the choice of hardware and the type of system: batch, time sharing, single user, multiuser, distributed, real time, or general purpose.

Beyond this highest design level, the requirements may be much harder to specify.

The requirements can, however, be divided into two basic groups:

User goals and

System goals.

Users want certain obvious properties in a system. The system should be convenient to use, easy to learn and to use, reliable, safe, and fast. Of course, these specifications are not particularly useful in the system design, since there is no general agreement on how to achieve them. A similar set of requirements can be defined by those people who must design, create, maintain, and operate the system. The system should be easy to design, implement, and maintain; and it should be flexible, reliable, error free, and efficient. Again, these requirements are vague and may be interpreted in various ways.

Mechanisms and Policies

One important principle is the separation of **policy** from **mechanism**.

Mechanisms determine ***how*** to do something; policies determine ***what*** will be done.

For example, the timer construct is a mechanism for ensuring CPU protection, but deciding how long the timer is to be set for a particular user is a policy decision. The separation of policy and mechanism is important for flexibility.

Policies are likely to change across places or over time. In the worst case, each change in policy would require a change in the underlying mechanism. A general mechanism insensitive to changes in policy would be more desirable. A change in policy would then require redefinition of only certain parameters of the system. For instance, consider a mechanism for giving priority to certain types of programs over others. If the mechanism is properly separated from policy, it can be used either to support a policy decision that I/O-intensive programs should have priority over CPU-intensive ones or to support the opposite policy.

Policy decisions are important for all resource allocation. Whenever it is necessary to decide whether or not to allocate a resource, a policy decision must be made. Whenever the question is ***how*** rather than ***what***, it is a mechanism that must be determined.

Implementation

Once an operating system is designed, it must be implemented. Because operating systems are collections of many programs, written by many people over a long period of time, it is difficult to make general statements about how they are implemented.

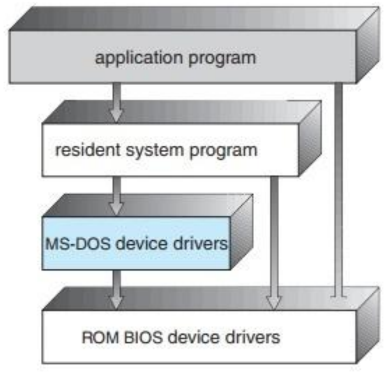
Early operating systems were written in assembly language. Now, although some operating systems are still written in assembly language, most are written in a higher-level language such as C or an even higher-level language such as C++. Actually, an operating system can be written in more than one language.

The lowest levels of the kernel might be assembly language. Higher-level routines might be in C, and system programs might be in C or C++, in interpreted scripting languages like PERL or Python, or in shell scripts. In fact, a given Linux distribution probably includes programs written in all of those languages.

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

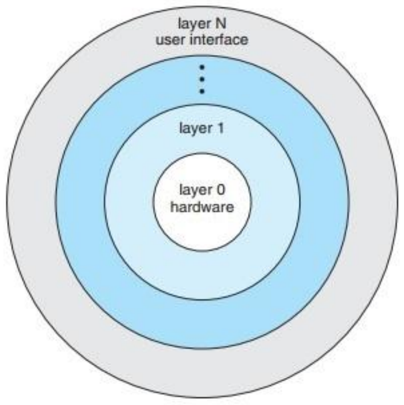


MS-DOS layer structure.

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 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.

Layered Approach



A layered operating system.

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.

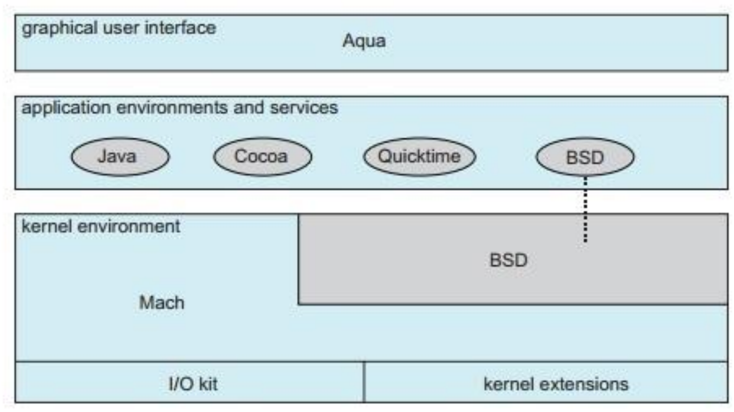
An operating-system layer is an implementation of an abstract object made up of data and the operations that can manipulate those data. A typical operating-system layer—say, layer *M*—consists of data structures and a set of routines that can be invoked by higher-level layers. Layer *M*, in turn, can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

The main advantage of the layered approach is simplicity of construction and debugging.

The major difficulty with the layered approach involves appropriately defining the various layers.

Microkernels

In a **microkernel** (also known as μ-kernel) is the near-minimum amount of software that can provide the mechanisms needed to implement an**operating system** (**OS**). These mechanisms include low-level address space management, thread management, and inter-process communication (IPC).



The Mac OS X structure.

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.

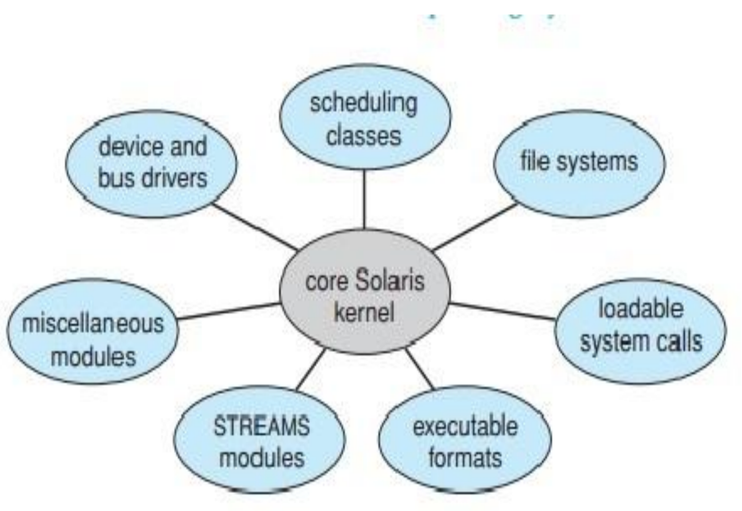
Typically, however, microkernels provide minimal process and memory management, in addition to a communication facility. Figure illustrates the architecture of a typical microkernel.

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,**

One benefit of the microkernel approach is that it makes extending the operating system easier.

Example: The Mac OS-X kernel (also known as **Darwin**) is also partly based on the Mach microkernel. Another example is QNX, a real-time operating system for embedded systems.

Modules



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

1. Scheduling classes
2. File systems
3. Loadable system calls
4. Executable formats
5. STREAMS modules
6. Miscellaneous
7. Device and bus drivers

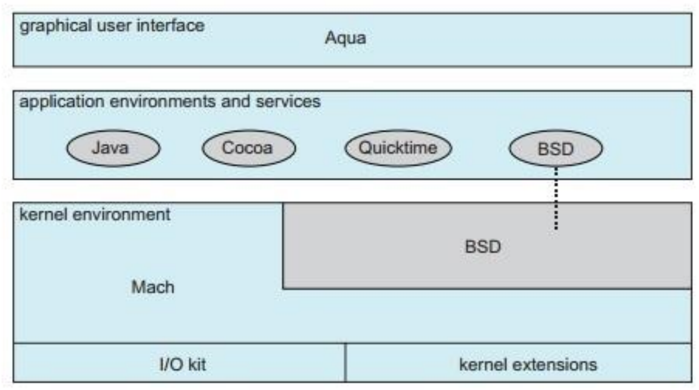
Hybrid Systems

A **hybrid** kernel is an **operating system** kernel architecture that attempts to combine aspects and benefits of microkernel and monolithic kernel architectures used in computer **operating 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.

Mac OS X

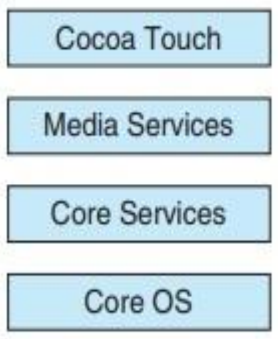


The Mac OS X structure.

The Apple Mac OS X operating system uses a hybrid structure. As shown in Figure , it is a layered system. The top layers include the ***Aqua*** user interface 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.

iOS



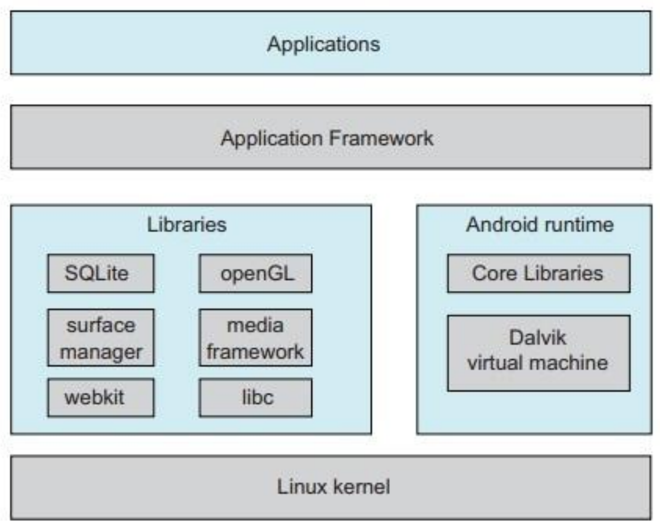
Architecture of Apple's iOS.

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.

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.

Android



Architecture of Google's 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.

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.

SYSTEM PROGRAMS

Another aspect of a modern system is its collection of system programs. In 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.

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 process schedulers that start processes according to a specified schedule, system error monitoring services, and print servers. Typical systems have dozens of daemons.

In addition, operating systems that run important activities in user context rather than in kernel context may use daemons to run these activities.

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.

OPEN-SOURCE OPERATING SYSTEMS

Operating systems has been made easier by the availability of a vast number of open-source 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. With the source code in hand, a student can modify the operating system and then compile and run the code to try out those changes, which is an excellent learning tool.

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.

VIRTUAL MACHINE

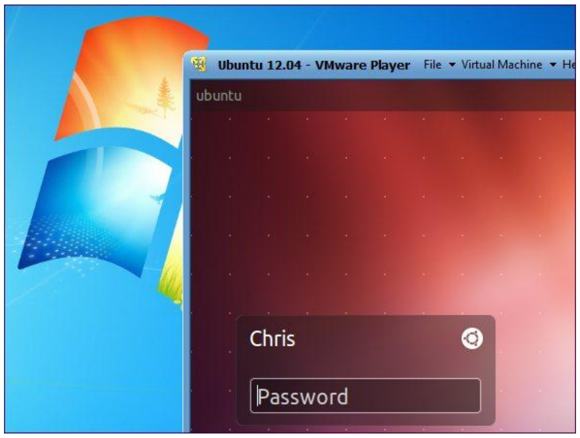
A virtual machine is a program that acts as a virtual computer. It runs on your current operating system – the “host” operating system – and provides virtual hardware to “guest” operating systems.

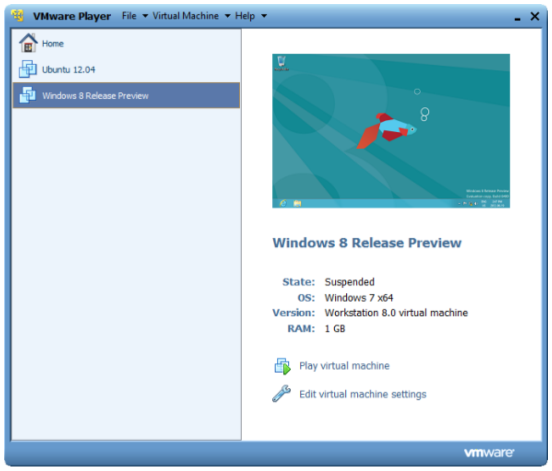
The guest operating systems run in windows on your host operating system, just like any other program on your computer.

The guest operating system runs normally, as if it were running on a physical computer – from the guest operating system’s perspective, the virtual machine appears to be a real, physical computer.

Virtual machines provide their own virtual hardware, including a virtual CPU, memory, hard drive, network interface, and other devices. The virtual hardware devices provided by the virtual machine are mapped to real hardware on your physical machine. For example, a virtual machine's virtual hard disk is stored in a file located on your hard drive.

You can have several virtual machines installed on your system; you're only limited by the amount of storage you have available for them. Once you've installed several operating systems, you can open your virtual machine program and choose which virtual machine you want to boot – the guest operating system starts up and runs in a window on your host operating system, although you can also run it in full-screen mode.





Windows 8 Release Preview

State: Suspended
OS: Windows 7 x64
Version: Workstation 8.0 virtual machine
RAM: 1 GB

Uses for Virtual Machines:

Virtual machines have a number of popular uses:

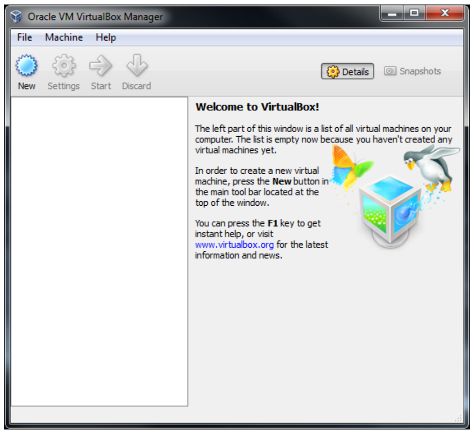
- **Test new versions of operating systems**: You can **run the development version of Windows 8 in a virtual machine** on your Windows 7 computer. This allows you to experiment with Windows 8 without **installing an unstable version of Windows on your computer**.
- **Experiment with other operating systems**: You can **install various distributions of Linux and other more obscure operating systems** in a virtual machine to experiment with them and learn how they work. If you're interested in **Ubuntu**, you can install it in a virtual machine and play with it at your own pace — in a window on your normal desktop.
- **Use software requiring an outdated operating system**: If you've got an important application that only runs on Windows XP, you can **install XP in a virtual machine** and run the application in the virtual machine. The virtual machine is actually running Windows XP, so compatibility shouldn't be a problem. This allows you to use an application that only works with Windows XP without actually installing Windows XP on your computer – especially important considering many new laptops and other hardware may not fully support Windows XP.

- **Run software designed for another operating system**s: Mac and Linux users can run Windows in a virtual machine to run Windows software on their computers without the compatibility headaches of **Wine** and Crossover. Unfortunately, games can be a problem – virtual machine programs introduce overhead and no virtual machine application will allow you to run the latest 3D games in a virtual machine. Some 3D effects are supported, but **3D graphics** are the least well supported thing you can do in a virtual machine.
- **Test software on multiple platforms**: If you need to test whether an application works on multiple operating systems – or just different versions of Windows – you can install each in a virtual machine instead of keeping separate computers around for each.
- **Consolidate servers**: For businesses running multiple **servers**, existing servers can be placed into virtual machines and run on a single computer. Each virtual machine is an isolated container, so this doesn't introduce the security headaches involved with running different servers on the same operating system. The virtual machines can also be moved between physical servers.

Recommended Virtual Machine Software

VirtualBox is a great, open-source application that runs on Windows, Mac OS X, and Linux. One of the best things about VirtualBox is that there's no commercial version – you get all the features for free, including advanced features like "snapshots," which allow you to take a snapshot of a virtual machine's state and revert to that state in the future – a great feature for testing.

