MODULE – I
Introduction to Operating Systems

Operating system is a program that acts as an intermediary between a user of a computer and the computer hardware. Goals of Operating system are:

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

Components of Computer Systems

A computer system can be divided roughly into four components: the hardware, the operating system, the application programs and the users as shown in Figure 1.1.

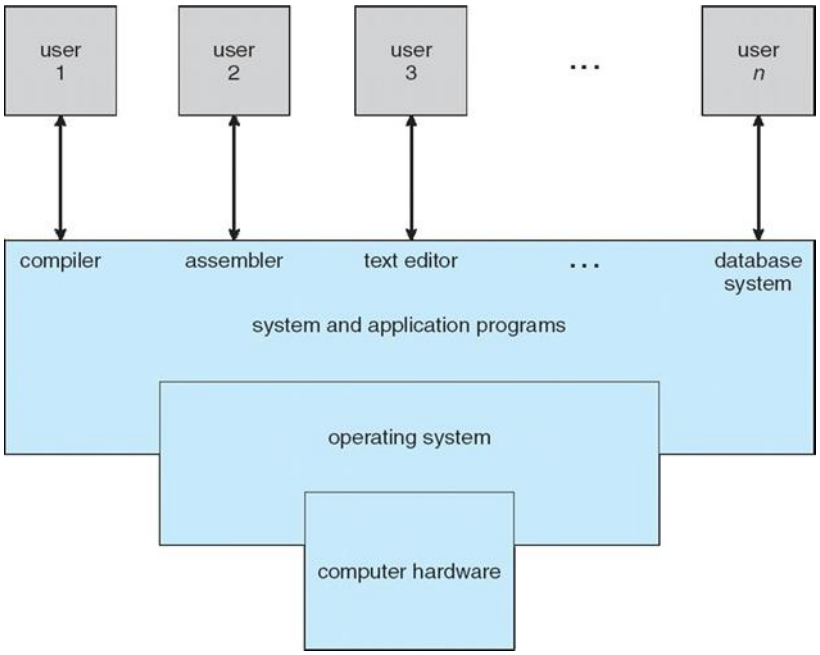


Figure 1.1 Components of Computer System

- Hardware – provides basic computing resources Ex: Central Processing Unit (CPU), memory, and Input / Output (I/O) devices.
- Application programs – define the ways in which the system resources are used to solve the computing problems of the users Ex: Word processors, compilers, web browsers, database systems, video games.
- Operating system - Controls and coordinates use of hardware among various applications and users.
- Users - People, machines, other computers.

Operating System Viewpoints

Operating systems role can be understood from two viewpoints: the user and the system viewpoints.

User View

- The user's view of the computer varies according to the interface being used.
- Most computer users sit in front of a**Personal Computers (PC)** consisting of a monitor/keyboard/ mouse, and system unit. Such a system is designed for one user to monopolize its resources rather than for multiple users.
- The goal is to maximize the work (or play) that the user is performing.

- The operating system is designed for ease of use with some attention paid to performance and without considering resource utilization how various hardware and software resources are shared.
- In other cases, a user sits at a terminal connected to **main frame or mini computers**.
- These users can share resources and exchange messages.
- The OS is designed to maximize resource utilization to assure that all available CPU time, memory, and I/0 are used efficiently and that no individual user takes more than her fair share.
- **Workstations** are connected to networks of other workstations and servers.
- These users have dedicated resources at their disposal, but they also share resources such as networking and servers such as file, print servers.
- Therefore, such an OS is designed to compromise between individual usability and resource utilization.
- **Hand held computers** are standalone units for individual users. Some are connected to networks, either directly by wire or through wireless modems and networking.
- Their operating systems are designed mostly for individual usability, but performance per unit of battery life is important as well.
- Some computers have little or no user view. For example, **embedded computers** in home devices and automobiles may have numeric keypads and may turn indicator lights on or off to show status, but they and their operating systems are designed primarily to run without user intervention.

System View

- From the computer's or system's point of view, the operating system can be viewed as a **resource locator (hardware) and control program**.
- A computer system has many resources that may be required to solve a problem such as CPU time, memory space, file-storage space, I/0 devices, and so on.
- The operating system acts as the manager of these resources. Facing numerous and possibly conflicting requests for resources, the operating system must decide how to allocate them to specific programs and users so that it can operate the computer system efficiently and fairly.
- A 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

1. Computer System Operation

- 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 (Figure1.2).
- Each device controller is in charge of a specific type of device (for example, disk drives, audio devices, and video displays).
- The CPU and the device controllers can execute concurrently.
- For a computer to start running for instance, when it is powered up or rebooted-it needs to have an initial program to run called **initial or bootstrap program (firmware)**. It is usually stored in ROM or EEPROM. It initializes all aspects of the system.
- To accomplish this goal, the bootstrap program must locate and load into memory the operating system kernel. The operating system then starts executing the first process, such as "init," and waits for some event to occur.

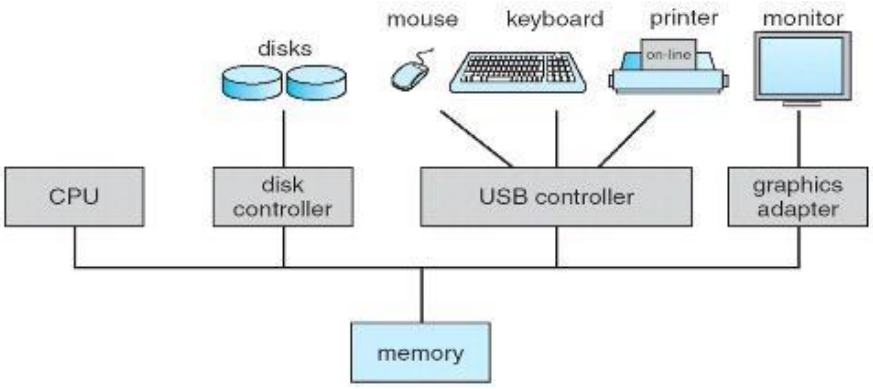


Figure 1.2 Modern Computer System

- 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 **system (monitor) call**.
- When the CPU is interrupted, it stops what it is doing and immediately transfers execution to a fixed location. The fixed location usually contains the starting address where the service routine for the interrupt is located. The interrupt service routine executes; on completion, the CPU resumes the interrupted computation. A time line of this operation is shown in Figure 1.3.

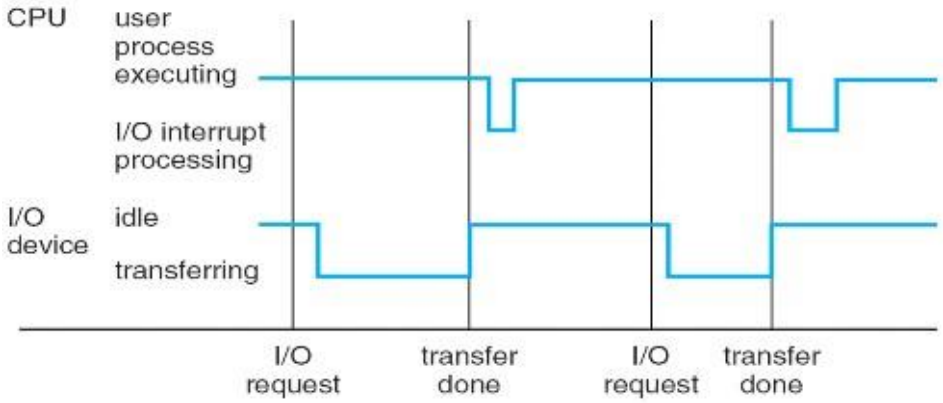


Figure 1.3 Interrupt time line for a single process doing output

- Interrupts must be handled quickly.
- Since only a predefined number of interrupts is possible, a table of pointers to interrupt routines is used to provide the necessary speed. This table is called **interrupt vector**. Whenever interrupt occurs interrupt routines address is fetched from the interrupt vector and is processed.