VMware Player is another high-quality virtual machine program for Windows and Linux. VMware Player is the free counterpart to VMware Workstation, a commercial application, so you don't get all the advanced features you would with VirtualBox. However, both VirtualBox and VMware Player are solid programs that offer the basic features – creating and running virtual machines – for free. If one of them doesn't work quite right, try the other.

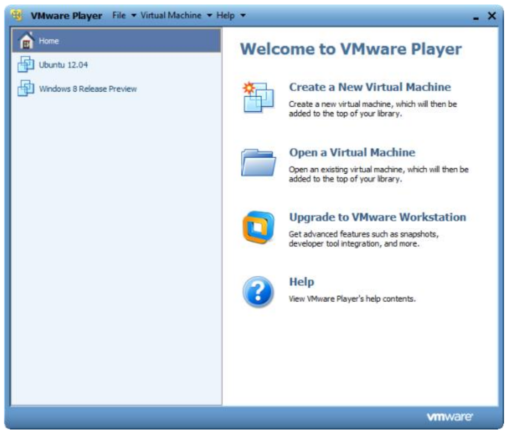


Welcome to VirtualBox!

The left part of this window is a list of all virtual machines on your computer. The list is empty now because you haven't created any virtual machines yet.

In order to create a new virtual machine, press the **New** button in the main tool bar located at the top of the window.

You can press the **F1** key to get instant help, or visit www.virtualbox.org for the latest information and news.



Welcome to VMware Player

Create a New Virtual Machine
Create a new virtual machine, which will then be added to the top of your library.

Open a Virtual Machine
Open an existing virtual machine, which will then be added to the top of your library.

Upgrade to VMware Workstation
Get advanced features such as snapshots, developer tool integration, and more.

Help
View VMware Player's help contents.

OPERATING-SYSTEM GENERATION

It is possible to design, code, and implement an operating system specifically for one machine at one site. More commonly, however, operating systems are designed to run on any of a class of machines at a variety of sites with a variety of peripheral configurations.

The system must then be configured or generated for each specific computer site, a process sometimes known as **system generation SYSGEN**.

The operating system is normally distributed on disk, on CD-ROM or DVD-ROM, or as an "ISO" image, which is a file in the format of a CD-ROM or DVD-ROM. To generate a system, we use a special program. This SYSGEN program reads from a given file, or asks the operator of the system for information concerning the specific configuration of the hardware system, or probes the hardware directly to determine what components are there. The following kinds of information must be determined.

- **What CPU is to be used?**

What options (extended instruction sets, floating point arithmetic, and so on) are installed? For multiple CPU systems, each CPU may be described.

- **How will the boot disk be formatted?**

How many sections, or "partitions," will it be separated into, and what will go into each partition? How much memory is available? Some systems will determine this value
themselves by referencing memory location after memory location until an "illegal address" fault is generated. This procedure defines the final legal address and hence the amount of available memory.

- **What devices are available?**

The system will need to know how to address each device (the device number), the device interrupt number, the device's type and model, and any special device characteristics.

- **What operating-system options are desired, or what parameter values are to be used?**

These options or values might include how many buffers of which sizes should be used, what type of CPU-scheduling algorithm is desired, what the maximum number of processes to be supported is, and so on.

Once this information is determined, it can be used in several ways. At one extreme, a system administrator can use it to modify a copy of the source code of the operating system. The operating system then is completely compiled. Data declarations, initializations, and constants, along with conditional compilation, produce an output-object version of the operating system that is tailored to the system described.

At a slightly less tailored level, the system description can lead to the creation of tables and the selection of modules from a precompiled library. These modules are linked together to form the generated operating system. Selection allows the library to contain the device drivers for all supported I/O devices, but only those needed are linked into the operating system. Because the system is not recompiled, system generation is faster, but the resulting system may be overly general. At the other extreme, it is possible to construct a system that is completely table driven. All the code is always part of the system, and selection occurs at execution time, rather than at compile or link time. System generation involves simply creating the appropriate tables to describe the system.

The major differences among these approaches are the size and generality of the generated system and the ease of modifying it as the hardware configuration changes. Consider the cost of modifying the system to support a newly acquired graphics terminal or another disk drive. Balanced against that cost, of course, is the frequency (or infrequency) of such changes.

SYSTEM BOOT

[

- Booting the system is done by loading the kernel into main memory, and starting its execution.
- The CPU is given a reset event, and the instruction register is loaded with a predefined memory location, where execution starts.
    - The initial bootstrap program is found in the BIOS read-only memory.
    - This program can run diagnostics, initialize all components of the system, loads and starts the Operating System loader. (Called **boot strapping**)
    - The loader program loads and starts the operating system.
    - When the Operating system starts, it sets up needed data structures in memory, sets several registers in the CPU, and then creates and starts the first user level program. From this point, the operating system only runs in response to interrupts.

]

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. <u>ROM is convenient because it needs no initialization and cannot easily be infected by a computer virus</u>.

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**, since their characteristics fall somewhere between those of hardware and those of software. A problem with firmware in general is that executing code there is slower than executing code in RAM.

Some systems store the operating system in firmware and copy it to RAM for fast execution. A final issue with firmware is that it is relatively expensive, so usually only small amounts are available.

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**.

UNIT – II PROCESS MANAGEMENT

- 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 question that arises in discussing operating systems involves what to call all the CPU activities. A batch system executes **jobs**, whereas a time-shared system has **user programs**, or **tasks**.

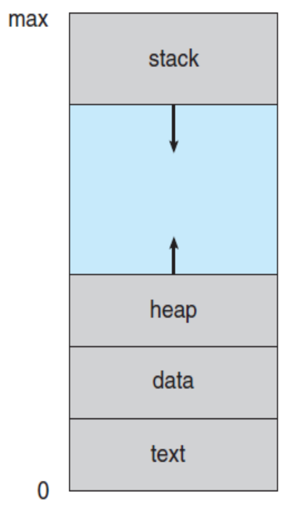
Even on a single-user system, a user may be able to run several programs at one time: a word processor, a Web browser, and an e-mail package.

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 in this text.

The Process:



- 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. We emphasize that a program by itself is not a process.

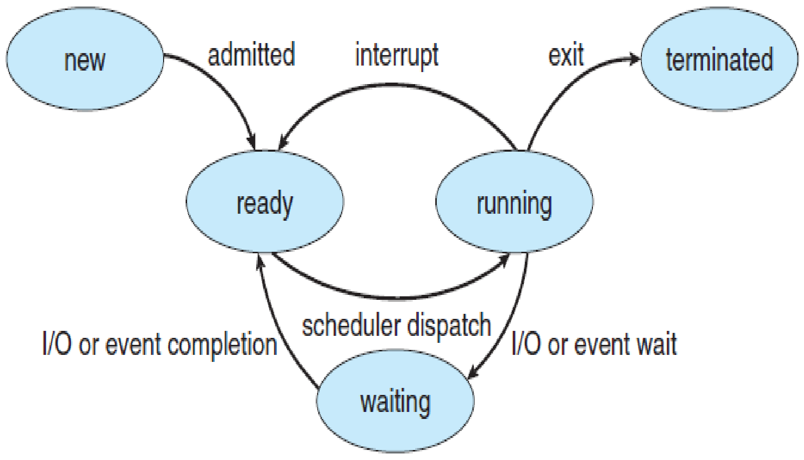
- A program is a ***passive*** entity, such as a file containing a list of instructions stored on disk (often called an **executable file**).
- In contrast, a process is an ***active*** entity, with a program counter specifying the next instruction to execute and a set of associated resources. A program 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).

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: The state diagram corresponding to these states is presented in above Figure.



New- The process is being created.

Running- Instructions are being executed.

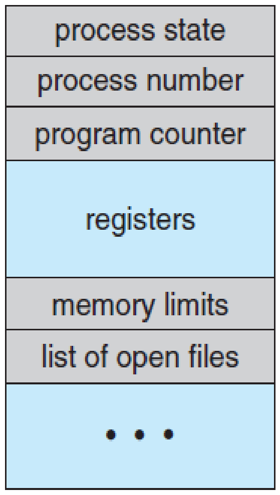
Waiting.-The process is waiting for some event to occur (such as an I/O completion)

Ready.-The process is waiting to be assigned to a processor.

Terminated- The process has finished execution.

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.*

Process Control Block-[PCB]:



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.

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.

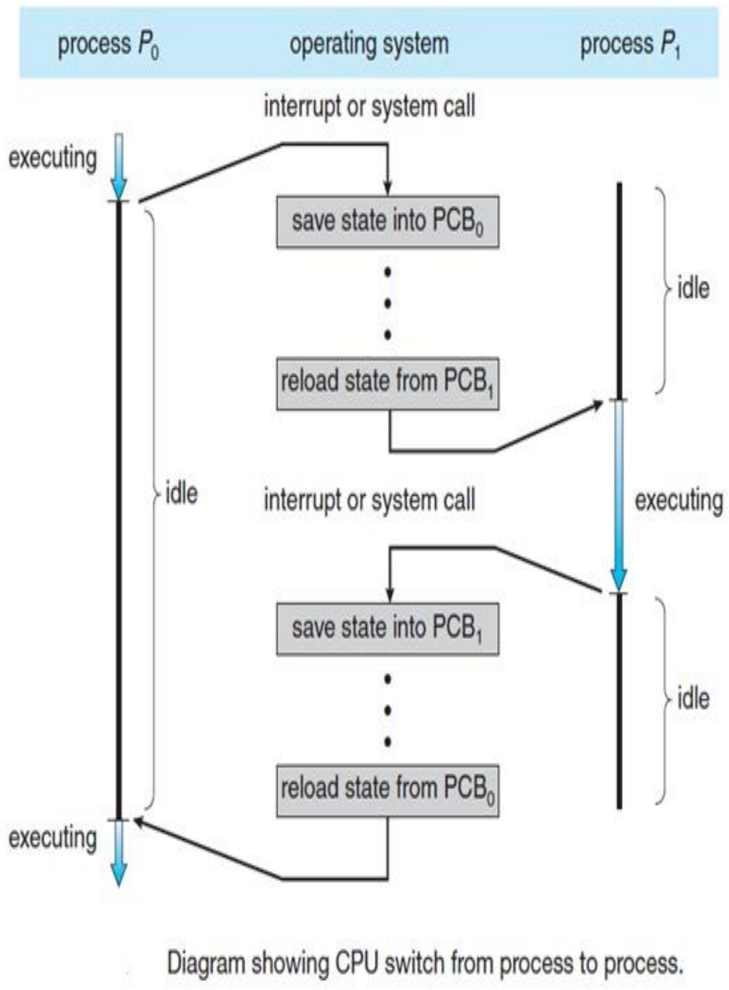


Diagram showing CPU switch from process to process.

- **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.
- **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.

In brief, the PCB simply serves as the repository for any information that may vary from process to process.

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.

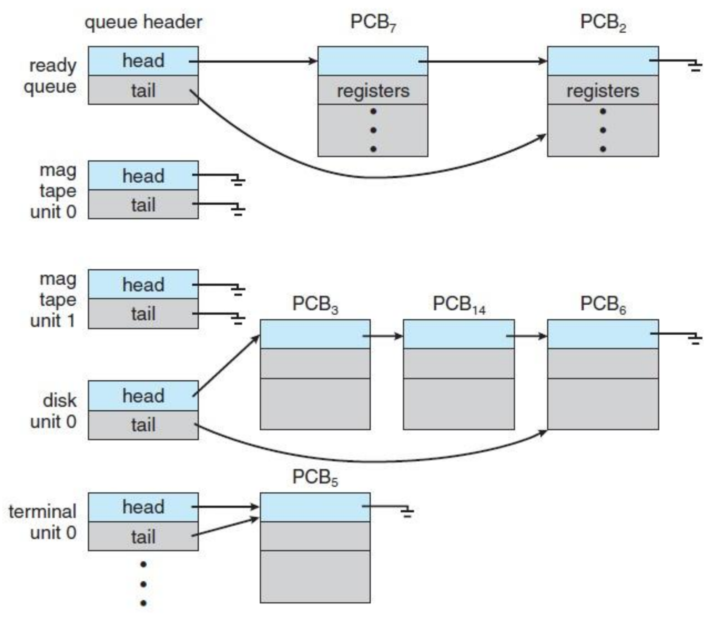
To meet these objectives, the **process scheduler** selects an available process (possibly from a set of several available processes) for program execution on the CPU.

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**.

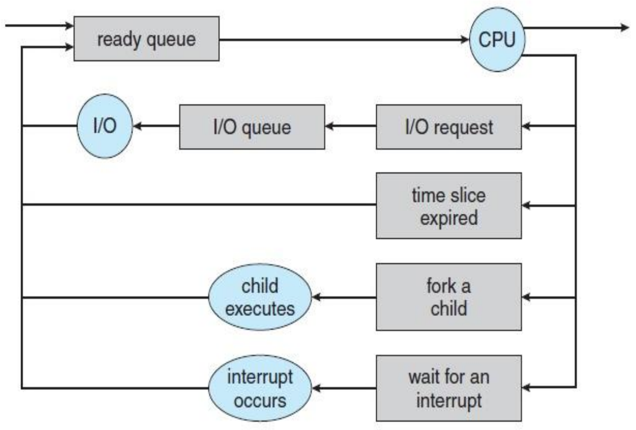


The ready queue and various I/O device queues.

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. Suppose the process makes an I/O request to a shared device, such as a disk.

Since there are many processes in the system, the disk may be busy with the I/O request of some other process. The process therefore may have to wait for the disk. The list of processes waiting for a particular I/O device is called a **device queue**. Each device has its own device queue (following Figure).



Queueing-diagram representation of process scheduling.

A common representation of process scheduling is a **queuing diagram**, such as that in Figure. 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.

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 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).

Some operating systems, such as time-sharing systems, may introduce an additional, intermediate level of scheduling. This **medium-term scheduler** is diagrammed in Figure. 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:

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.

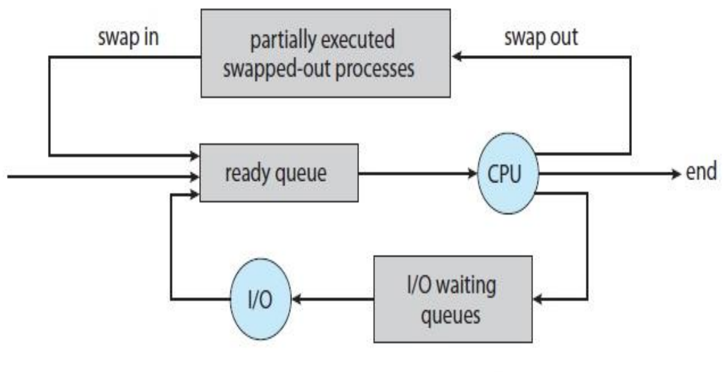


Figure 3.7 Addition of medium-term scheduling to the queueing diagram.

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.

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.

In this section, we explore the mechanisms involved in creating processes and illustrate process creation on UNIX and Windows systems.

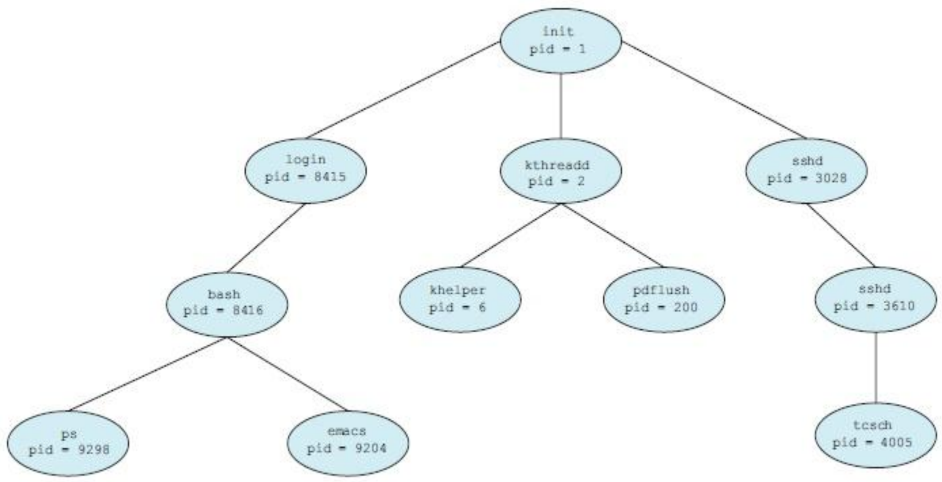
Process Creation

During the course of execution, a process may create several new processes.

As mentioned earlier,

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.



A tree of processes on a typical Linux system.

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 With in the kernel.

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:

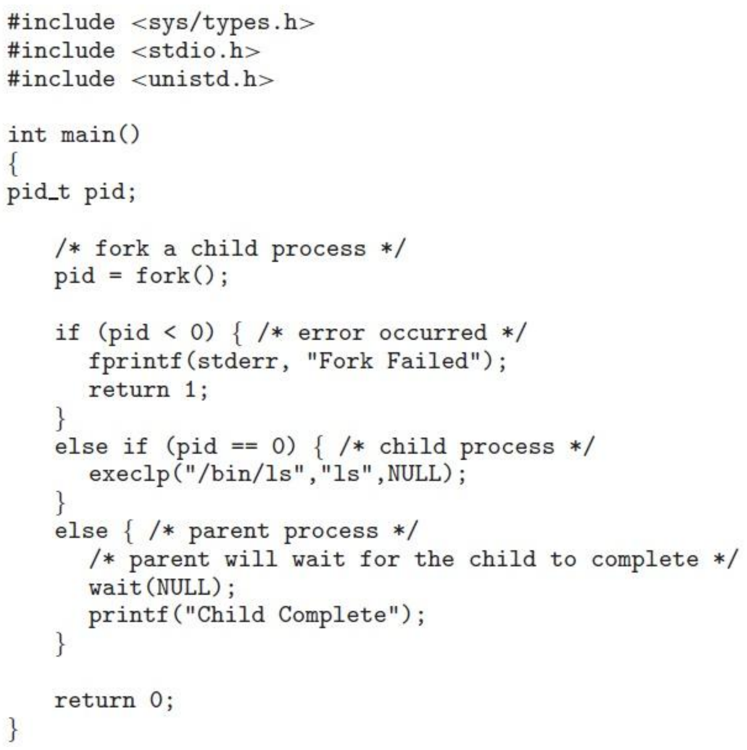
1. The parent continues to execute concurrently with its children.

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 (same program and data as the parent).

2. The child process has a new program loaded into it.



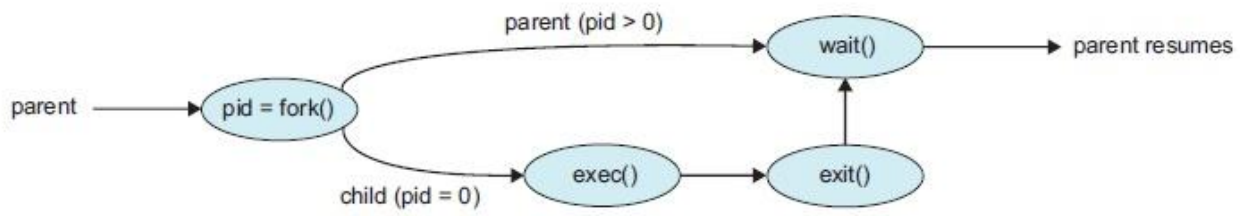
```
#include <sys/types.h>
#include <stdio.h>
#include <unistd.h>

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;
}
```

Figure - Creating a separate process using the UNIX fork() system call.

The C program shown in above Figure illustrates the UNIX system calls previously described. 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 the following Figure.



Process creation using the `fork()` system call.

Of course, there is nothing to prevent the child from ***not*** invoking exec()and continuing to execute as a copy of the parent process. In this scenario, the parent and child are concurrent processes running the same code instructions. Because the child is a copy of the parent, each process has its own copy of any data.

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 (for example, TerminateProcess() in Windows).

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.

To illustrate process execution and termination, consider that, in Linux and UNIX systems, we can terminate a process by using the exit() system call, providing an exit status as a parameter:

```
/* exit with status 1 */
exit(1);
```

In fact, under normal termination, exit() may be called either directly (as shown above) or indirectly (by a return statement in main()).

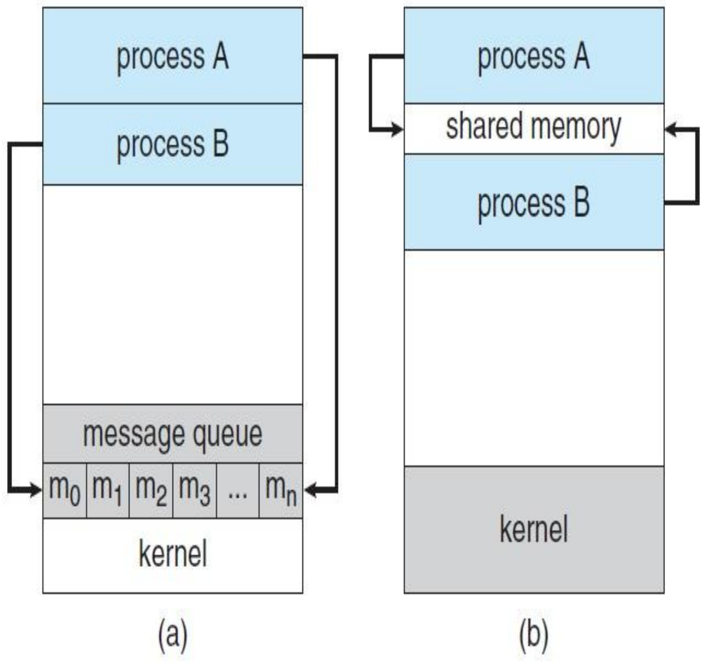
INTERPROCESS COMMUNICATION

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.



Communications models. (a) Message passing. (b) Shared memory.

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. 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.

• **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**
- **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 above Figure.

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.

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.

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,

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.

The producer – consumer problem also provides a useful metaphor for the client – server paradigm. We generally think of a server as a producer and a client as a consumer. For example, a web server produces (that is, provides) HTML files and images, which are consumed (that is, read) by the client web browser requesting the resource.

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.

```
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;
}
```

The producer process using shared memory.

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

The shared-memory environment requires that these processes share a region of memory and that the code for accessing and manipulating the shared memory be written explicitly by the application programmer.

Another way to achieve the same effect is for the operating system to provide the means for cooperating processes to communicate with each other via a message-passing facility.

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 straight-forward. 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.

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.

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 either **blocking** or **nonblocking**— also known as **synchronous** and **asynchronous**. (Throughout this text, you will encounter the concepts of synchronous and asynchronous behavior in relation to various operating-system algorithms.)

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.

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.

COMMUNICATION IN CLIENT – SERVER SYSTEMS

how processes can communicate using shared memory and message passing?.

These techniques can be used for communication in client – server systems. Communication in client – server systems may use

(1) sockets,

(2) remote procedure calls (RPCs),

(3) pipes.

Pipes provide a relatively simple ways for processes to communicate with one another. Ordinary pipes allow communication between parent and child processes, while named pipes permit unrelated processes to communicate.

Sockets:

A **socket** is defined as an endpoint for communication.

A socket is defined as an endpoint for communication. A connection between a pair of applications consists of a pair of sockets, one at each end of the communication channel.

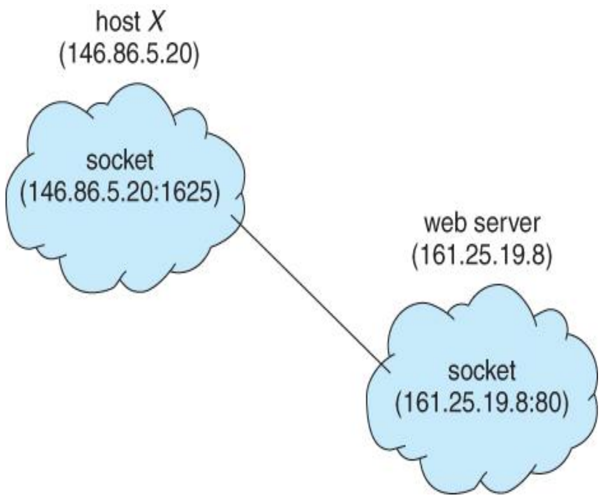
A pair of processes communicating over a network employs a pair of sockets — one for each process. A socket is identified by an IP address concatenated with a port number.

In general, sockets use a client – server architecture. The server waits for incoming client requests by listening to a specified port.

Once a request is received, the server accepts a connection from the client socket to complete the connection.

Servers implementing specific services (such as telnet, FTP, and HTTP) listen to well-known ports (a telnet server listens to port 23; an FTP server listens to port 21; and a web, or HTTP, server listens to port 80).

All ports below 1024 are considered ***well known;*** we can use them to implement standard services.



When a client process initiates a request for a connection, it is assigned a port by its host computer. This port has some arbitrary number greater than 1024. For example, if a client on host X with IP address 146.86.5.20 wishes to establish a connection with a web server (which is listening on port 80) at address 161.25.19.8, host X may be assigned port 1625. The connection will consist of a pair of sockets: (146.86.5.20:1625) on host X and (161.25.19.8:80) on the web server. This situation is illustrated in Figure 3.20.

The packets traveling between the hosts are delivered to the appropriate process based on the destination port number.

- Communication channels via sockets may be of one of two major forms:
  - **Connection-oriented (TCP, Transmission Control Protocol)** connections emulate a telephone connection. All packets sent down the connection are guaranteed to arrive in good condition at the other end, and to be delivered to the receiving process in the order in which they were sent. The TCP layer of the network protocol takes steps to verify all packets sent, re-send packets if necessary, and arrange the received packets in the proper order before delivering them to the receiving process. There is a certain amount of overhead involved in this procedure, and if one packet is missing or delayed, then any packets which follow will have to wait until the errant packet is delivered before they can continue their journey.
  - **Connectionless (UDP, User Datagram Protocol)** emulate individual telegrams. There is no guarantee that any particular packet will get through undamaged (or at all), and no

guarantee that the packets will get delivered in any particular order. There may even be duplicate packets delivered, depending on how the intermediary connections are configured. UDP transmissions are much faster than TCP, but applications must implement their own error checking and recovery procedures.

- Sockets are considered a low-level communications channel, and processes may often choose to use something at a higher level, such as those covered in the next two sections.

Remote Procedure Calls

In contrast IPC messages, the messages exchanged in RPC communication are well structured and are thus no longer just packets of data.

RPCs are another form of distributed communication. An RPC occurs when a process (or thread) calls a procedure on a remote application.

Each message is addressed to an RPC daemon listening to a port on the remote system, and each contains an identifier specifying the function to execute and the parameters to pass to that function.

The function is then executed as requested, and any output is sent back to the requester in a separate message.

A **port** is simply a number included at the start of a message packet. Whereas a system normally has one network address, it can have many ports within that address to differentiate the many network services it supports.

If a remote process needs a service, it addresses a message to the proper port. For instance, if a system wished to allow other systems to be able to list its current users, it would have a daemon supporting such an RPC attached to a port — say, port 3027. Any remote system could obtain the needed information (that is, the list of current users) by sending an RPC message to port 3027 on the server. The data would be received in a reply message.

The semantics of RPCs allows a client to invoke a procedure on a remote host as it would invoke a procedure locally.

The RPC system hides the details that allow communication to take place by providing a **stub** on the client side.

Typically, a separate stub exists for each separate remote procedure. When the client invokes a remote procedure, the RPC system calls the appropriate stub, passing it the parameters provided to the remote procedure.

This stub locates the port on the server and **marshals** the parameters. Parameter marshalling involves packaging the parameters into a form that can be transmitted over a network. The stub then transmits a message to the server using message passing.

A similar stub on the server side receives this message and invokes the procedure on the server. If necessary, return values are passed back to the client using the same technique.

On Windows systems, stub code is compiled from a specification written in the **Microsoft Interface Definition Language (MIDL)**, which is used for defining the interfaces between client and server programs.

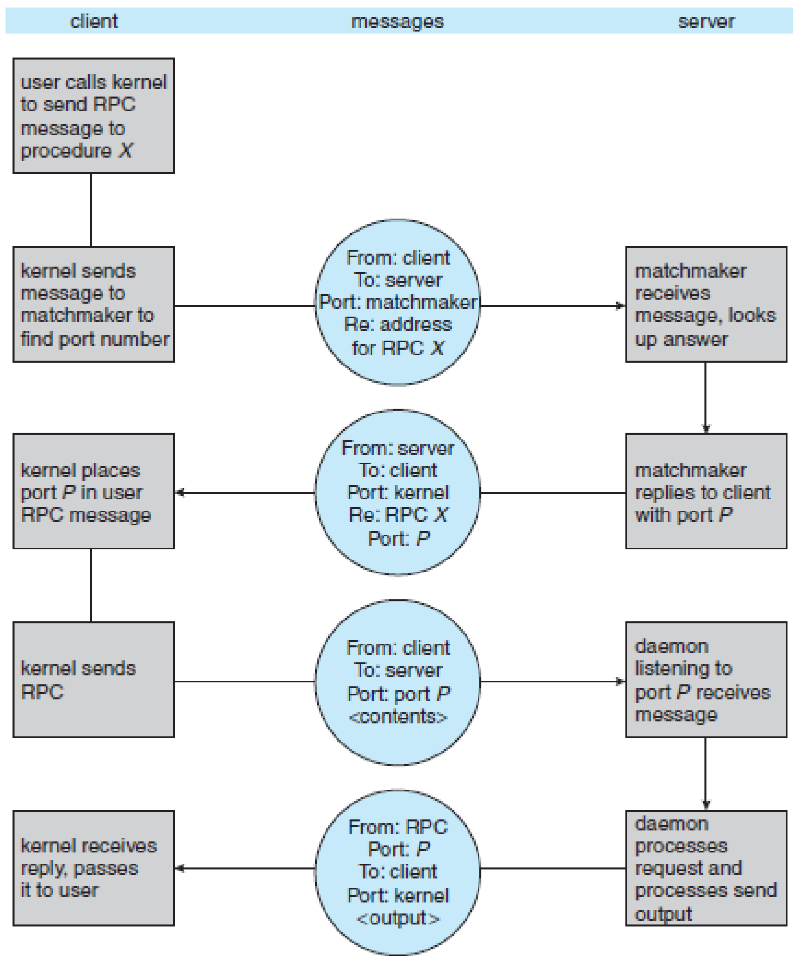

client
messages
server
user calls kernel to send RPC message to procedure *X*
kernel sends message to matchmaker to find port number
From: client To: server Port: matchmaker Re: address for RPC *X*
matchmaker receives message, looks up answer
kernel places port *P* in user RPC message
From: server To: client Port: kernel Re: RPC *X* Port: *P*
matchmaker replies to client with port *P*
kernel sends RPC
From: client To: server Port: port *P* <contents>
daemon listening to port *P* receives message
kernel receives reply, passes it to user
From: RPC Port: *P* To: client Port: kernel <output>
daemon processes request and processes send output

Figure 3.23 Execution of a remote procedure call (RPC).