2. 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**. Main memory is usually implemented in a semiconductor technology called **dynamic random access memory (DRAM)**.
- Because the **read-only memory (ROM)** cannot be changed, only static programs are stored there.
- EEPROM cannot be changed frequently and so contains mostly static programs. For example, smart phones have EEPROM to store their factory installed programs.
- A typical **instruction-execution cycle** first **fetches** an instruction from memory and

stores that instruction in the **instruction register**. The instruction is then **decoded** and may cause operands to be fetched from memory and **stored** in some internal register. After that instruction is **executed** and the result may be stored back in memory.

- 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 **volatilestorage** device that loses its contents when power is turned off or otherwise lost.
- Thus, most computer systems provide **secondary storage** as an extension of main memory. The most common secondary-storage device is a **magnetic disk**, which provides storage for both programs and data. Most programs (system and application) are stored on a disk until they are loaded into memory.
- Other memory devices include cache memory, CD-ROM, magnetic tapes, and so on.
- The wide variety of storage systems in a computer system can be organized in a hierarchy (Figure 1.4) according to speed and cost. The higher levels are expensive, but they are fast.
- In addition to differing in speed and cost, the various storage systems are either **volatile or nonvolatile**.
- **Volatile storage** loses its contents when the power to the device is removed. In the absence of expensive battery and generator backup systems, data must be written **nonvolatile storage** to for safekeeping.
- In the hierarchy shown in Figure 1.4, the storage systems above the electronic disk are volatile, whereas those below nonvolatile. An electronic disk can be designed to be either volatile or nonvolatile.

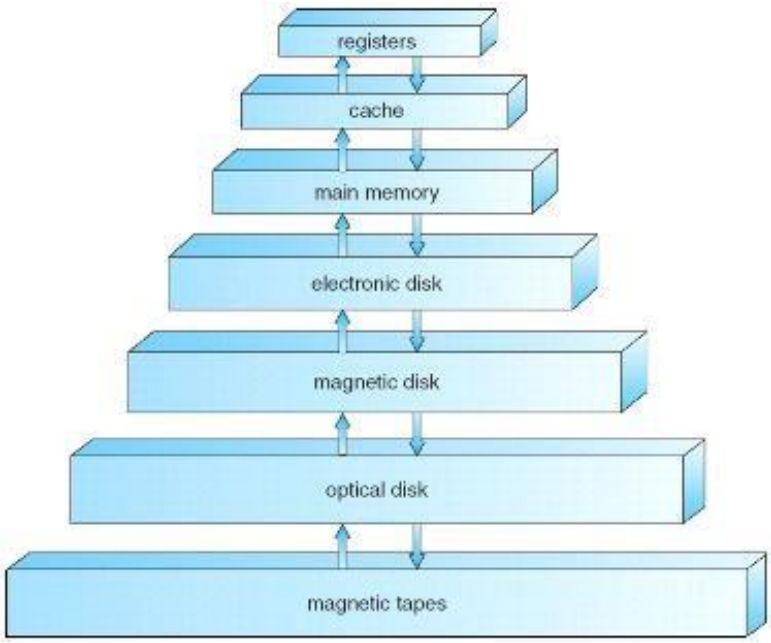


Figure 1.4 Storage device hierarchy

3. I/O structure

- 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. Seven or more devices can be attached to the **small computer systems interface (SCSI)** controller.

- Typical OS have a device driver for each device controller.
- To start the I/O operation, the device driver loads the appropriate registers within the device controller. The controller starts the transfer of data from the device to its local buffer. Once the transfer of data is complete, the device controller informs the device driver via an interrupt that it has finished its data transfer.
- This form of interrupt driven method in fine for transfer of small amounts of data but can produce high overhead when used for bulk data movement such as disk I/O. To solve this **direct memory access (DMA)** is used. After setting up the buffers, pointers, and counters for the I/O device, the device controller transfers an entire block of data directly to or from its own buffer storage to memory, with no intervention by the CPU.
- **High-end systems** use switch rather than bus architecture. On these systems, multiple components can talk to other components concurrently, rather than competing for cycles on a shared bus. In this case, DMA is even more effective.
- Figure 1.5 shows the interplay of all components of a computer system.

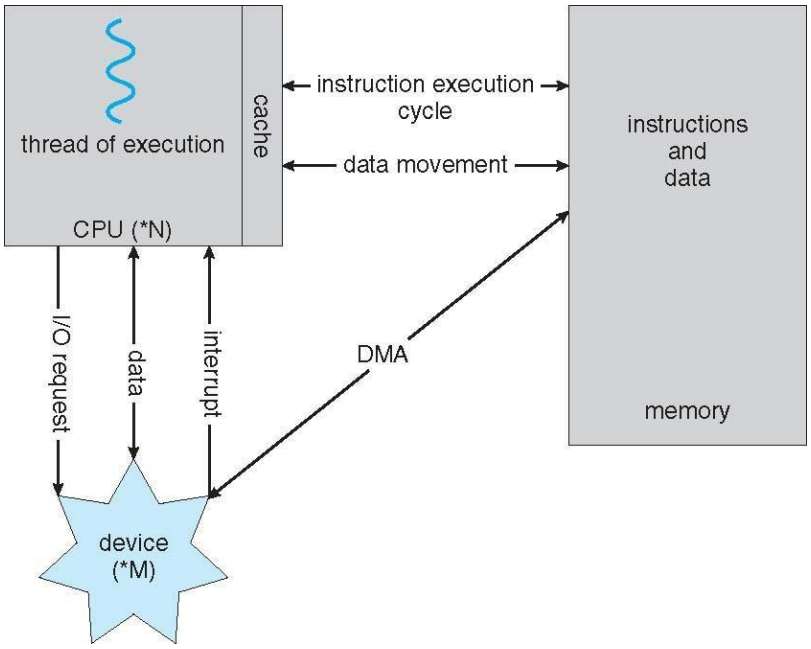


Figure 1.5 How a modern computer system works.

Computer System Architecture

1. Single-Processor Systems

- Most systems vise a single processor. For Example, systems range from PDAs through mainframes.
- 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 systems have other special-purpose processors as well.
- All of these special-purpose processors run a limited instruction set and do not run user processes. For example, PCs contain a microprocessor in the keyboard to convert the keystrokes into codes to be sent to the CPU.
- 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.