Two approaches are common. First, the binding information may be predetermined, in the form of fixed port addresses. At compile time, an RPC call has a fixed port number associated with it. Once a program is compiled, the server cannot change the port number of the requested service. Second, binding can be done dynamically by a rendezvous mechanism. Typically, an operating system provides a rendezvous (also called a **matchmaker**) daemon on a fixed RPC port. A client then sends a message containing the name of the RPC to the rendezvous daemon requesting the port address of the RPC it needs to execute. The port number is returned, and the RPC calls can be sent to that port until the process terminates (or the server crashes). This method requires the extra overhead of the initial request but is more flexible than the first approach. Figure 3.23 shows a sample interaction.

Pipes

A **pipe** acts as a conduit allowing two processes to communicate. Pipes were one of the first IPC mechanisms in early UNIX systems. They typically provide one of the simpler ways for processes to communicate with one another, although they also have some limitations. In implementing a pipe, four issues must be considered:

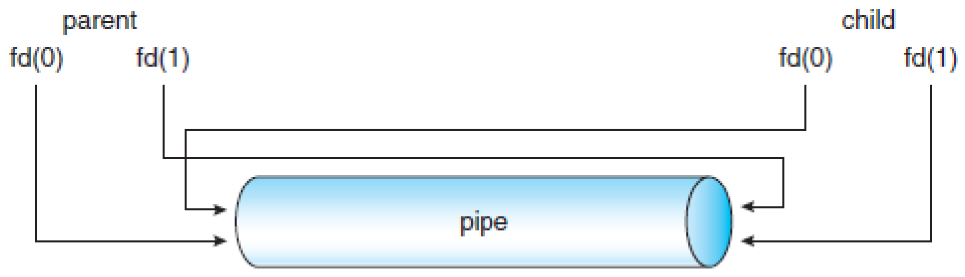


Figure 3.24 File descriptors for an ordinary pipe.

- Does the pipe allow bidirectional communication, or is communication unidirectional?
- If two-way communication is allowed, is it half duplex (data can travel only one way at a time) or full duplex (data can travel in both directions at the same time)?
- Must a relationship (such as ***parent–child***) exist between the communicating processes?
- Can the pipes communicate over a network, or must the communicating processes reside on the same machine?
- In the following sections, we explore two common types of pipes used on both UNIX and Windows systems: ordinary pipes and named pipes.

Ordinary Pipes

Ordinary pipes allow two processes to communicate in standard producer – consumer fashion:

the producer writes to one end of the pipe (the **write-end**) and the consumer reads from the other end (the **read-end**). As a result, ordinary pipes are unidirectional, allowing only one-way communication. If two-way communication is required, two pipes must be used, with each pipe sending data in a different direction. We next illustrate constructing ordinary pipes on both UNIX and Windows systems. In both program examples, one process writes the message Greetings to the pipe, while the other process reads this message from the pipe.

On UNIX systems, ordinary pipes are constructed using the function

pipe(int fd[])

This function creates a pipe that is accessed through the int fd[] file descriptors: fd[0] is the read-end of the pipe, and fd[1] is the write-end. UNIX treats a pipe as a special type of file. Thus, pipes can be accessed using ordinary read() and write() system calls.

Named Pipes

Ordinary pipes provide a simple mechanism for allowing a pair of processes to communicate. However, ordinary pipes exist only while the processes are communicating with one another.

On both UNIX and Windows systems, once the processes have finished communicating and have terminated, the ordinary pipe ceases to exist.

Named pipes provide a much more powerful communication tool. Communication can be bidirectional, and no parent – child relationship is required. Once a named pipe is established, several processes can use it for communication.

In fact, in a typical scenario, a named pipe has several writers. Additionally, named pipes continue to exist after communicating processes have finished. Both UNIX and Windows systems support named pipes, although the details of implementation differ greatly. Next, we explore named pipes in each of these systems.

MULTI-THREADED PROGRAMMING

What is Thread?

A thread is a flow of execution through the process code, with its own program counter that keeps track of which instruction to execute next, system registers which hold its current working variables, and a stack which contains the execution history.

A thread shares with its peer threads few information like code segment, data segment and open files. When one thread alters a code segment memory item, all other threads see that.

A thread is also called a **lightweight process**. Threads provide a way to improve application performance through parallelism. Threads represent a software approach to improving performance of operating system by reducing the overhead thread is equivalent to a classical process.

Each thread belongs to exactly one process and no thread can exist outside a process. Each thread represents a separate flow of control. Threads have been successfully used in implementing network servers and web server. They also provide a suitable foundation for parallel execution of applications on shared memory multiprocessors. The following figure shows the working of a single-threaded and a multithreaded process.

Difference between Process and Thread

S.N.	Process	Thread
1	Process is heavy weight or resource intensive.	Thread is light weight, taking lesser resources than a process.
2	Process switching needs interaction with operating system.	Thread switching does not need to interact with operating system.
3	In multiple processing environments, each process executes the same code but has its own memory and file resources.	All threads can share same set of open files, child processes.

4	If one process is blocked, then no other process can execute until the first process is unblocked.	While one thread is blocked and waiting, a second thread in the same task can run.
5	Multiple processes without using threads use more resources.	Multiple threaded processes use fewer resources.
6	In multiple processes each process operates independently of the others.	One thread can read, write or change another thread's data.

Advantages of Thread

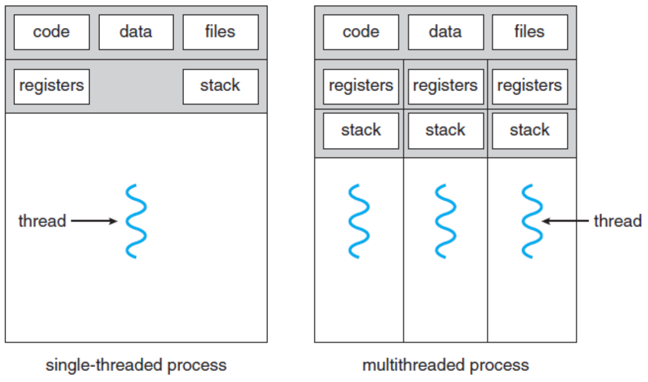
- Threads minimize the context switching time.
- Use of threads provides concurrency within a process.
- Efficient communication.
- It is more economical to create and context switch threads.
- Threads allow utilization of multiprocessor architectures to a greater scale and efficiency.

Types of Thread

Threads are implemented in following two ways −

- **User Level Threads** − User managed threads.
- **Kernel Level Threads** − Operating System managed threads acting on kernel, an operating system core.

A thread is a basic unit of CPU utilization; it comprises a thread ID, a program counter, a register set, and a stack. It shares with other threads belonging to the same process its code section, data section, and other operating-system resources, such as open files and signals.



A traditional (or ***heavyweight***) process has a single thread of control. If a process has multiple threads of control, it can perform more than one task at a time. The following Figure illustrates the difference between a traditional **single-threaded** process and a **multithreaded** process.

Motivation

Most software applications that run on modern computers are multithreaded. An application typically is implemented as a separate process with several threads of control.

A web browser might have one thread display images or text while another thread retrieves data from the network.

For example. A word processor may have a thread for displaying graphics, another thread for responding to keystrokes from the user, and a third thread for performing spelling and grammar checking in the background. Applications can also be designed to leverage processing capabilities on multicore systems. Such applications can perform several CPU-intensive tasks in parallel across the multiple computing cores.

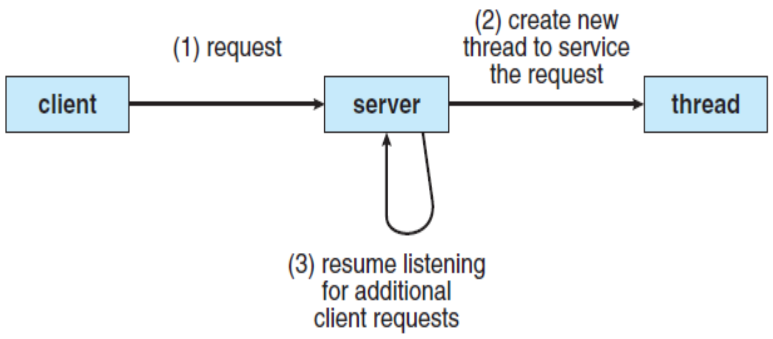

(1) request
(2) create new thread to service the request
client
server
thread
(3) resume listening for additional client requests

Figure 4.2 Multithreaded server architecture.

One solution is to have the server run as a single process that accepts requests. When the server receives a request, it creates a separate process to service that request. In fact, this process-creation method was in common use before threads became popular. Process creation is time consuming and resource intensive, however. If the new process will perform the same tasks as the existing process, why incur all that overhead? It is generally more efficient to use one process that contains multiple threads. If the web-server process is multithreaded, the server will create a separate thread that listens for client requests. When a request is made, rather than creating another process, the server creates a new thread to service the request and resume listening for additional requests. This is illustrated in Figure 4.2.

Finally, most operating-system kernels are now multithreaded. Several threads operate in the kernel, and each thread performs a specific task, such as managing devices, managing memory, or interrupt handling. For example, Solaris has a set of threads in the kernel specifically for interrupt handling; Linux uses a kernel thread for managing the amount of free memory in the system.

Benefits

The benefits of multithreaded programming can be broken down into four major categories:

- **Responsiveness**. Multithreading an interactive application may allow a program to continue running even if part of it is blocked or is performing a lengthy operation, thereby increasing responsiveness to the user.
- **Resource sharing**. Processes can only share resources through techniques such as shared memory and message passing.
- **Economy**. Allocating memory and resources for process creation is costly. Because threads share the resources of the process to which they belong, it is more economical to create and context-switch threads.
- **Scalability.** The benefits of multithreading can be even greater in a multiprocessor architecture, where threads may be running in parallel on different processing cores. A single-threaded process can run on only one processor, regardless how many are available. We explore this issue further in the following section.

Multithreading Models

Some operating system provide a combined user level thread and Kernel level thread facility.

Solaris is a good example of this combined approach. In a combined system, multiple threads within the same application can run in parallel on multiple processors and a blocking system call need not block the entire process. Multithreading models are three types

- Many to many relationship.
- Many to one relationship.
- One to one relationship.

Many to Many Model

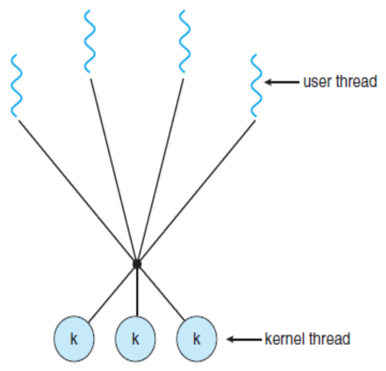


Figure 4.7 Many-to-many model.

The many-to-many model multiplexes any number of user threads onto an equal or smaller number of kernel threads.

The following diagram shows the many-to-many threading model where 6 user level threads are multiplexing with 6 kernel level threads. In this model, developers can create as many user threads as necessary and the corresponding Kernel threads can run in parallel on a multiprocessor machine. This model provides the best accuracy on concurrency and when a thread performs a blocking system call, the kernel can schedule another thread for execution.

Many to One Model

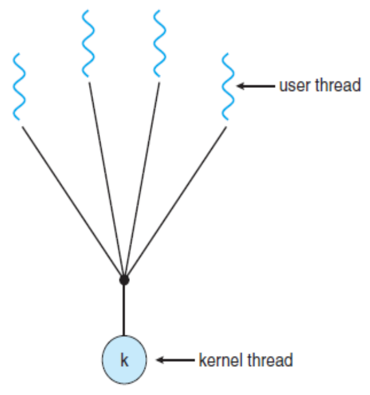


Figure 4.5 Many-to-one model.

Many-to-one model maps many user level threads to one Kernel-level thread. Thread management is done in user space by the thread library. When thread makes a blocking system call, the entire process will be blocked. Only one thread can access the Kernel at a time, so multiple threads are unable to run in parallel on multiprocessors.

If the user-level thread libraries are implemented in the operating system in such a way that the system does not support them, then the Kernel threads use the many-to-one relationship modes.

One to One Model

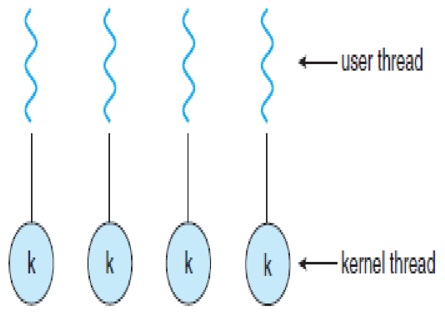


Figure 4.6 One-to-one model.

There is one-to-one relationship of user-level thread to the kernel-level thread. This model provides more concurrency than the many-to-one model. It also allows another thread to run when a thread makes a blocking system call. It supports multiple threads to execute in parallel on microprocessors.

Disadvantage of this model is that creating user thread requires the corresponding Kernel thread. OS/2, windows NT and windows 2000 use one to one relationship model.

Difference between User-Level & Kernel-Level Thread

S.N.	User-Level Threads	Kernel-Level Thread
1	User-level threads are faster to create and manage.	Kernel-level threads are slower to create and manage.
2	Implementation is by a thread library at the user level.	Operating system supports creation of Kernel threads.

3	User-level thread is generic and can run on any operating system.	Kernel-level thread is specific to the operating system.
4	Multi-threaded applications cannot take advantage of multiprocessing.	Kernel routines themselves can be multithreaded.

<u>**Threading Issues**</u>

The fork( ) and exec( ) System Calls

Recall that when fork() is called, a separate, duplicate process is created

• How should fork() behave in a multithreaded program? - Should all threads be duplicated?

- Should only the thread that made the call to fork() be duplicated?

• In some systems, different versions of fork() exist depending on the desired behavior

- Some UNIX systems have fork1() and forkall() • fork1() only duplicates the calling thread

• forkall() duplicates all of the threads in a process

- In a POSIX-compliant system, fork() behaves the same as fork1()

• The exec() system call continues to behave as expected - Replaces the entire process that called it, including all threads

• If planning to call exec() after fork(), then there is no need to duplicate all of the threads in the calling process - All threads in the child process will be terminated when exec() is called

- Use fork1(), rather than forkall() if using in conjunction with exec()

Signal Handling

Signals are used in UNIX systems to notify a process that a particular event has occurred

- CTRL-C is an example of an asynchronous signal that might be sent to a process

• An asynchronous signal is one that is generated from outside the process that receives it

- Divide by 0 is an example of a synchronous signal that might be sent to a process

• A synchronous signal is delivered to the same process that caused the signal to occur

• All signals follow the same basic pattern: - A signal is generated by particular event

- The signal is delivered to a process

- The signal is handled by a signal handler (all signals are handled exactly once)

Signal handling is straightforward in a single-threaded process - The one (and only) thread in the process receives and handles the signal

In a multithreaded program, where should signals be delivered? - Options:

(1) Deliver the signal to the thread to which the signal applies

(2) Deliver the signal to every thread in the process

(3) Deliver the signal only to certain threads in the process

(4) Assign a specific thread to receive all signals for the process

• Option 1 - Deliver the signal to the thread to which the signal applies - Most likely option when handling synchronous signals (e.g. only the thread that attempts to divide by zero needs to know of the error)

• Option 2 - Deliver the signal to every thread in the process - Likely to be used in the event that the process is being terminated (e.g. a CTRLC is sent to terminate the process, all threads need to receive this signal and terminate)

Thread Cancellation

• Thread cancellation is the act of terminating a thread before it has completed - Example - clicking the stop button on your web browser will stop the thread that is rendering the web page

• The thread to be cancelled is called the target thread • Threads can be cancelled in a couple of ways - Asynchronous cancellation terminates the target thread immediately

• Thread may be in the middle of writing data ... not so good

- Deferred cancellation allows the target thread to periodically check if it should be cancelled

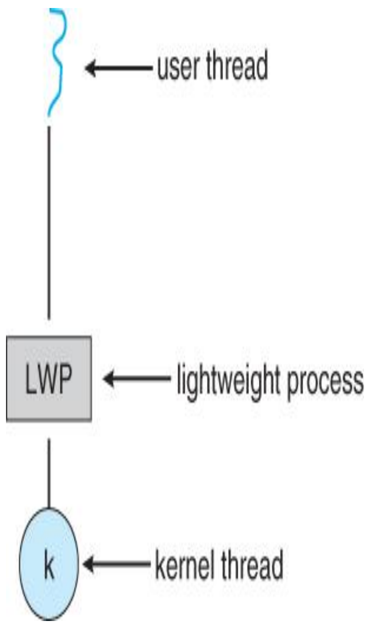
• Allows thread to terminate itself in an orderly fashion

- Threads that are no longer needed may be cancelled by another thread in one of two ways:
    1. **Asynchronous Cancellation** cancels the thread immediately.
    2. **Deferred Cancellation** sets a flag indicating the thread should cancel itself when it is convenient. It is then up to the cancelled thread to check this flag periodically and exit nicely when it sees the flag set.
- (Shared) resource allocation and inter-thread data transfers can be problematic with asynchronous cancellation.

Thread-Local Storage

- Most data is shared among threads, and this is one of the major benefits of using threads in the first place.
- However sometimes threads need thread-specific data also.
- Most major thread libraries (pThreads, Win32, Java) provide support for thread-specific data, known as **thread-local storage** or **TLS.** Note that this is more like static data than local variables, because it does not cease to exist when the function ends.

Scheduler Activations



- Many implementations of threads provide a virtual processor as an interface between the user thread and the kernel thread, particularly for the many-to-many or two-tier models.
- This virtual processor is known as a "Lightweight Process", LWP.
    - There is a one-to-one correspondence between LWPs and kernel threads.
    - The number of kernel threads available, (and hence the number of LWPs) may change dynamically.
    - The application (user level thread library) maps user threads onto available LWPs.
    - kernel threads are scheduled onto the real processor(s) by the OS.
    - The kernel communicates to the user-level thread library when certain events occur (such as a thread about to block) via an **upcall**, which is handled in the thread library by an **upcall handler**. The upcall also provides a new LWP for the upcall handler to run on, which it can then use to reschedule the user thread that is about to become blocked. The OS will also issue upcalls when a thread becomes unblocked, so the thread library can make appropriate adjustments.
- If the kernel thread blocks, then the LWP blocks, which blocks the user thread.
- Ideally there should be at least as many LWPs available as there could be concurrently blocked kernel threads. Otherwise if all LWPs are blocked, then user threads will have to wait for one to become available.

PROCESS SCHEDULING

CPU Scheduling

CPU scheduling is the basis of multiprogrammed operating systems. By switching the CPU among processes, the operating system can make the computer more productive. In this chapter, we introduce basic CPU-scheduling concepts and present several CPU- scheduling algorithms. We also consider the problem of selecting an algorithm for a particular system.

Basic Concepts

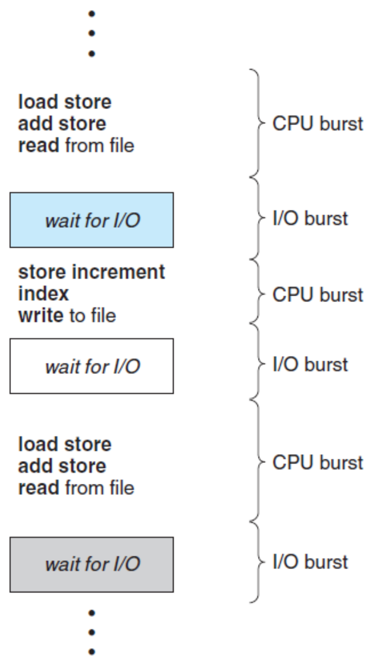


Figure 6.1 Alternating sequence of CPU and I/O bursts.

In a single-processor system, only one process can run at a time. Others must wait until the CPU is free and can be rescheduled. The objective of multiprogramming is to have some process running at all times, to maximize CPU utilization. The idea is relatively simple. A process is executed until it must wait, typically for the completion of some I/O request. In a simple computer system, the CPU then just sits idle. All this waiting time is wasted; no useful work is accomplished. With multiprogramming, we try to use this time productively. Several processes are kept in memory at one time.

When one process has to wait, the operating system takes the CPU away from that process and gives the CPU to another process. This pattern continues. Every time one process has to wait, another process can take over use of the CPU.

Scheduling of this kind is a fundamental operating-system function. Almost all computer resources are scheduled before use. The CPU is, of course, one of the primary computer resources. Thus, its scheduling is central to operating-system design.

CPU – I/O Burst Cycle

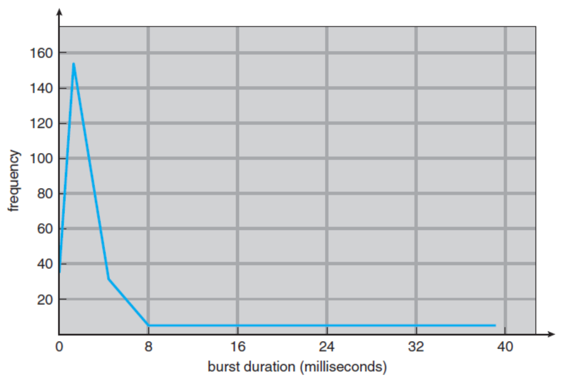


Figure 6.2 Histogram of CPU-burst durations.

The success of CPU scheduling depends on an observed property of processes: process execution consists of a **cycle** of CPU execution and I/O wait. Processes alternate between these two states. Process execution begins with a **CPU burst**. That is followed by an **I/O burst**, which is followed by another CPU burst, then another I/O burst, and so on. Eventually, the final CPU burst ends with a system request to terminate execution (Figure 6.1).

The durations of CPU bursts have been measured extensively. Although they vary greatly from process to process and from computer to computer, they tend to have a frequency curve similar to that shown in Figure 6.2. The curve is generally characterized as exponential or hyper exponential, with a large number of short CPU bursts and a small number of long CPU bursts.

An I/O-bound program typically has many short CPU bursts. A CPU-bound program might have a few long CPU bursts. This distribution can be important in the selection of an appropriate CPU-scheduling algorithm.

CPU Scheduler

Whenever the CPU becomes idle, the operating system must select one of the processes in the ready queue to be executed. The selection process is carried out by the **short-term scheduler**, or CPU scheduler. The scheduler selects a process from the processes in memory that are ready to execute and allocates the CPU to that process.

Note that the ready queue is not necessarily a first-in, first-out (FIFO) queue. As we shall see when we consider the various scheduling algorithms, a ready queue can be implemented as a FIFO queue, a priority queue, a tree, or simply an unordered linked list. Conceptually, however, all the processes in the ready queue are lined up waiting for a chance to run on the CPU. The records in the queues are generally process control blocks (PCBs) of the processes.

Preemptive Scheduling

CPU-scheduling decisions may take place under the following four circum-stances:

When a process switches from the running state to the waiting state (for example, as the result of an I/O request or an invocation of wait() for the termination of a child process)

When a process switches from the running state to the ready state (for example, when an interrupt occurs)

When a process switches from the waiting state to the ready state (for example, at completion of I/O)

When a process terminates

For situations 1 and 4, there is no choice in terms of scheduling. A new process (if one exists in the ready queue) must be selected for execution. There is a choice, however, for situations 2 and 3.

When scheduling takes place only under circumstances 1 and 4, we say that the scheduling scheme is **nonpreemptive** or **cooperative**. Otherwise, it is **preemptive**. Under nonpreemptive scheduling, once the

CPU has been allocated to a process, the process keeps the CPU until it releases the CPU either by terminating or by switching to the waiting state. This scheduling method was used by Microsoft Windows 3.x. Windows 95 introduced preemptive scheduling, and all subsequent versions of Windows operating systems have used preemptive scheduling. The Mac OS X operating system for the Macintosh also uses preemptive scheduling; previous versions of the Macintosh operating system relied on cooperative scheduling. Cooperative scheduling is the only method that can be used on certain hardware platforms, because it does not require the special hardware (for example, a timer) needed for preemptive scheduling.

Another component involved in the CPU-scheduling function is the **dispatcher**. The dispatcher is the module that gives control of the CPU to the process selected by the short-term scheduler. This function involves the following:

Switching context

Switching to user mode

Jumping to the proper location in the user program to restart that program

The dispatcher should be as fast as possible, since it is invoked during every process switch. The time it takes for the dispatcher to stop one process and start another running is known as the **dispatch latency**.

Scheduling Criteria

Different CPU-scheduling algorithms have different properties, and the choice of a particular algorithm may favor one class of processes over another. In choosing which algorithm to use in a particular situation, we must consider the properties of the various algorithms.

Many criteria have been suggested for comparing CPU-scheduling algorithms. Which characteristics are used for comparison can make a substantial difference in which algorithm is judged to be best. The criteria include the following:

CPU utilization. We want to keep the CPU as busy as possible. Conceptually, CPU utilization can range from 0 to 100 percent. In a real system, it should range from 40 percent (for a lightly loaded system) to 90 percent (for a heavily loaded system).

Throughput. If the CPU is busy executing processes, then work is being done. One measure of work is the number of processes that are completed per time unit, called **throughput**. For long processes, this rate may be one process per hour; for short transactions, it may be ten processes per second.

Turnaround time. From the point of view of a particular process, the important criterion is how long it takes to execute that process. The interval from the time of submission of a process to the time of completion is the turnaround time. Turnaround time is the sum of the periods spent waiting to get into memory, waiting in the ready queue, executing on the CPU, and doing I/O.

Waiting time. The CPU-scheduling algorithm does not affect the amount of time during which a process executes or does I/O. It affects only the amount of time that a process spends waiting in the ready queue. Waiting time is the sum of the periods spent waiting in the ready queue.

Response time. In an interactive system, turnaround time may not be the best criterion. Often, a process can produce some output fairly early and can continue computing new results while previous results are being output to the user. Thus, another measure is the time from the submission of a request until the first response is produced. This measure, called response time, is the time it takes to start responding, not the time it takes to output the response. The turnaround time is generally limited by the speed of the output device.

CPU-SCHEDULING ALGORITHMS

CPU scheduling deals with the problem of deciding which of the processes in the ready queue is to be allocated the CPU. There are many different CPU-scheduling algorithms. In this section, we describe several of them.

First-Come, First-Served Scheduling

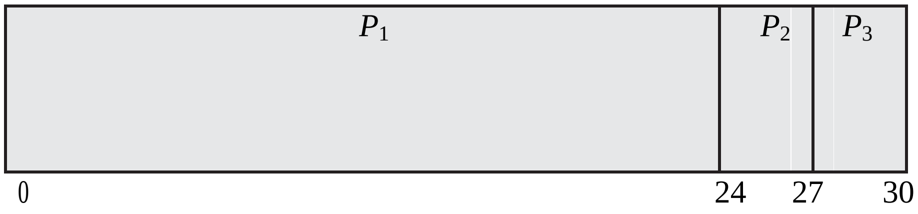
By far the simplest CPU-scheduling algorithm is the **first-come, first-served (FCFS)** scheduling algorithm. With this scheme, the process that requests the CPU first is allocated the CPU first. The implementation of the FCFS policy is easily managed with a FIFO queue. When a process enters the ready queue, its PCB is linked onto the tail of the queue. When the CPU is free, it is allocated to the process at the head of the queue. The running process is then removed from the queue. The code for FCFS scheduling is simple to write and understand.

On the negative side, the average waiting time under the FCFS policy is often quite long. Consider the following set of processes that arrive at time 0, with the length of the CPU burst given in milliseconds:

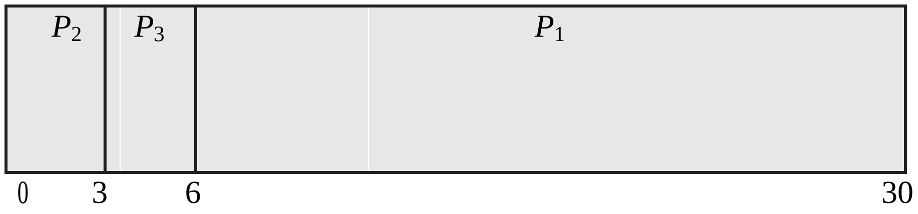
Process	Burst Time
P_1	24

P_2	3
P_3	3

If the processes arrive in the order P_1, P_2, P_3, and are served in FCFS order, we get the result shown in the following **Gantt chart**, which is a bar chart that illustrates a particular schedule, including the start and finish times of each of the participating processes:



The waiting time is 0 milliseconds for process P_1 , 24 milliseconds for process P_2 , and 27 milliseconds for process P_3. Thus, the average waiting time is (0+24 + 27)/3 = 17 milliseconds. If the processes arrive in the order P_2, P_3 , P_1, however, the results will be as shown in the following Gantt chart:



The average waiting time is now (6 + 0 + 3)/3 = 3 milliseconds. This reduction is substantial. Thus, the average waiting time under an FCFS policy is generally not minimal and may vary substantially if the processes' CPU burst times vary greatly.

Shortest-Job-First Scheduling

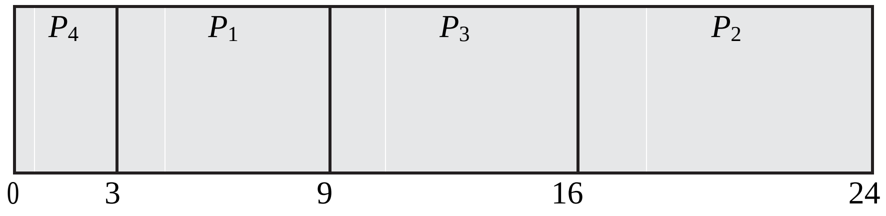
A different approach to CPU scheduling is the **shortest-job-first (SJF)** scheduling algorithm. This algorithm associates with each process the length of the process's next CPU burst. When the CPU is available, it is assigned to the process that has the smallest next CPU burst. If the next CPU bursts of two processes are the same, FCFS scheduling is used to break the tie. Note that a more appropriate term for this scheduling method would be the ***shortest-next-CPU-burst*** algorithm, because scheduling depends on the length of the next CPU burst of a process, rather than its total length. We use the term SJF because most people and textbooks use this term to refer to this type of scheduling.

As an example of SJF scheduling, consider the following set of processes, with the length of the CPU burst given in milliseconds:

Process	Burst Time
P_1	6

P_2	8
P_3	7
P_4	3

Using SJF scheduling, we would schedule these processes according to the following Gantt chart:



The waiting time is 3 milliseconds for process P_1, 16 milliseconds for process P_2, 9 milliseconds for process P_3, and 0 milliseconds for process P_4. Thus, the average waiting time is (3 + 16 + 9 + 0)/4 = 7 milliseconds. By comparison, if we were using the FCFS scheduling scheme, the average waiting time would be 10.25 milliseconds.