2. Multi-Processor Systems

- Multiprocessor systems have two or more processors in close communication, sharing the computer bus and sometimes the clock, memory, and peripheral devices.
- Multiprocessor systems have **three** main **advantages**:
    - **Increased throughput.** By increasing the number of processors, we expect to get more work done in less time. The speed-up ratio with *N* processors is not *N*, however; rather, it is less than N. When multiple processors cooperate on a task, a certain amount of overhead is incurred in keeping all the parts working correctly. This overhead, plus contention for shared resources, lowers the expected gain from additional processors
    - **Economy of scale.** Multiprocessor systems can cost less than equivalent multiple single-processor systems, because they can share peripherals, mass storage, and power supplies.
    - **Increased reliability.** If functions can be distributed properly among several processors, then the failure of one processor will not halt the system, only slow it down. If we have ten processors and one fails, then each of the remaining nine processors can pick up a share of the work of the failed processor. Thus, the entire system runs only 10 percent slower, rather than failing altogether.
- Increased reliability of a computer system is crucial in many applications. The ability to continue providing service proportional to the level of surviving hardware is called **graceful degradation.** Some systems go beyond graceful degradation and are called **fault tolerant,** because they can suffer a failure of any single component and still continue operation.
- There are **two types** of multiprocessor systems: **asymmetric and symmetric multiprocessors.**
    - In asymmetric multiprocessing each processor is assigned a specific task. A master processor controls the system; the other processors either look to the master for instruction or have predefined tasks. This scheme defines a master-slave relationship. The master processor schedules and allocates work to the slave processors.
    - Symmetric multiprocessing (SMP) means that all processors are peers; no master-slave relationship exists between processors.

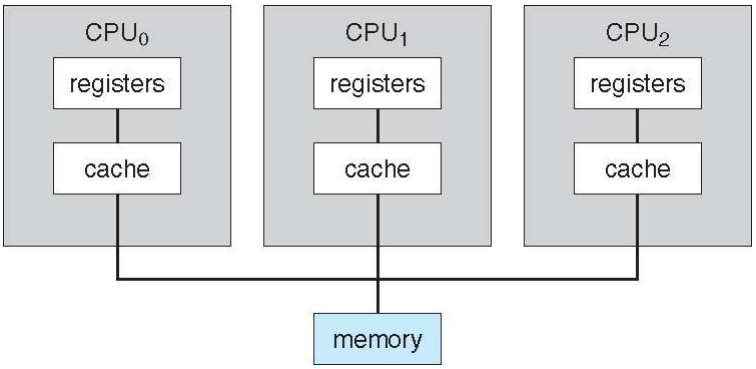


Figure1.6 Symmetric multiprocessing architecture.

Figure 1.6 illustrates a typical SMP architecture. An example of the SMP system is Solaris. The benefit of this model is that many processes can run

simultaneously—*N* processes can run if there are *N* CPUs—without causing a significant deterioration of performance. However, we must carefully control I/O to ensure that the data reach the appropriate processor. Also, since the CPUs are separate, one may be sitting idle while another is overloaded, resulting in inefficiencies. These inefficiencies can be avoided if the processors share certain data structures.

- Multiprocessing adds CPUs to increase computing power.
- A recent trend in CPU design is to include **multiple computing cores** on a single chip. They can be more efficient than multiple chips with single cores because on-chip communication is faster than between-chip communication. In addition, one chip with multiple cores uses significantly less power than multiple single-core chips. As a result, multicore systems are especially well suited for server systems such as database and Web servers.
- In Figure 1.7, we show a dual-core design with two cores on the same chip. In this design, each core has its own register set as well as its own local cache; other designs might use a shared cache or a combination of local and shared caches, these processors appears as *N* standard processors to OS.

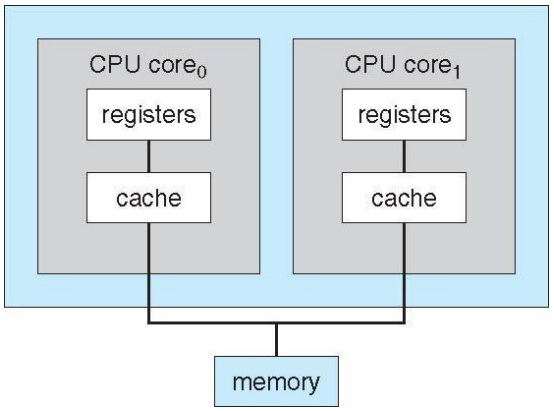


Figure 1.7 A dual-core design with two cores placed on the same chip.

- Lastly, **blade** servers are a recent development in which multiple processors boards, I/O boards, and networking boards are placed in the same chassis. The difference between these and traditional multiprocessor systems is that each blade-processor board boots independently and runs its own operating system.

3. Clustered Systems

- Another type of multiple-CPU system is the **clustered system.**
- Clustered systems differ from multiprocessor systems, in that they are composed of two or more individual systems coupled together.
- Clustered computers share storage and are closely linked via a **local-area network (LAN)** or a faster interconnect such as InfiniBand.
- Clustering is usually used to provide **high-availability** service; that is, service will continue even if one or more systems in the cluster fail.
- A layer of cluster software runs on the cluster nodes. Each node can monitor one or more of the others (over the LAN). If the monitored machine fails, the monitoring machine can take ownership of its storage and restart the applications that were running on the failed machine. The users and clients of the applications see only a brief interruption of service.
- Clustering can be structured asymmetrically or symmetrically.

- In **asymmetric clustering,** one machine is in **hot-standby mode** while the other is running the applications. The hot-standby host machine does nothing but monitor the active server. If that server fails, the hot-standby host becomes the active server.
- In **symmetric mode,** two or more hosts are running applications, and are monitoring each other. This mode is obviously more efficient, as it uses all of the available hardware. Such systems can supply significantly greater computational power than single-processor or even SMP systems because they are capable of running an application concurrently on all computers in the cluster. However, applications must be written specifically, to take advantage of the cluster by using a technique known as parallelization, which consists of dividing a program into separate components that run in parallel on individual computers in the cluster.
- Other forms of clusters include **parallel clusters and clustering over a wide-area network(WAN)**.
- **Parallel clusters** allow multiple hosts to access the same data on the shared storage. For example, Oracle Parallel Server. is a version of Oracle's database that has been designed to run on a parallel cluster. Each machine runs Oracle, and a layer of software tracks access to the shared disk. Each machine has full access to all data in the database. To provide this shared access to data, the system must also supply access control and locking to ensure that no conflicting operations occur. This function, commonly known as a **distributed lock manager (DLM),** is included in some cluster technology.
- Cluster technology is changing rapidly. Some cluster products support dozens of systems in a cluster, as well as clustered nodes that are separated by miles. Many of these improvements are made possible by **storage-area networks (SANs)**, which allow many systems to attach to a pool of storage. If the applications and their data are stored on the SAN, then the cluster software can assign the application to run on any host that is attached to the SAN. If the host fails, then any other host can take over. In a database cluster, dozens of hosts can share the same database, greatly increasing performance and reliability. Figure 1.8 depicts the general structure of a clustered system.

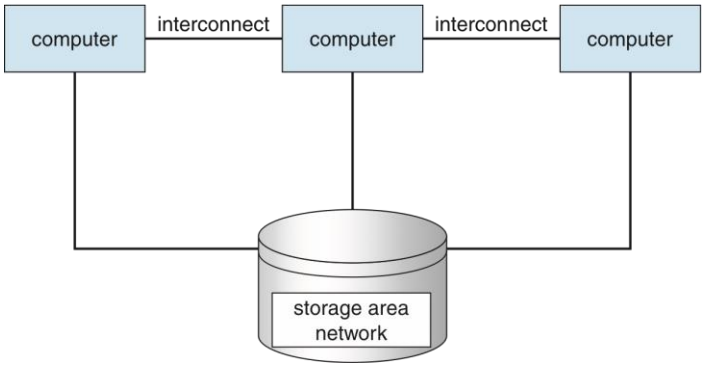


Figure 1.8 General structure of a clustered system.

Operating-System Structure

- One of the most important aspects of operating systems is the ability to **multi-program**.
- In a **non-multiprogrammed** system, the CPU would sit idle. One by one job is loaded in to memory for execution. If the job in the memory completes its execution, then next job is brought into the memory. While doing this, CPU would sits idle.
- The **multiprogramming** idea is as follows: All the jobs to be executed are kept in secondary storage called job pool. The operating system keeps several of these jobs in memory simultaneously (Figure 1.9). This set of jobs can be a subset of the jobs kept in

the job pool. The operating system simply switches to, and executes, another job. When *that* job needs to wait, the CPU is switched to *another* job, and so on. Eventually, the first job finishes waiting and gets the CPU back. As long as at least one job needs to execute, the CPU is never idle.

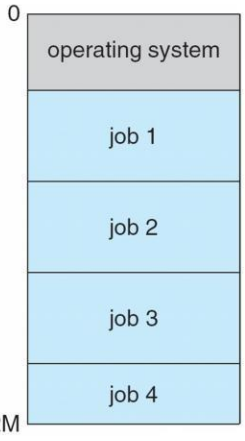


Figure1.9 Memory layout for a multiprogramming system.

- Example: 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.
- **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** (or **hands-on) 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 an input device such as a keyboard or a mouse, and waits for immediate results on an output device. Accordingly, the **response time** should be short—typically less than one second.
- A time-shared operating system allows many users to share the computer simultaneously. Since each action or command in a time-shared system tends to be short, only a little CPU time is needed for each user.
- As the system switches rapidly from one user to the next, each user is given the impression that the entire computer system is dedicated to his use, even though it is being shared among many users.
- Time-sharing and multiprogramming require several jobs to be kept simultaneously in memory. Since in general main memory is too small to accommodate all jobs, the jobs are kept initially on the disk in the **job pool.** 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 using **job scheduling** algorithms.

Operating-System Operations

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. 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 that is responsible for dealing 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 that was running. 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 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.
- At the very least, we need two separate **modes** of operation: **user mode** and **kernel mode** (also called **supervisor mode, system mode, or privileged mode).**
- A bit, called the **mode bit,** is added to the hardware of the computer to indicate the current mode: kernel (0) or user (1).
- 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), it must transition from user to kernel mode to fulfill the request. This is shown in Figure 1.10.

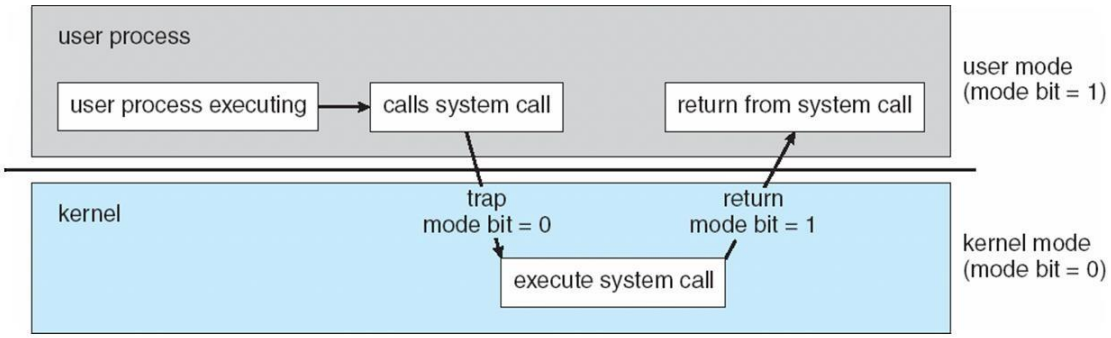


Figure 1.10 Transition from user to kernel mode.

- At system boot time, the hardware starts in kernel mode. The operating system is then loaded and starts user applications in user mode. Whenever a trap or interrupt occurs, the hardware switches from user mode to kernel mode (that is, changes the state of the mode bit to 0).
- Thus, whenever the operating system gains control of the computer, it is in kernel mode. The system always switches to user mode (by setting the mode bit to 1) before passing control to a user program.
- The dual mode of operation provides us with the means for protecting the operating system from **errant users**. We accomplish this protection by designating some of the machine instructions that may cause harm as **privileged instructions.** The hardware allows privileged instructions to be executed only in kernel mode.
    - If an attempt is made to execute a privileged instruction in user mode, the hardware does not execute the instruction but rather treats it as illegal and traps it to the operating system. The instruction to switch to user mode is an example of a privileged instruction. Some other examples include I/O control, timer management, and interrupt management.
- We can now see the **life cycle of instruction execution** in a computer system.
- Initial control is within the operating system, where instructions are executed in kernel mode. When control is given to a user application, the mode is set to user mode. Eventually, control is switched back to the operating system via an interrupt, a trap, or a system call. System calls provide the means for a user program to ask the operating system to perform tasks reserved for the operating system on the user program's behalf. When a system call is executed, it is treated by the hardware as software interrupt. Control passes through the interrupt vector to a service routine in the operating system, and the mode bit is set to kernel mode. The kernel examines the interrupting instruction to determine what system call has occurred; a parameter indicates what type of service the user program is requesting. The kernel verifies that the parameters are correct and legal,

executes the request, and returns control to the instruction following the system call.

- The lack of a hardware-supported dual mode can cause serious shortcomings in an operating system. For instance, MS-DOS was written for the Intel 8088 architecture, which has no mode bit and therefore no dual mode. Recent versions of the Intel CPU, such is the Pentium, do provide dual-mode operation. Accordingly, most contemporary operating systems, such as Microsoft Windows 2000 and Windows XP, and Linux and Solaris for x86 systems, take advantage of this feature and provide greater protection for the operating system.

Timer

- We must ensure that the operating system maintains control over the CPU. We must prevent a user program from getting stuck in an infinite loop or not calling system services and never returning 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.
- 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.
- Instructions that modify the content of the timer are privileged. Thus, we can use the timer to prevent a user program from running too long.
- A simple technique is to initialize a counter with the amount of time that a program is allowed to run. A program with a 7-minute time limit, for example, would have its counter initialized to 420. Every second, the timer interrupts and the counter is decremented by 1. As long as the counter is positive, control is returned to the user program. When the counter becomes negative, the operating system terminates the program for exceeding the assigned time limit.

Process Management

- A **program** is a **passive** entity, such as the contents of a file stored on disk, whereas a **process** is an **active** entity that is process is a program under execution. Example: a word-processing program being run by an individual user on a PC is a process.
- A process needs certain **resources** - including CPU time, memory, files, and I/O devices - to accomplish its task. These resources are either given to the process when it is created or - allocated to it while it is running.
- In addition to the various physical and logical resources that a process obtains when it is created, various initialization data (input) may be passed along.
- For example, consider a process whose function is to display the status of a file on the screen of a terminal. The process will be given as an input the name of the file and will execute the appropriate instructions and system calls to obtain and display on the terminal the desired information. When the process terminates, the operating system will reclaim any reusable resources.
- A single-threaded process has one **program counter** specifying the next instruction to execute. The execution of such a process must be sequential. The CPU executes one instruction of the process after another, until the process completes.
- A multithreaded process has multiple program counters, each pointing to the next instruction to execute for a given thread.
- The **operating system is responsible** for the following activities in connection with

process management:

- Creating and deleting both user and system processes
- Suspending and resuming processes
- Providing mechanisms for process synchronization
- Providing mechanisms for process communication
- Providing mechanisms for deadlock handling

Memory Management

- We know that, the main memory is central to the operation of a modern computer system.
- The central processor reads instructions from main memory during the instruction-fetch cycle and both reads and writes data from main memory during the data-fetch cycle.
- 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.
- To improve both the utilization of the CPU and the speed of the computer's response to its users, general-purpose computers must keep several programs in memory, creating a need for memory management.
- 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 by whom
    - Protecting processing from access to other processes memory location.
    - Allocating and de-allocating memory space as needed.