Priority Scheduling

The SJF algorithm is a special case of the general **priority-scheduling** algorithm. A priority is associated with each process, and the CPU is allocated to the process with the highest priority. Equal-priority processes are scheduled in FCFS order. An SJF algorithm is simply a priority algorithm where the priority (p) is the inverse of the (predicted) next CPU burst. The larger the CPU burst, the lower the priority, and vice versa.

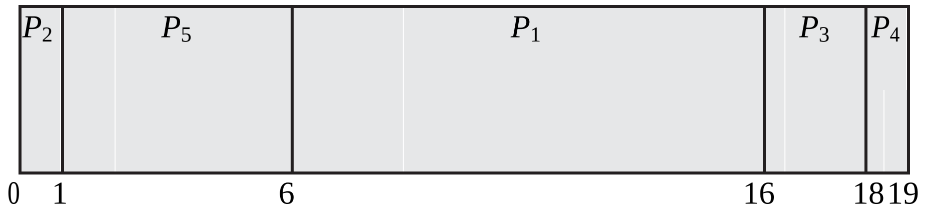
Note that we discuss scheduling in terms of ***high*** priority and ***low*** priority. Priorities are generally indicated by some fixed range of numbers, such as 0 to 7 or 0 to 4,095. However, there is no general agreement on whether 0 is the highest or lowest priority. Some systems use low numbers to represent low priority; others use low numbers for high priority. This difference can lead to confusion. In this text, we assume that low numbers represent high priority.

As an example, consider the following set of processes, assumed to have arrived at time 0 in the order P_1 , P_2, · · ·, P_5 , with the length of the CPU burst given in milliseconds:

Process	Burst Time	Priority
$P1$	10	3
P_2	1	1
P_3	2	4
P_4	1	5

$P5$	5	2

Using priority scheduling, we would schedule these processes according to the following Gantt chart:



The average waiting time is 8.2 milliseconds.

Priorities can be defined either internally or externally.

Internally defined priorities use some measurable quantity or quantities to compute the priority of a process. For example, time limits, memory requirements, the number of open files, and the ratio of average I/O burst to average CPU burst have been used in computing priorities.

External priorities are set by criteria outside the operating system, such as the importance of the process, the type and amount of funds being paid for computer use, the department sponsoring the work, and other, often political, factors.

Round-Robin Scheduling

The **round-robin (RR)** scheduling algorithm is designed especially for time-sharing systems. It is similar to FCFS scheduling, but preemption is added to enable the system to switch between processes. A small unit of time, called a **time quantum** or **time slice**, is defined. A time quantum is generally from 10 to 100 milliseconds in length. The ready queue is treated as a circular queue.

The CPU scheduler goes around the ready queue, allocating the CPU to each process for a time interval of up to 1 time quantum.

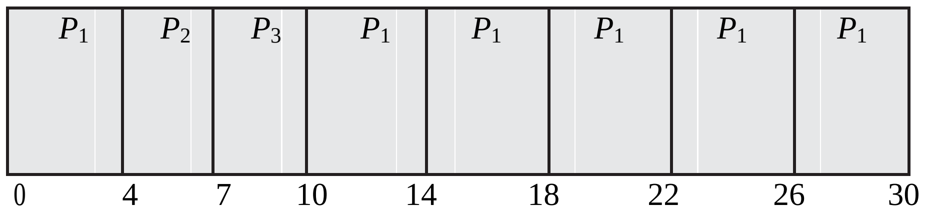
To implement RR scheduling, we again treat the ready queue as a FIFO queue of processes. New processes are added to the tail of the ready queue. The CPU scheduler picks the first process from the ready queue, sets a timer to interrupt after 1 time quantum, and dispatches the process.

One of two things will then happen. The process may have a CPU burst of less than 1 time quantum. In this case, the process itself will release the CPU voluntarily. The scheduler will then proceed to the next process in the ready queue. If the CPU burst of the currently running process is longer than 1 time quantum, the timer will go off and will cause an interrupt to the operating system. A context switch will be executed, and the process will be put at the tail of the ready queue. The CPU scheduler will then select the next process in the ready queue.

The average waiting time under the RR policy is often long. Consider the following set of processes that arrive at time 0, with the length of the CPU burst given in milliseconds:

Process	Burst Time
P_1	24
P_2	3
P_3	3

If we use a time quantum of 4 milliseconds, then process P_1 gets the first 4 milliseconds. Since it requires another 20 milliseconds, it is preempted after the first time quantum, and the CPU is given to the next process in the queue, process P_2. Process P_2 does not need 4 milliseconds, so it quits before its time quantum expires. The CPU is then given to the next process, process P_3. Once each process has received 1 time quantum, the CPU is returned to process P_1 for an additional time quantum. The resulting RR schedule is as follows:



Let's calculate the average waiting time for this schedule. P_1 waits for 6 milliseconds (10 - 4), P_2 waits for 4 milliseconds, and P_3 waits for 7 milliseconds. Thus, the average waiting time is 17/3 = 5.66 milliseconds.

In the RR scheduling algorithm, no process is allocated the CPU for more than 1 time quantum in a row (unless it is the only runnable process). If a process's CPU burst exceeds 1 time quantum, that process is preempted and is put back in the ready queue. The RR scheduling algorithm is thus preemptive.

If there are n processes in the ready queue and the time quantum is q, then each process gets $1/n$ of the CPU time in chunks of at most q time units. Each process must wait no longer than $(n - 1) \times q$ time units until its next time quantum. For example, with five processes and a time quantum of 20 milliseconds, each process will get up to 20 milliseconds every 100 milliseconds.

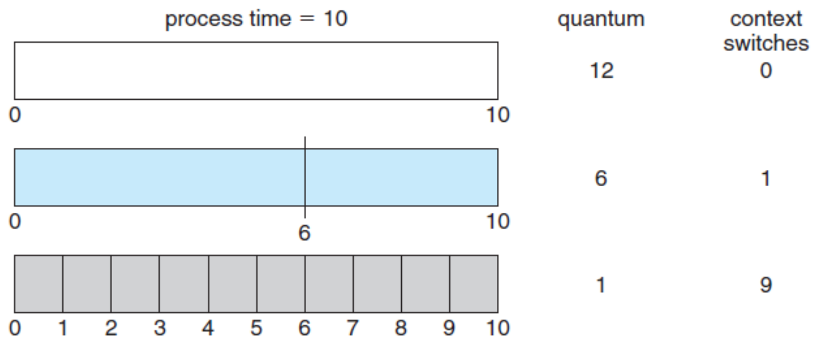


Figure 6.4 How a smaller time quantum increases context switches.

The performance of the RR algorithm depends heavily on the size of the time quantum. At one extreme, if the time quantum is extremely large, the RR policy is the same as the FCFS policy. In contrast, if the time quantum is extremely small (say, 1 millisecond), the RR approach can result in a large number of context switches. Assume, for example, that we have only one process of 10 time units. If the quantum is 12 time units, the process finishes in less than 1 time quantum, with no overhead. If the quantum is 6 time units, however, the process requires 2 quanta, resulting in a context switch. If the time quantum is 1 time unit, then nine context switches will occur, slowing the execution of the process accordingly (Figure 6.4).

Multiple-Level Queues Scheduling

Multiple-level queues are not an independent scheduling algorithm. They make use of other existing algorithms to group and schedule jobs with common characteristics.

- Multiple queues are maintained for processes with common characteristics.
- Each queue can have its own scheduling algorithms.
- Priorities are assigned to each queue.

For example, CPU-bound jobs can be scheduled in one queue and all I/O-bound jobs in another queue. The Process Scheduler then alternately selects jobs from each queue and assigns them to the CPU based on the algorithm assigned to the queue.

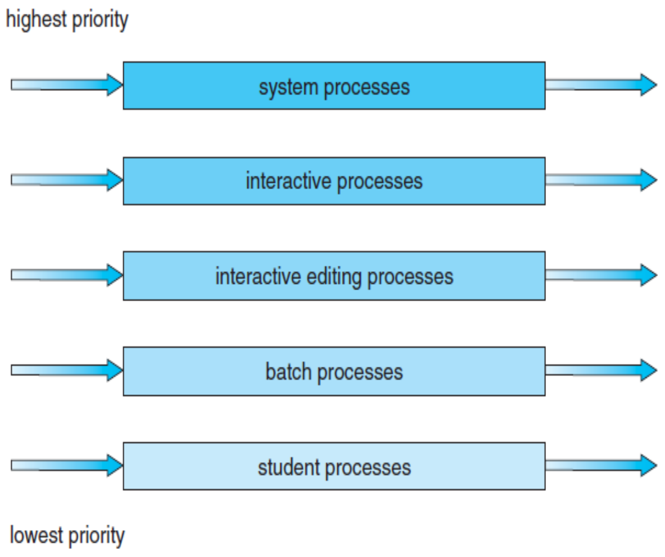


Figure 6.6 Multilevel queue scheduling.

Let’s look at an example of a multilevel queue scheduling algorithm with five queues, listed below in order of priority:

System processes
Interactive processes
Interactive editing processes
Batch processes
Student processes

Each queue has absolute priority over lower-priority queues. No process in the batch queue, for example, could run unless the queues for system processes, interactive processes, and interactive editing processes were all empty. If an interactive editing process entered the ready queue while a batch process was running, the batch process would be preempted.

Multilevel Feedback Queue Scheduling

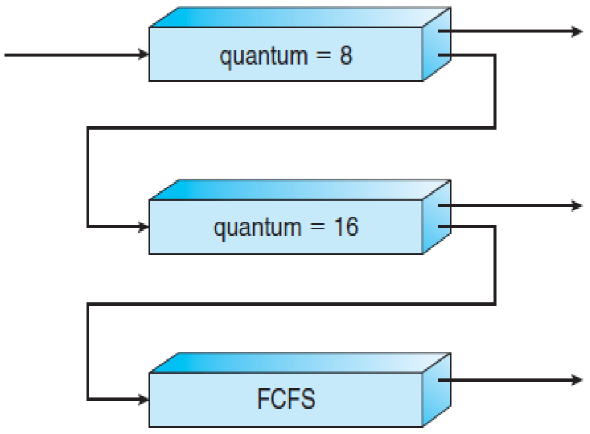


Figure 6.7 Multilevel feedback queues.

Normally, when the multilevel queue scheduling algorithm is used, processes are permanently assigned to a queue when they enter the system. If there are separate queues for foreground and background processes, for example, processes do not move from one queue to the other, since processes do not change their foreground or background nature. This setup has the advantage of low scheduling overhead, but it is inflexible.

The **multilevel feedback queue** scheduling algorithm, in contrast, allows a process to move between queues. The idea is to separate processes according to the characteristics of their CPU bursts. If a process uses too much CPU time, it will be moved to a lower- priority queue. This scheme leaves I/O-bound and interactive processes in the higher-priority queues. In addition, a process that waits too long in a lower-priority queue may be moved to a higher-priority queue. This form of aging prevents starvation.

For example, consider a multilevel feedback queue scheduler with three queues, numbered from 0 to 2 (Figure 6.7). The scheduler first executes all processes in queue 0. Only when queue 0 is empty will it execute processes in queue 1. Similarly, processes in queue 2 will be executed only if queues 0 and 1 are empty. A process that arrives for queue 1 will preempt a process in queue 2. A process in queue 1 will in turn be preempted by a process arriving for queue 0.

A process entering the ready queue is put in queue 0. A process in queue 0 is given a time quantum of 8 milliseconds. If it does not finish within this time, it is moved to the tail of queue 1. If queue 0 is empty, the process at the head of queue 1 is given a quantum of 16 milliseconds. If it does not complete, it is preempted and is put into queue 2. Processes in queue 2 are run on an FCFS basis but are run only when queues 0 and 1 are empty.

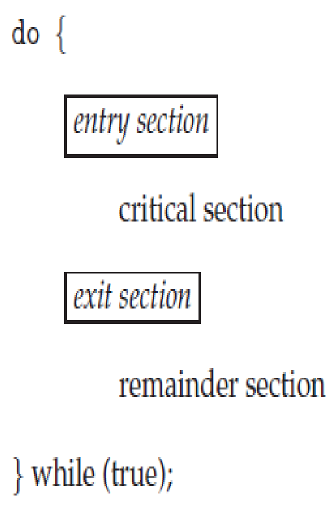
PROCESS SYNCHRONIZATION

A **cooperating process** is one that can affect or be affected by other processes executing in the system. Cooperating processes can either directly share a logical address space (that is, both code and data) or be allowed to share data only through files or messages. The former case is achieved through the use of threads,

The Critical-Section Problem

We begin our consideration of process synchronization by discussing the so-called critical-section problem. Consider a system consisting of *n* processes $\{P_0, P_1, ..., P_{n-1}\}$. Each process has a segment of code, called a **critical section**, in which the process may be changing common variables, updating a table, writing a file, and soon.

The important feature of the system is that, when one process is executing in its critical section, no other process is allowed to execute in its critical section. That is, no two processes are executing in their critical sections at the same time.



```
do {
     entry section
          critical section
     exit section
          remainder section
} while (true);
```

Figure 5.1 General structure of a typical process P_i.

The ***critical-section problem*** is to design a protocol that the processes can use to cooperate. Each process must request permission to enter its critical section.

The section of code implementing this request is the **entry section**.

The critical section may be followed by an **exit section**.

The remaining code is the **remainder section**.

The general structure of a typical process P_i is shown in Figure 5.1. The entry section and exit section are enclosed in boxes to highlight these important segments of code.

A solution to the critical-section problem must satisfy the following three requirements:

Mutual exclusion. If process P_i is executing in its critical section, then no other processes can be executing in their critical sections.

Progress. If no process is executing in its critical section and some processes wish to enter their critical sections, then only those processes that are not executing in their remainder sections can participate in deciding which will enter its critical section next, and this selection cannot be postponed indefinitely.

Bounded waiting. There exists a bound, or limit, on the number of times that other processes are allowed to enter their critical sections after a process has made a request to enter its critical section and before that request is granted.

We assume that each process is executing at a nonzero speed. However, we can make no assumption concerning the relative speed of the *n* processes.

At a given point in time, many kernel-mode processes may be active in the operating system. As a result, the code implementing an operating system (***kernel code***) is subject to several possible race conditions. Consider as an example a kernel data structure that maintains a list of all open files in the system. This list must be modified when a new file is opened or closed (adding the file to the list or removing it from the list) . If two processes were to open files simultaneously, the separate updates to this list could result in a race condition. Other kernel data structures that are prone to possible race conditions include structures for maintaining memory allocation, for maintaining process lists, and for interrupt handling. It is up to kernel developers to ensure that the operating system is free from such race conditions.

Two general approaches are used to handle critical sections in operating systems:

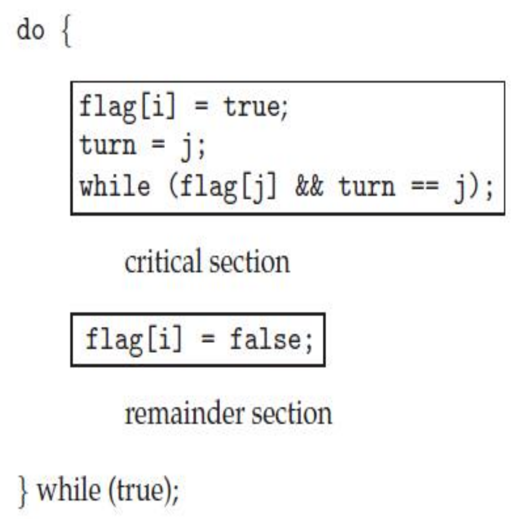
preemptive kernels. A preemptive kernel allows a process to be preempted while it is running in kernel mode.