Storage Management

1. **File-System Management**
    - Computers can store information on several different types of physical media. For example: Magnetic disk, optical disk, and magnetic tape are the most common.
    - A file is a collection of related information defined by its creator.
    - Also, 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 by whom and in what ways (for example read, write, append) files may be accessed.
    - The **operating system is responsibilities** in connection with **file system 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
2. **Mass-Storage Management**
    - We know that 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 of the programs such as compilers, assemblers, word processors, editors etc. are stored on a disk until loaded into memory and then use the disk as both the source and destination of their processing.
    - Hence, proper management of disk storage is of central importance to a computer system.
    - The **OS is responsibilities** in connection with **Mass-Storage (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.
- Magnetic tape drives & their tapes, CD & DVD drives and platters are typical **tertiary storage** devices.
- Tertiary storage is not crucial to system performance, but it still must be managed. Some of the functions that operating systems can provide include mounting and un-mounting media in devices, allocating and freeing the devices for exclusive use by processes, and migrating data from secondary to tertiary storage.

3. Caching

- **Caching** is an important principle of computer systems.
- Information is normally kept in some storage system. As it is used, it is copied into a faster storage system such as **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 addition, internal programmable registers, such as index registers, provide a high-speed cache for main memory. Because caches have **limited size**, **cache management** is an important design problem. Careful selection of the cache size and of a replacement policy can result in greatly increased performance.

Level	1	2	3	4
Name	registers	cache	main memory	disk storage
Typical size	< 1 KB	> 16 MB	> 16 GB	> 100 GB
Implementation technology	custom memory with multiple ports, CMOS	on-chip or off-chip CMOS SRAM	CMOS DRAM	magnetic disk
Access time (ns)	0.25 – 0.5	0.5 – 25	80 – 250	5,000.000
Bandwidth (MB/sec)	20,000 – 100,000	5000 – 10,000	1000 – 5000	20 – 150
Managed by	compiler	hardware	operating system	operating system
Backed by	cache	main memory	disk	CD or tape

Figure 1.11 Performance of various levels of storage.

- Data transfer from cache to CPU and registers is usually a hardware function, with no operating-system intervention. In contrast, transfer of data from magnetic disk to main memory is usually controlled by the operating system.
- 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 (Figure 1.12). 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 register back to the magnetic disk.

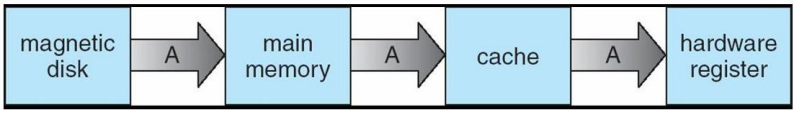


Figure 1.12 Migration of integer A from disk to register.

- The situation becomes more complicated in a multiprocessor environment where, in addition to maintaining internal registers, each of the CPUs also contains a local cache. In such an environment, a copy of A may exist simultaneously in several caches. Since the various CPUs can all execute concurrently, we must make sure that an update to the value of A in one cache is immediately reflected in all other caches where A resides. This situation is called **cache coherency,** and it is usually a hardware problem.
- In a distributed environment, the situation becomes even more complex. In this environment, several copies (or replicas) of the same file can be kept on different computers that are distributed in space. Since the various replicas may be accessed and updated concurrently, some distributed systems ensure that, when a replica is updated in one place, all other replicas are brought up to date as soon as possible.

4. 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/Osubsystem.**
- The **I/O subsystem** consists of several **components (responsibilities)**:
    - 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

- If a computer system has multiple users and allows the concurrent execution of multiple processes, then access to data must be regulated.
- **Protection** is any mechanism for controlling the access of processes or users to the resources defined by a computer system. This mechanism must provide means for specification of the controls to be imposed and means for enforcement.
- Protection can improve reliability by detecting latent errors at the interfaces between component subsystems. Early detection of interface errors can often prevent contamination of a healthy subsystem by another subsystem that is malfunctioning.
- A system can have adequate protection but still be prone to failure and allow inappropriate access.
- The job of **security** to defend a system from external and internal attacks. Such attacks spread across a huge range and include viruses and worms, denial-of-service attacks (which use all of a system's resources and so keep legitimate users out of the system), identity theft, and theft of service (unauthorized use of a system).
- Protection and security require the system to be able to distinguish among all its users.
- Most operating systems maintain a list of user names and associated **user identifiers (user IDs).**
- Group functionality can be implemented as a system-wide list of group names and **group identifiers.** A user can be in one or more groups, depending on OS design decisions.
- In the course of normal use of a system, the user ID and group ID for a user are sufficient. However, a user sometimes needs to **escalate privileges** to gain extra permissions for an activity. The user may need access to a device that is restricted, for example. Operating

systems provide various methods to allow privilege escalation. This can be achieved in UNIX using setuid.

Distributed Systems

- Collection of separate, possibly heterogeneous systems networked together to provide the users with access to the various resources that the system maintains.
- Access to a shared resource increases computation speed, functionality, data availability, and reliability. Some operating systems generalize network access as a form of file access, with the details of networking contained in the network interface's device driver.
- A **network,** in the simplest terms, is a communication path between two or more systems. Distributed systems depend on networking for their functionality.
- Networks are characterized based on the distances between their nodes.
    - Local Area Network (**LAN**)
    - Wide Area Network (**WAN**)
    - Metropolitan Area Network (**MAN**)
- Network Operating System provides features between systems across network such as
    - File sharing across the network and that includes a communication scheme that allows different processes on different computers to exchange messages.
    - A computer running a network operating system acts autonomously from all other computers on the network, although it is aware of the network and is able to communicate with other networked computers.
    - The different operating systems communicate closely enough to provide the illusion that only a single operating system controls the network.

Special-Purpose Systems

1. **Real-Time Embedded Systems**
    - Embedded computers are the most prevalent form of computers in existence. These devices are found everywhere, for example car engines, robots, VCRs and microwave ovens.
    - They tend to have very specific tasks. 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. Scientific experiments systems, medical imaging systems, industrial control systems, and certain display systems are real-time systems.
    - A real-time system has well-defined, fixed time constraints. Processing must be done within the defined constraints, or the system will fail. That is, it would not do for a robot arm to be instructed to halt afterit had smashed into the car it was building. A real-time system functions correctly only if it returns the correct result within its time constraints.
2. **Multimedia Systems**
    - A recent trend in technology is the incorporation of **multimedia data** into computer systems. Multimedia describes a wide range of applications that are in popular use today.
    - These include audio files such as MP3 DVD movies, video conferencing, and short video clips of movie previews or news stories downloaded over the Internet.
    - Multimedia applications may also include live webcasts (broadcasting over the World

Wide Web) of speeches or sporting events and even live webcams that allow a viewer in Manhattan to observe customers at a café in Paris.

3. **Handheld Systems**
   - **Handheld systems** include personal digital assistants (PDAs), such as Palm and Pocket-PCs, and cellular telephones use special-purpose embedded operating systems.
   - Developers of handheld systems and applications **face many challenges**, most of which are due to the limited size of such devices.
   - **Firstly**, the amount of physical memory in a handheld systems depends upon the device, but typically is somewhere between 512 KB and 128 MB. As a result, the operating system and applications must manage memory efficiently.
   - A **second** issue of concern to developers of handheld devices is the speed of the processor used in the devices. Faster processors require more power. To include a faster processor in a handheld device would require a larger battery, which would take up more space and would have to be replaced (or recharged) more frequently.
   - The **last issue** confronting program designers for handheld devices is I/O. A lack of physical space limits input methods to small keyboards, handwriting recognition, or small screen-based keyboards. The small display screens limit output options. Familiar tasks, such as reading e-mail and browsing web pages, must be condensed into smaller displays. One approach for displaying the content in web pages is **web clipping,** where only a small subset of a web page is delivered and displayed on the handheld device.
   - Generally, the limitations in the functionality of PDAs are balanced by their convenience and portability. Their use continues to expand as network connections become more available and other options, such as digital cameras and MP3 players, expand their utility.

Computing Environments

1. **Traditional Computing**
   - As computing matures, the lines separating many of the traditional computing environments are blurring.
   - Office environment
     - PCs connected to a network, terminals attached to mainframe or minicomputers providing batch and timesharing.
     - Now portals allowing networked and remote systems access to same resources.
     - **Network computers** are essentially terminals that understand web-based computing.
     - Handheld computers can synchronize with PCs to allow very portable use of company information.
   - Home networks
     - Most users had a single computer with a slow modem connection to the office, the Internet, or both.
     - Today, network-connection speeds cost are relatively inexpensive.
     - These fast data connections are allowing home computers to serve up web pages and to run networks that include printers, client PCs, and servers.
     - Use **firewalls** to protect their networks from security breaches

2. **Client-Server Computing**
   - Many of today's systems act as **server systems to** satisfy requests generated by **client systems.** This form of specialized distributed system, called **client-server** system, has the general structure depicted in Figure 1.13.

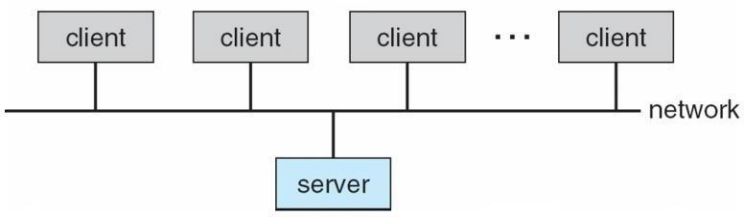


Figure 1.13General structure of a client-server system

- 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 back 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.

3. Peer-to-Peer Computing

- 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 the two 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.

4. 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.
- 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.

Operating-System Services

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). One is a **command-line interface (CLI),** which uses text commands. 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.
- **Program execution.** The system must be able to load a program into memory and to run that program. The program must be able to end its execution, either normally or abnormally (indicating error).
- **I/O operations.** A running program may require I/O, which may involve a file or an I/O device. For efficiency and protection, users usually cannot control I/O devices directly. Therefore, the operating system must provide a means to do I/O.
- **File-system manipulation.** Programs need to read and write files and directories. They also need to create and delete them by name, search for a given file, and list file information. Finally, some programs include permissions management to allow or deny access to files or directories based on file ownership.
- **Communications.** There are many circumstances in which one process needs to exchange information with another process. Such communication may occur between processes that are executing on the same computer or between processes.
- Communications may be implemented via **shared memory**or through **message passing**.
- **Error detection.** The operating system needs to be constantly aware of possible errors. Errors may occur in the CPU and memory (such as a memory error or a power failure), in I/O devices (such as a parity error on tape, a connection failure on a network, or lack of paper in the printer), and in the user program (such as an arithmetic overflow, an attempt to access an illegal memory location). For each type of error, the operating system should take the appropriate action to ensure correct and consistent computing. Debugging facilities can greatly enhance the user's and programmer's abilities to use the system efficiently.

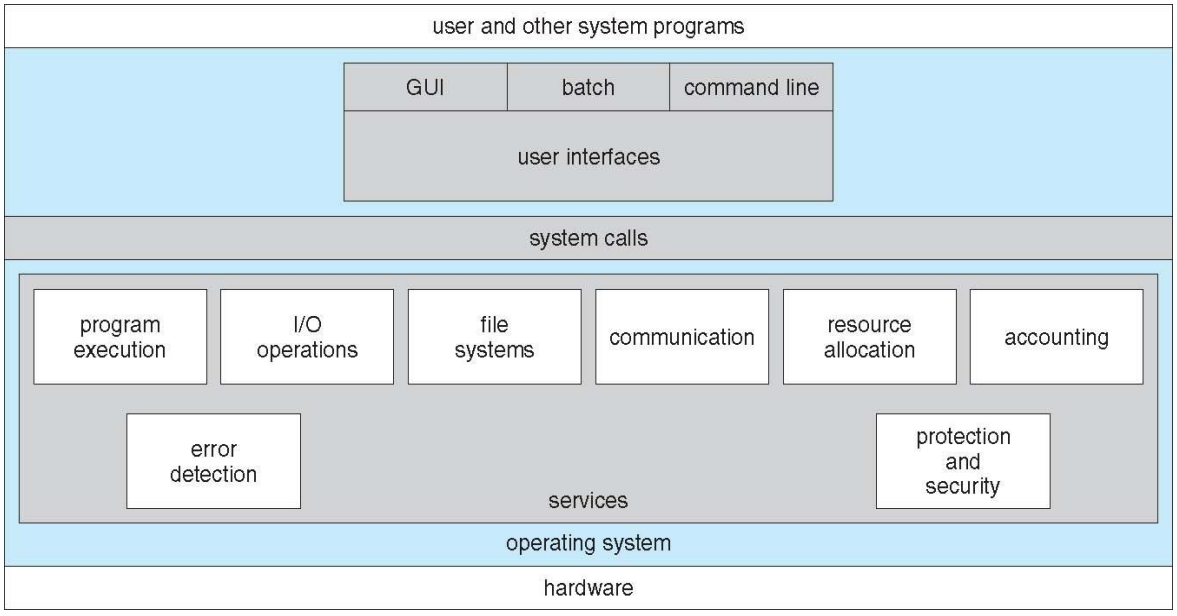


Figure 1.14 A view of operating system services.

Another set of operating-system functions exists **not for helping the user but rather for ensuring the efficient operation of the systemitself**.

- **Resource allocation.** When there are multiple users or multiple jobs running at the same time, resources must be allocated to each of them. For example: CPU, memory, I/O devices, printers, modems, USB storage drives, and other peripheral devices.
- **Accounting.** We want to keep track of which users use how much and what kinds of computer resources. This record is used for accounting or for accumulating usage statistics.
- **Protection and security. Protection** involves ensuring that all access to system resources is controlled. When several separate processes execute concurrently, it should not be possible for one process to interfere with the others or with the operating system itself. **Security** of the system from outsiders is also important. Such security starts with requiring each user to authenticate himself or herself to the system, usually by means of a password, to gain access to system resources.