A **nonpreemptive kernel** does not allow a process running in kernel mode to be preempted; a kernel- mode process will run until it exits kernel mode, blocks, or voluntarily yields control of the CPU.

Obviously, a nonpreemptive kernel is essentially free from race conditions on kernel data structures, as only one process is active in the kernel at a time. We cannot say the same about preemptive kernels, so they must be carefully designed to ensure that shared kernel data are free from race conditions. Preemptive kernels are especially difficult to design for SMP architectures, since in these environments it is possible for two kernel-mode processes to run simultaneously on different processors.

Peterson's Solution

The classic software-based solution to the critical-section problem known as **Peterson's solution**. Because of the way modern computer architectures perform basic machine-language instructions, such as load and store, there are no guarantees that Peterson's solution will work correctly on such architectures. However, we present the solution because it provides a good algorithmic description of solving the critical-section problem and illustrates some of the complexities involved in designing software that addresses the requirements of mutual exclusion, progress, and bounded waiting



```
do {

    flag[i] = true;
    turn = j;
    while (flag[j] && turn == j);

        critical section

    flag[i] = false;

        remainder section

} while (true);
```

Figure 5.2 The structure of process P_i in Peterson's solution.

Peterson's solution is restricted to two processes that alternate execution between their critical sections and remainder sections. The processes are numbered P_0 and P_1. For convenience, when presenting P_i , we use P_j to denote the other process; that is, j equals 1 − i.

Peterson's solution requires the two processes to share two data items:

int turn;

boolean flag[2];

The variable turn indicates whose turn it is to enter its critical section. That is, if turn == i, then process P_i is allowed to execute in its critical section. The flag array is used to indicate if a process is ready to enter its critical section. For example, if flag[i] is true, this value indicates that P_i is ready to enter its critical section. With an explanation of these data structures complete, we are now ready to describe the algorithm shown in Figure 5.2.

To enter the critical section, process P_i first sets flag[i] to be true and then sets turn to the value j , thereby asserting that if the other process wishes to enter the critical section, it can do so. If both processes try to enter at the same time, turn will be set to both i and j at roughly the same time. Only one of these assignments will last; the other will occur but will be overwritten immediately. The eventual value of turn determines which of the two processes is allowed to enter its critical section first.

We now prove that this solution is correct. We need to show that:

Mutual exclusion is preserved.

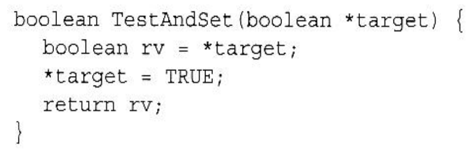
The progress requirement is satisfied.

The bounded-waiting requirement is met.

To prove property 1, we note that each P_i enters its critical section only if either flag[j] == false or turn == i. Also note that, if both processes can be executing in their critical sections at the same time, then flag[0] == flag[1] == true. These two observations imply that P_0 and P_1 could not have successfully executed their while statements at about the same time, since the value of turn can be either 0 or 1 but cannot be both. Hence, one of the processes — say, P_j — must have successfully executed the while statement, whereas P_i had to execute at least one additional statement ("turn == j"). However, at that time, flag[j] == true and turn == j, and this condition will persist as long as P_j is in its critical section; as a result, mutual exclusion is preserved.

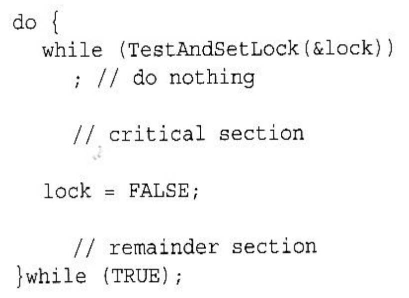
Synchronization Hardware

- To generalize the solution(s) expressed above, each process when entering their critical section must set some sort of ***lock***, to prevent other processes from entering their critical sections simultaneously, and must release the lock when exiting their critical section, to allow other processes to proceed. Obviously it must be possible to attain the lock only when no other process has already set a lock. Specific implementations of this general procedure can get quite complicated, and may include hardware solutions as outlined in this section.



```
boolean TestAndSet(boolean *target) {
  boolean rv = *target;
  *target = TRUE;
  return rv;
}
```

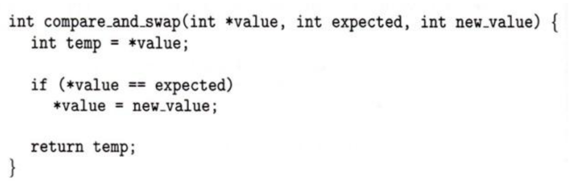
Figure 5.3 The definition of the TestAndSet() instruction.



```
do {
  while (TestAndSetLock(&lock))
    ; // do nothing

    // critical section

  lock = FALSE;

    // remainder section
}while (TRUE);
```

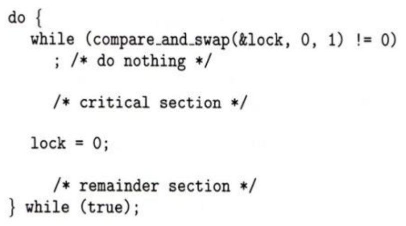
Figure 5.4 Mutual-exclusion implementation with TestAndSet().

- One simple solution to the critical section problem is to simply prevent a process from being interrupted while in their critical section, which is the approach taken by non preemptive kernels. Unfortunately this does not work well in multiprocessor environments, due to the difficulties in disabling and the re-enabling interrupts on all processors. There is also a question as to how this approach affects timing if the clock interrupt is disabled.
- Another approach is for hardware to provide certain ***atomic*** operations. These operations are guaranteed to operate as a single instruction, without interruption. One such operation is the "Test and Set", which simultaneously sets a boolean lock variable and returns its previous



```
int compare_and_swap(int *value, int expected, int new_value) {
  int temp = *value;

  if (*value == expected)
    *value = new_value;

  return temp;
}
```

Figure 5.5 The definition of the compare_and_swap() instruction.



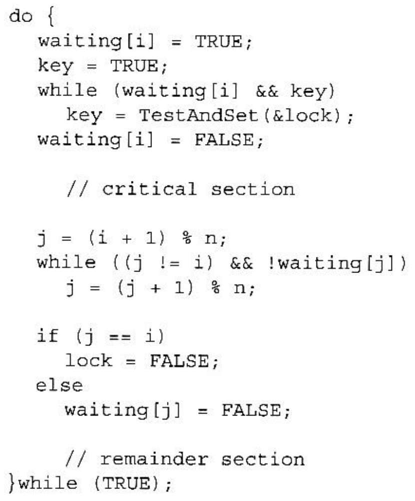
```
do {
  while (compare_and_swap(&lock, 0, 1) != 0)
    ; /* do nothing */

    /* critical section */

  lock = 0;

    /* remainder section */
} while (true);
```

Figure 5.6 Mutual-exclusion implementation with the compare_and_swap() instruction.

value, as shown in Figures 5.3 and 5.4:

Another variation on the test-and-set is an atomic swap of two booleans, as shown in Figures 5.5 and 5.6:

- The above examples satisfy the mutual exclusion requirement, but unfortunately do not guarantee bounded waiting. If there are multiple processes trying to get into their critical sections, there is no guarantee of what order they will enter, and any one process could have the bad luck to wait forever until they got their turn in the critical section. (Since there is no guarantee as to the relative ***rates*** of the processes, a very fast process could theoretically release the lock, whip through their remainder section, and re-lock the lock before a slower process got a chance. As more and more processes are involved vying for the same resource, the odds of a slow process getting locked out completely increase.)



```
do {
   waiting[i] = TRUE;
   key = TRUE;
   while (waiting[i] && key)
      key = TestAndSet(&lock);
   waiting[i] = FALSE;

      // critical section

   j = (i + 1) % n;
   while ((j != i) && !waiting[j])
      j = (j + 1) % n;

   if (j == i)
      lock = FALSE;
   else
      waiting[j] = FALSE;

      // remainder section
}while (TRUE);
```

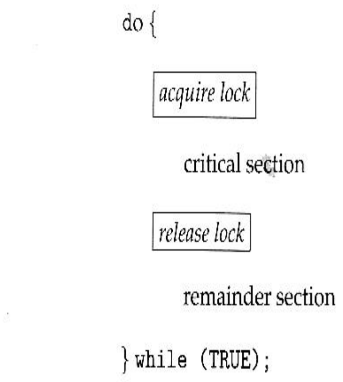
- Figure 5.7 illustrates a solution using test-and-set that does satisfy this requirement, using two shared data structures, `boolean lock` and `boolean waiting[ N ]`, where N is the number of processes in contention for critical sections:

The key feature of the above algorithm is that a process blocks on the AND of the critical section being locked and that this process is in the waiting state. When exiting a critical section, the exiting process does not just unlock the critical section and let the other processes have a free-for-all trying to get in. Rather it first looks in an orderly progression (starting with the next process on the list) for a process that has been waiting, and if it finds one, then it releases that particular process from its waiting state, without unlocking the critical section, thereby allowing a specific process into the critical section while continuing to block all the others. Only if there are no other processes currently waiting is the general lock removed, allowing the next process to come along access to the critical section.

- Unfortunately, hardware level locks are especially difficult to implement in multi-processor architectures. Discussion of such issues is left to books on advanced computer architecture.

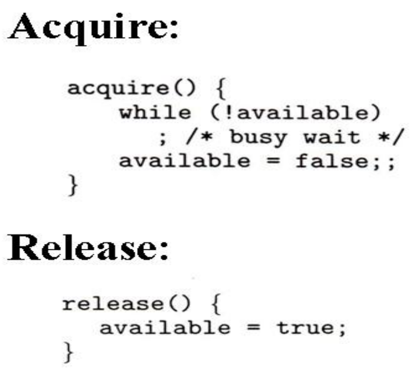
Mutex Locks :

- The hardware solutions presented above are often difficult for ordinary programmers to access, particularly on multi-processor machines, and particularly because they are often platform-dependent.
- Therefore most systems offer a software API equivalent called ***mutex locks*** or simply ***mutexes.*** (For mutual exclusion)
- The terminology when using mutexes is to ***acquire*** a lock prior to entering a critical section, and to ***release*** **it when exiting, as shown in Figure**



```
do {
    acquire lock
        critical section
    release lock
        remainder section
} while (TRUE);
```

- Just as with hardware locks, the acquire step will block the process if the lock is in use by another process, and both the acquire and release operations are atomic.
- Acquire and release can be implemented as shown here, based on a boolean variable "available":



Acquire:

```
acquire() {
    while (!available)
       ; /* busy wait */
    available = false;;
}
```

Release:

```
release() {
   available = true;
}
```

- One problem with the implementation shown here, (and in the hardware solutions presented earlier), is the busy loop used to block processes in the acquire phase. These types of locks are referred to as ***spinlocks***, because the CPU just sits and spins while blocking the process.
- Spinlocks are wasteful of cpu cycles, and are a really bad idea on single-cpu single-threaded machines, because the spinlock blocks the entire computer, and doesn't allow any other process to release the lock. (Until the scheduler kicks the spinning process off of the cpu.)
- On the other hand, spinlocks do not incur the overhead of a context switch, so they are effectively used on multi-threaded machines when it is expected that the lock will be released after a short time.

Semaphores

A **semaphore** S is an integer variable that, apart from initialization, is accessed only through two standard atomic operations: wait() and signal(). The wait() operation was originally termed P (from the Dutch ***proberen,*** "to test"); signal() was originally called V (from ***verhogen,*** "to increment"). The definition of wait() is as follows:

```
wait(S) {
    while (S <= 0)
   // busy wait
   S--;
  }
```

The definition of signal() is as follows:

```
signal(S) {
        S++;
   }
```

All modifications to the integer value of the semaphore in the wait() and signal() operations must be executed indivisibly. That is, when one process modifies the semaphore value, no other process can simultaneously modify that same semaphore value. In addition, in the case of wait(S), the testing of the integer value of S ($S \leq 0$), as well as its possible modification (S--), must be executed without interruption. We shall see how these operations can be implemented in Section 5.6.2. First, let's see how semaphores can be used.

Semaphore Usage

Operating systems often distinguish between counting and binary semaphores. The value of a **counting semaphore** can range over an unrestricted domain. The value of a **binary semaphore** can range only between 0 and 1. Thus, binary semaphores behave similarly to mutex locks. In fact, on systems that do not provide mutex locks, binary semaphores can be used instead for providing mutual exclusion.

Counting semaphores can be used to control access to a given resource consisting of a finite number of instances. The semaphore is initialized to the number of resources available. Each process that wishes to use a resource performs a wait() operation on the semaphore (thereby decrementing the count). When a process releases a resource, it performs a signal() operation (incrementing the count). When the count for the semaphore goes to 0, all resources are being used. After that, processes that wish to use a resource will block until the count becomes greater than 0.

We can also use semaphores to solve various synchronization problems. For example, consider two concurrently running processes: P_1 with a statement S_1 and P_2 with a statement S_2 . Suppose we require that S_2 be executed only after S_1 has completed. We can implement this scheme readily by letting P_1 and P_2 share a common semaphore synch, initialized to 0. In process P_1, we insert the statements

```
S1 ;
signal(synch);
```

In process P_2 , we insert the statements

```
wait(synch);
S2 ;
```

Because synch is initialized to 0, P_2 will execute S_2 only after P_1 has invoked signal(synch), which is after statement S_1 has been executed.

Semaphore Implementation

The definitions of the wait() and signal() semaphore operations just described present the same problem. To overcome the need for busy waiting, we can modify the definition of the wait() and signal() operations as follows: When a process executes the wait() operation and finds that the semaphore value is not positive, it must wait. However, rather than engaging in busy waiting, the process can block itself. The block operation places a process into a waiting queue associated with the semaphore, and the state of the process is switched to the waiting state. Then control is transferred to the CPU scheduler, which selects another process to execute.

A process that is blocked, waiting on a semaphore S, should be restarted when some other process executes a signal() operation. The process is restarted by a wakeup() operation, which changes the process from the waiting state to the ready state. The process is then placed in the ready queue. (The CPU may or may not be switched from the running process to the newly ready process, depending on the CPU-scheduling algorithm.)

To implement semaphores under this definition, we define a semaphore as follows:

```
typedef struct {
    int value;
    struct process *list;
} semaphore;
```

Each semaphore has an integer value and a list of processes list. When a process must wait on a semaphore, it is added to the list of processes. A signal() operation removes one process from the list of waiting processes and awakens that process.

Now, the wait() semaphore operation can be defined as

```
wait(semaphore *S) {
    S->value--;
    if (S->value < 0) {
        add this process to S->list;
        block();
    }
}
```

and the signal() semaphore operation can be defined as

```
signal(semaphore *S) {
    S->value++;
    if (S->value <= 0) {
        remove a process P from S->list;
        wakeup(P);
    }
}
```

The block() operation suspends the process that invokes it. The wakeup(P) operation resumes the execution of a blocked process P. These two operations are provided by the operating system as basic system calls.

Note that in this implementation, semaphore values may be negative, whereas semaphore values are never negative under the classical definition of semaphores with busy waiting. If a semaphore value is negative, its magnitude is the number of processes waiting on that semaphore. This fact results from switching the order of the decrement and the test in the implementation of the wait() operation.

Deadlocks and Starvation

The implementation of a semaphore with a waiting queue may result in a situation where two or more processes are waiting indefinitely for an event that can be caused only by one of the waiting processes. The event in question is the execution of a signal() operation. When such a state is reached, these processes are said to be **deadlocked**.