System Calls

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

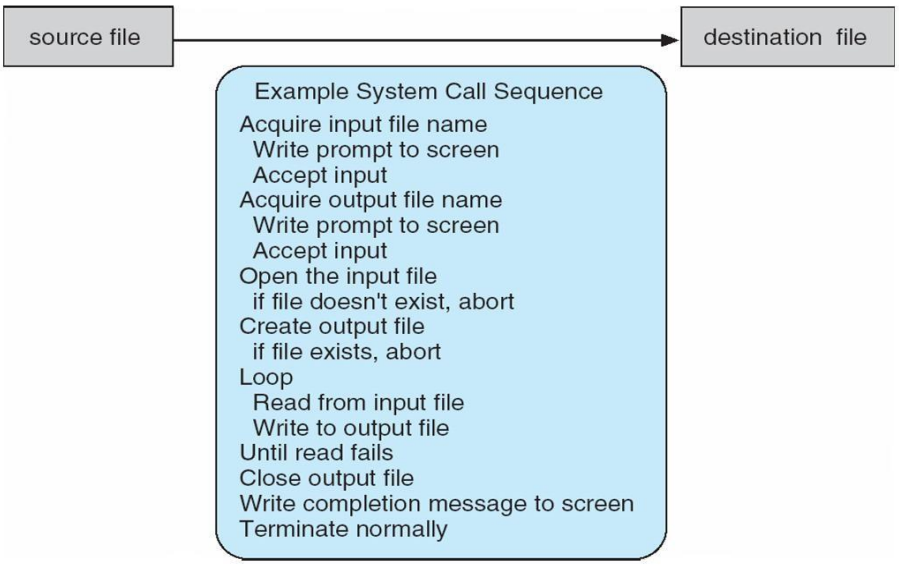


Figure 1.15 System call sequence to copy the contents of one file to another file.

- Typically, application developers design programs according to an **application programming interface (API).**
- **Three** of the most common APIs available to application programmers are the **Win32 API** for **Windows** systems, the **POSIX API** for **POSIX**-based systems (UNIX, Linux, and Mac OS X), and the **Java API** for designing programs that run on the **Java virtual machine**.
- Behind the scenes, the functions that make up an API typically invoke the actual system calls on behalf of the application programmer. For example, the Win32 function CreateProcess() actually calls the NTCreateProcess() system call in the Windows kernel.
- Why would an application programmer prefer programming according to an API rather than invoking actual system calls?

- One benefit of programming according to an API concerns program portability: An application programmer designing a program using an API can expect her program to compile and run on any system that supports the same API. Furthermore, actual system calls can often be more detailed and difficult to work with than the API available to an application programmer.
- The caller needs to know nothing about how the system call is implemented or what it does during execution.
- Thus, most of the details of the operating-system interface are hidden from the programmer by the API and are managed by the run-time support library.
- The relationship between an API, the system-call interface, and the operating system is shown in Figure 1.17, which illustrates how the operating system handles a user application invoking the open() system call.

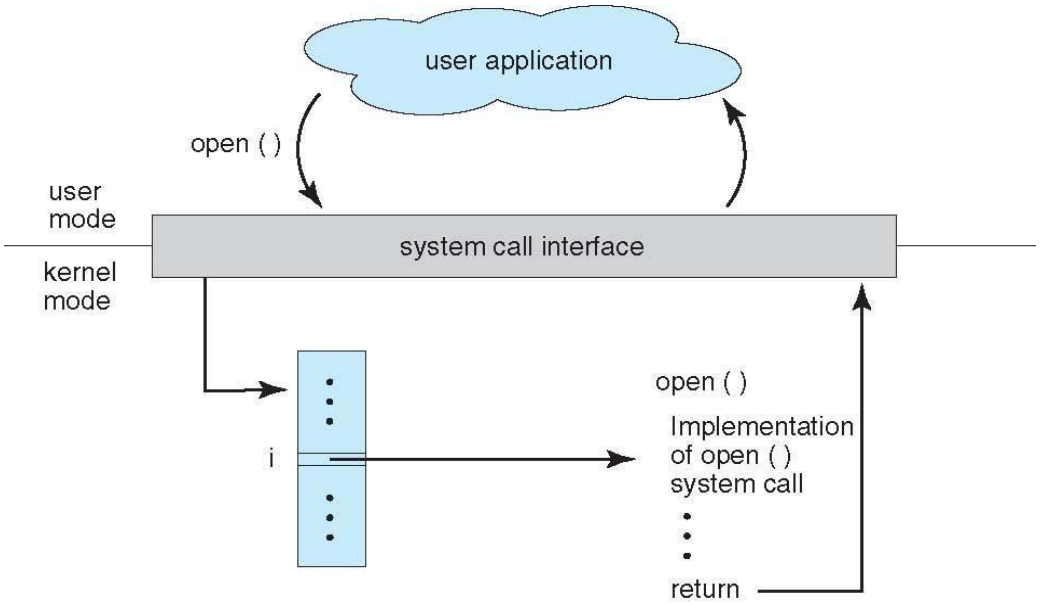


Figure 1.16 The handling of a user application invoking the open () system call.

Three general methods are used to pass parameters to the operating system.

1. The simplest approach is to pass the parameters in registers.
2. In some cases, there may be more parameters than registers. In these cases, the parameters are generally stored in a *block*, or table, in memory, and the address of the block is passed as a parameter in a register (Figure 1.17). This is the approach taken by Linux and Solaris.
3. Parameters also can be placed, or pushed, onto the *stack* by the program and poppedoff the stack by the operating system. Some operating systems prefer the block or stack method, because those approaches do not limit the number or length of parameters being passed.

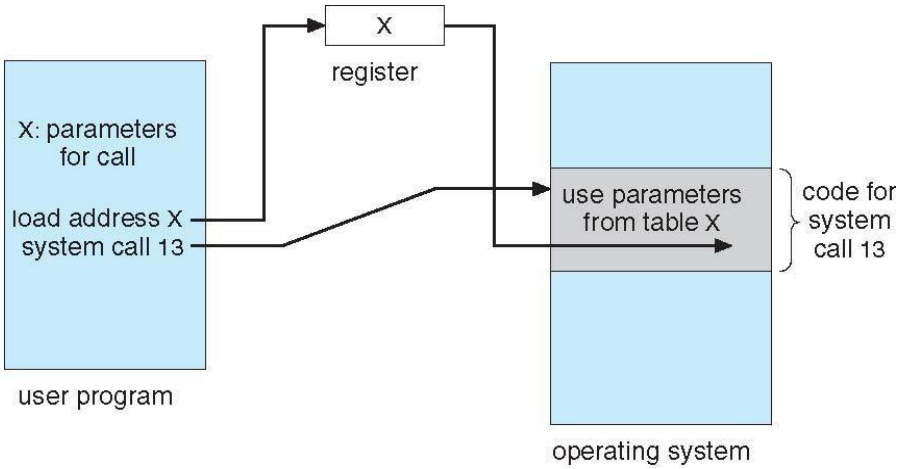


Figure 1.17 Passing of parameters as a table.

Types of System Calls

1. **Process Control**
   - **create process, terminate process** – To create and terminate the new process.
   - **end, abort** - A running program needs to be able to halt its execution either normally (end) or abnormally (abort). If a system call is made to terminate the currently running program abnormally, or if the program runs into a problem and causes an error trap, a dump of memory is sometimes taken and an error message generated.
   - **load, execute** - process or job executing one program may want to load and execute another program. This feature allows the command interpreter to execute a program as directed by, for example, a user command, the click of a mouse, or a batch command.
   - **get process attributes, set process attributes** - If we create a new job or process, or perhaps even a set of jobs or processes, we should be able to control its execution. This control requires the ability to determine and reset the attributes of a job or process, including the job's priority, its maximum allowable execution time, and so on.
   - **wait for time, wait event** - Having created new jobs or processes, we may need to wait for them to finish their execution. We may want to wait for a certain amount of time to pass (wait time); more probably, we will want to wait for a specific event to occur (wait event).
   - **signal event** - The jobs or processes should signal when event has occurred.
   - **allocate and free memory** – when process is created it needs memory. Allocate is used to assign memory to process and free memory is to remove assigned memory.
2. **File Management**
   - **create file, delete file** - We first need to be able to create and delete files. Either system call requires the name of the file and perhaps some of the file's attributes.
   - **open file** - Once the file is created, we need to open it to use it.
   - **read, write, reposition** - Used to read, write, or reposition (rewinding or skipping to the end of the file, for example) file.
   - **close file** - Finally, we need to close the file, indicating that we are no longer using it.
   - **get and set file attributes** – Used to set and get file and directory attributes such as owner, type, created time, modified time etc.
3. **Device Management**
   - **request device, release device** - A process may need several resources to execute—main memory, disk drives, access to files, and so on. If the resources are available, they can be granted, and control can be returned to the user process.