To illustrate this, consider a system consisting of two processes, P_0 and P_1, each accessing two semaphores, S and Q, set to the value 1:

```
     P0            P1
  wait(S);      wait(Q);
  wait(Q);      wait(S);
     .             .
     .             .
     .             .
  signal(S);    signal(Q);
  signal(Q);    signal(S);
```

Suppose that P_0 executes wait(S) and then P_1 executes wait(Q). When P_0 executes wait(Q), it must wait until P_1 executes signal(Q). Similarly, when P_1 executes wait(S) , it must wait until P_0 executes signal(S). Since these signal() operations cannot be executed, P_0 and P_1 are deadlocked.

We say that a set of processes is in a deadlocked state when every process in the set is waiting for an event that can be caused only by another process in the set. The events with which we are mainly concerned here are resource acquisition and release. Other types of events may result in deadlocks, as we show in Chapter 7. In that chapter, we describe various mechanisms for dealing with the deadlock problem.

Another problem related to deadlocks is **indefinite blocking** or **starvation**, a situation in which processes wait indefinitely within the semaphore. Indefinite blocking may occur if we remove processes from the list associated with a semaphore in LIFO (last-in, first-out) order.

Priority Inversion

A scheduling challenge arises when a higher-priority process needs to read or modify kernel data that are currently being accessed by a lower-priority process — or a chain of lower-priority processes. Since kernel data are typically protected with a lock, the higher-priority process will have to wait for a lower-priority one to finish with the resource. The situation becomes more complicated if the lower-priority process is preempted in favor of another process with a higher priority.

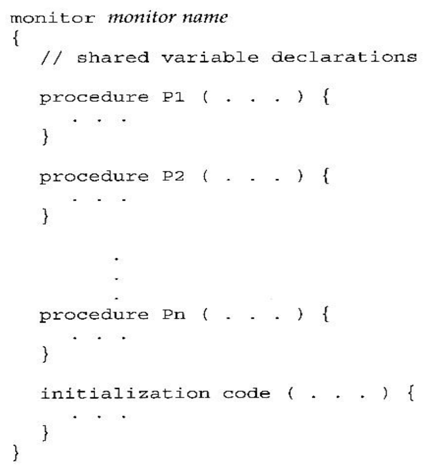
As an example, assume we have three processes — *L* , *M*, and *H* — whose priorities follow the order $L < M < H$. Assume that process *H* requires resource *R*, which is currently being accessed by process *L*. Ordinarily, process *H* would wait for *L* to finish using resource *R*. However, now suppose that process *M* becomes runnable, thereby preempting process *L*. Indirectly, a process with a lower priority — process *M*— has affected how long process *H* must wait for *L* to relinquish resource *R*.

This problem is known as **priority inversion**. It occurs only in systems with more than two priorities, so one solution is to have only two priorities. That is insufficient for most general-purpose operating systems, however. Typically these systems solve the problem by implementing a **priority-inheritance protocol**.

According to this protocol, all processes that are accessing resources needed by a higher-priority process inherit the higher priority until they are finished with the resources in question. When they are finished, their priorities revert to their original values. In the example above, a priority-inheritance protocol would allow process *L* to temporarily inherit the priority of process *H*, thereby preventing process *M* from preempting its execution. When process had finished using resource *R*, it would relinquish its inherited priority from *H* and assume its original priority. Because resource *R* would now be available, process *H* — not *M*— would run next.

Monitors

- Semaphores can be very useful for solving concurrency problems, ***but only if programmers use them properly.*** If even one process fails to abide by the proper use of semaphores, either accidentally or deliberately, then the whole system breaks down. (And since concurrency problems are by definition rare events, the problem code may easily go unnoticed and/or be heinous to debug.)
- For this reason a higher-level language construct has been developed, called ***monitors***.



```
monitor monitor name
{
   // shared variable declarations

   procedure P1 ( . . . ) {
      . . .
   }

   procedure P2 ( . . . ) {
      . . .
   }

         .
         .
         .

   procedure Pn ( . . . ) {
      . . .
   }

   initialization code ( . . . ) {
      . . .
   }
}
```

Figure - Syntax of a monitor.

Monitor Usage

- A monitor is essentially a class, in which all data is private, and with the special restriction that only one method within any given monitor object may be active at the same time. An additional restriction is that monitor methods may only access the shared data within the monitor and any data passed to them as parameters. I.e. they cannot access any data external to the monitor.

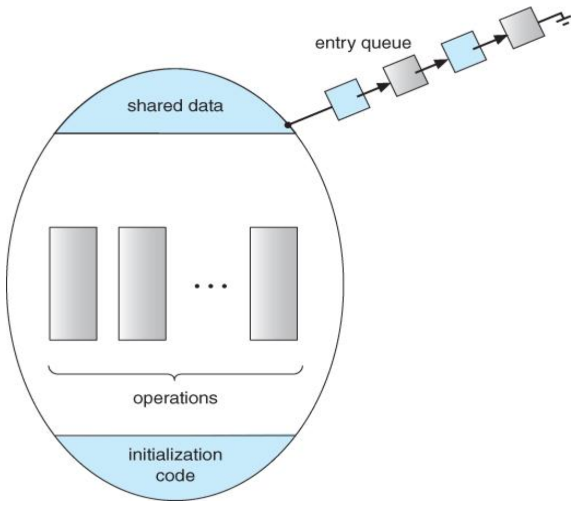
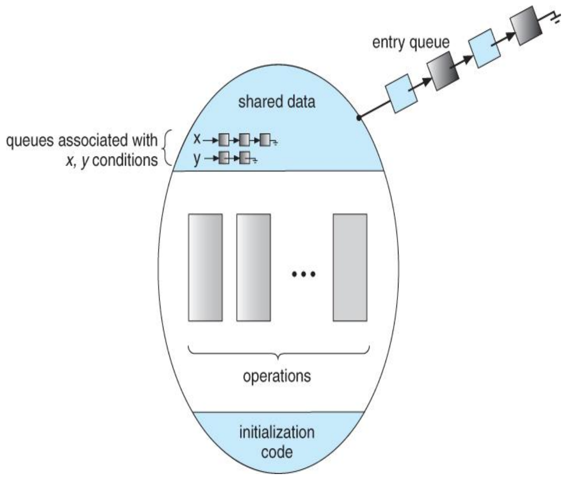


Figure - Schematic view of a monitor

- In order to fully realize the potential of monitors, we need to introduce one additional new data type, known as a ***condition***.
  - A variable of type condition has only two legal operations, ***wait*** and ***signal***. I.e. if X was defined as type condition, then legal operations would be X.wait( ) and X.signal( )
  - The wait operation blocks a process



  - until some other process calls signal, and adds the blocked process onto a list associated with that condition.
  - The signal process does nothing if there are no processes waiting on that condition. Otherwise it wakes up exactly one process from the condition's list of waiting processes. (Contrast this with counting semaphores, which always affect the semaphore on a signal call.)

But now there is a potential problem - If process P within the monitor issues a signal that would wake up process Q also within the monitor, then there would be two processes running simultaneously within the monitor, violating the exclusion requirement. Accordingly there are two possible solutions to this dilemma:

Signal and wait - When process P issues the signal to wake up process Q, P then waits, either for Q to leave the monitor or on some other condition.

Signal and continue - When P issues the signal, Q waits, either for P to exit the monitor or for some other condition.

There are arguments for and against either choice. Concurrent Pascal offers a third alternative - The signal call causes the signaling process to immediately exit the monitor, so that the waiting process can then wake up and proceed.

- Java and C# (C sharp) offer monitors bulit-in to the language. Erlang offers similar but different constructs.

CLASSIC PROBLEMS OF SYNCHRONIZATION

In this section, we present a number of synchronization problems as examples of a large class of concurrency-control problems. These problems are used for testing nearly every newly proposed synchronization scheme. In our solutions to the problems, we use semaphores for synchronization, since that is the traditional way to present such solutions. However, actual implementations of these solutions could use mutex locks in place of binary semaphores

The Bounded-Buffer Problem

The ***bounded-buffer problem*** is commonly used to illustrate the power of synchronization primitives. Here, we present a general structure of this scheme without committing ourselves to any particular implementation. We provide a related programming project in the exercises at the end of the chapter.

In our problem, the producer and consumer processes share the following data structures:

```
int n;
semaphore mutex = 1;
semaphore empty = n;
semaphore full = 0
```

We assume that the pool consists of n buffers, each capable of holding one item. The mutex semaphore provides mutual exclusion for accesses to the buffer pool and is initialized to the value 1. The empty and full semaphores count the number of empty and full buffers. The semaphore empty is initialized to the value n; the semaphore full is initialized to the value 0.

The code for the producer process is shown in Figure 5.9, and the code for the consumer process is shown in Figure 5.10. Note the symmetry between the producer and the consumer. We can interpret this code as the producer producing full buffers for the consumer or as the consumer producing empty buffers for the producer.

```
do {
   wait(full);
   wait(mutex);
      . . .
                        /* remove an item from buffer to next consumed */
      . . .
   signal(mutex);
   signal(empty);
      . . .
   /* consume the item in next consumed */
      . . .
} while (true);
```

The structure of the consumer process.

The Readers – Writers Problem

Suppose that a database is to be shared among several concurrent processes. Some of these processes may want only to read the database, whereas others may want to update (that is, to read and write) the database. We distinguish between these two types of processes by referring to the former as ***readers*** and to the latter as ***writers***. Obviously, if two readers access the shared data simultaneously, no adverse effects will result. However, if a writer and some other process (either a reader or a writer) access the database simultaneously, chaos may ensue.

To ensure that these difficulties do not arise, we require that the writers have exclusive access to the shared database while writing to the database. This synchronization problem is referred to as the **readers–writers problem**. Since it was originally stated, it has been used to test nearly every new synchronization primitive. The readers – writers problem has several variations, all involving priorities. The simplest one, referred to as the ***first*** readers – writers problem, requires that no reader be kept waiting unless a writer has already obtained permission to use the shared object. In other words, no reader should wait for other readers to finish simply because a writer is waiting. The ***second*** readers–writers problem requires that, once a writer is ready, that writer perform its write as soon as possible. In other words, if a writer is waiting to access the object, no new readers may start reading.

A solution to either problem may result in starvation. In the first case, writers may starve; in the second case, readers may starve. For this reason, other variants of the problem have been proposed. Next, we present a solution to the first readers – writers problem. See the bibliographical notes at the end of the chapter for references describing starvation-free solutions to the second readers – writers problem.

In the solution to the first readers – writers problem, the reader processes share the following data structures:

```
semaphore rw mutex = 1;
semaphore mutex = 1;
int read count = 0;
```

The semaphores mutex and rw mutex are initialized to 1; read count is initialized to 0. The semaphore rw mutex is common to both reader and writer

```
do {
    wait(rw mutex);

        . . .
    /* writing is performed */
        . . .
    signal(rw mutex);

} while (true);
```

Figure 5.11 The structure of a writer process.

processes. The mutex semaphore is used to ensure mutual exclusion when the variable read count is updated. The read count variable keeps track of how many processes are currently reading the object. The semaphore rw_mutex functions as a mutual exclusion semaphore for the writers. It is also used by the first or last reader that enters or exits the critical section. It is not used by readers who enter or exit while other readers are in their critical sections.

The code for a writer process is shown in Figure 5.11; the code for a reader process is shown in Figure 5.12. Note that, if a writer is in the critical section and *n* readers are waiting, then one reader is queued on rw mutex, and $n - 1$ readers are queued on mutex. Also observe that, when a writer executes signal(rw mutex), we may resume the execution of either the waiting readers or a single waiting writer. The selection is made by the scheduler.

The readers – writers problem and its solutions have been generalized to provide **reader–writer** locks on some systems. Acquiring a reader – writer lock requires specifying the mode of the lock: either *read* or *write* access. When a process wishes only to read shared data, it requests the reader – writer lock in read mode. A process wishing to modify the shared data must request the lock in write mode. Multiple processes are permitted to concurrently acquire a reader – writer lock in read mode, but only one process may acquire the lock for writing, as exclusive access is required for writers.

The Dining-Philosophers Problem

Consider five philosophers who spend their lives thinking and eating. The philosophers share a circular table surrounded by five chairs, each belonging to one philosopher. In the center of the table is a bowl of rice, and the table is laid with five single chopsticks (Figure 5.13).

When a philosopher thinks, she does not interact with her colleagues. From time to time, a philosopher gets hungry and tries to pick up the two chopsticks that are closest to her (the chopsticks that are between her and her left and right neighbors).

A philosopher may pick up only one chopstick at a time. Obviously, she cannot pick up a chopstick that is already in the hand of a neighbor. When a hungry philosopher has both her chopsticks at the same time, she eats without releasing the chopsticks. When she is finished eating, she puts down both chopsticks and starts thinking again.

The **dining-philosophers problem** is considered a classic synchronization problem neither because of its practical importance nor because computer scientists dislike philosophers but because it is an example of a large class of concurrency-control problems. It is a simple representation of the need to allocate several resources among several processes in a deadlock-free and starvation-free manner.

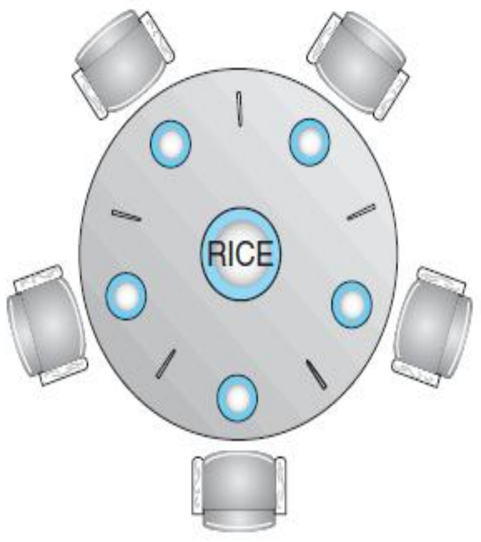


Figure 5.13 The situation of the dining philosophers.

One simple solution is to represent each chopstick with a semaphore. A philosopher tries to grab a chopstick by executing a wait() operation on that semaphore. She releases her chopsticks by executing the signal() operation on the appropriate semaphores. Thus, the shared data are

semaphore chopstick[5];

where all the elements of chopstick are initialized to 1.

The Structure of Philosopher i as follows:

```
do {
    wait(chopstick[i]);
    wait(chopstick[(i+1) % 5]);
        . . .
    /* eat for awhile */
        . . .
    signal(chopstick[i]);
    signal(chopstick[(i+1) % 5]);
        . . .
    /* think for awhile */
        . . .

} while (true);
```

Although this solution guarantees that no two neighbors are eating simultaneously, it nevertheless must be rejected because it could create a deadlock. Suppose that all five philosophers become hungry at the same time and each grabs her left chopstick. All the elements of chopstick will now be equal to 0. When each philosopher tries to grab her right chopstick, she will be delayed forever.

Several possible remedies to the deadlock problem are replaced by:

- Allow at most four philosophers to be sitting simultaneously at the table.
- Allow a philosopher to pick up her chopsticks only if both chopsticks are available (to do this, she must pick them up in a critical section).
- Use an asymmetric solution — that is, an odd-numbered philosopher picks up first her left chopstick and then her right chopstick, whereas an even-numbered philosopher picks up her right chopstick and then her left chopstick.

we present a solution to the dining-philosophers problem that ensures freedom from deadlocks. Note, however, that any satisfactory solution to the dining-philosophers problem must guard against the possibility that one of the philosophers will starve to death. A deadlock-free solution does not necessarily eliminate the possibility of starvation.