- **read, write, reposition** - Once the device has been requested (and allocated to us), we can read, write, and (possibly) reposition the device, just as we can with files.
- **get device attributes, set device attributes** –Sometimes, I/O devices are identified by special file names, directory placement, or file attributes. This can be done using get and set device attributes system calls.
- **logically attach or detach devices** - to attach device or detach to particular user etc.

4. **Information Maintenance**
   - **get time or date, set time or date** – used to get and set time or date.
   - **get system data, set system data** – used to get or set system information
   - **get process, file, or device attributes** – used to get process, file or device information.
   - **set process, file, or device attributes** – used to set process, file or device information.

5. **Communication**
   - **create, delete communication connection** - There are two common models of **interprocess** communication: the **messagepassing** model and the **shared-memory** model. These are create and delete connection between processes.
   - **send, receive messages** – are used to send and receive messages between hosts on the network or between processes.
   - **transfer status information** – used to get the status information of message transfer.
   - **attach and detach remote devices** - In the shared-memory model, processes use shared memory. Attach system call is used create and gain access to regions of memory owned by other processes. Detach is used to remove the access.

6. **Protection**
   - Protection provides a mechanism for controlling access to the resources provided by a computer system.
   - 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.

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

Figure 1.18 EXAMPLES OF WINDOWS AND UNIX SYSTEM CALLS

System Programs

System programs, also known as system utilities, provide a convenient environment for program-development and execution. They can be divided into these categories:

- **File management**. These programs create, delete, copy, rename, print, dump, list, and generally manipulate files and directories.
- **Status information**. Some programs simply ask the system for the date, time, amount of available memory or disk space, number of users, or similar status information. Others are more complex, providing detailed performance, logging, and debugging information. Typically, these programs format and print the output to the terminal or other output devices or files or display it in a window of the GUI. Some systems also support a registry which is used to store and retrieve configuration information.
- **File modification**. Several text editors may be available to create and modify the content of files stored on disk or other storage devices.
- **Programming-language support**. Compilers, assemblers, debuggers, and interpreters for common programming languages (such as C, C++, Java, Visual Basic, and PERL) are often provided to the user with the operating system.
- **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 electronic-mail messages, to log in remotely, or to transfer files from one machine to another.

In addition to systems programs, most operating systems are supplied with programs that are

useful in solving common problems or performing common operations. Such programs include web browsers, word processors and text formatters, spreadsheets, database systems, compilers, plotting and statistical-analysis packages, and games. These programs are known as **system utilities** or **application programs.**

Operating-System Design and Implementation

1. **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 shared, single user, multiuser, distributed, real time, or general purpose.
   - The requirements can be divided into two basic groups: **user goals and systemgoals**.
   - User needs: The system should be convenient to use, easy to learn and to use, reliable, safe, and fast.
   - System needs (designers): The system should be easy to design, implement, and maintain; it should be flexible, reliable, error free, and efficient.
   - Specifying and designing an operating system is a highly creative task. Although no textbook can tell you how to do it, general principles have been developed in the field of **software engineering.**
2. **Mechanisms and Policies**
   - 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.
   - Microkernel-based operating systems take the separation of mechanism and policy to one extreme by implementing a basic set of primitive building blocks. These blocks are almost policy free, allowing more advanced mechanisms and policies to be added via user-created kernel modules or via user programs themselves.
   - 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.

3. **Implementation**
   - Traditionally, operating systems have been written in **assembly language**. Now, however, they are most commonly written in **higher-level languages** such as **C or C++.**
   - Master Control Program (MCP) was written in a variant of ALGOL. MULTICS, developed at MIT. The Linux and Windows XP OS are written in **C**.
   - The **advantages** of using a higher-level language:
   - **First**, the code can be **writtenfaster**, is more **compact**, and is **easier** to **understand** and **debug**.
   - **Second**, improvements in compiler technology will improve the generated code for the entire operating system by simple recompilation.
   - **Finally**, an operating system is far easier to *port*—to move to some other hardware— if it is written in a higher-level language.
   - For example, MS-DOS was written in Intel 8088 assembly language. Consequently, it is

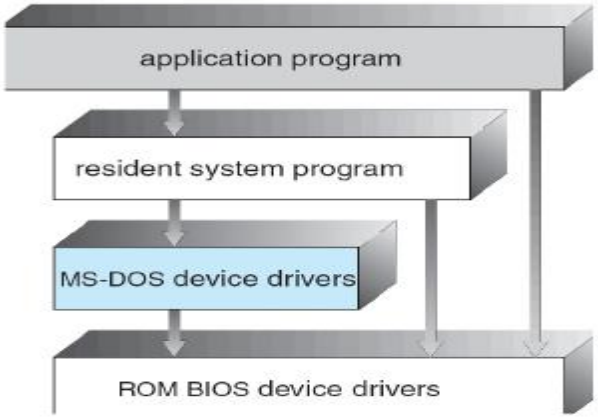
available on only the Intel family of CPUs. The Linux operating system, in contrast, is written in C and is available on a number of different CPUs, including Intel 80X86, Motorola 680X0, SPARC, and MIPS RX000.

- The only possible **disadvantages** of implementing an operating system in a higher-level language are **reduced speed** and **increased storage** requirements.
- Major performance improvements in operating systems are more likely to be the result of better data structures and algorithms than of excellent assembly-language code.
- To **identify bottlenecks**, we must be able to monitor system performance. Code must be added to compute and display measures of system behavior.
- The operating system does this task by producing trace listings of system behavior. All interesting events are **logged** with their time and important parameters and are written to a file. These same traces can be run as input for a simulation of a suggested improved system. Traces also can help people to find errors in operating-system behavior.

Operating-System Structure

1. Simple Structure

- Many commercial systems do not have well-defined structures; such operating 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 written to provide the most functionality in the least space, so it was not divided into modules carefully.
- In MS-DOS, the interfaces and levels of functionality are not well separated.
- MS-DOS was also **limited** by the **hardware** of its era. Because the Intel 8088 for which it was written provides no dual mode and no hardware protection, the designers of MS-DOS had no choice but to leave the base hardware accessible.
- Another example of limited structuring is the original **UNIX** operating system.
- UNIX is another system that initially was limited by hardware functionality. It consists of two separable parts: the kernel and the system programs.
- The kernel provides the file system, CPU scheduling, memory management, and other operating-system functions through system calls.



2. Layered Approach

- With proper hardware support, operating systems can be **broken into pieces that are smaller** and more appropriate than those allowed by the original MS-DOS or UNIX systems. The operating system can then **retain** much **greater control** over the computer and over the applications that make use of that computer.
- A system can be made modular in many ways. One method is the **layered approach,** in which the operating system is broken up into a number of layers (levels). The bottom

layer (layer 0) is the hardware; the highest (layer N) is the user interface. This layering structure is depicted in Figure 1.19.

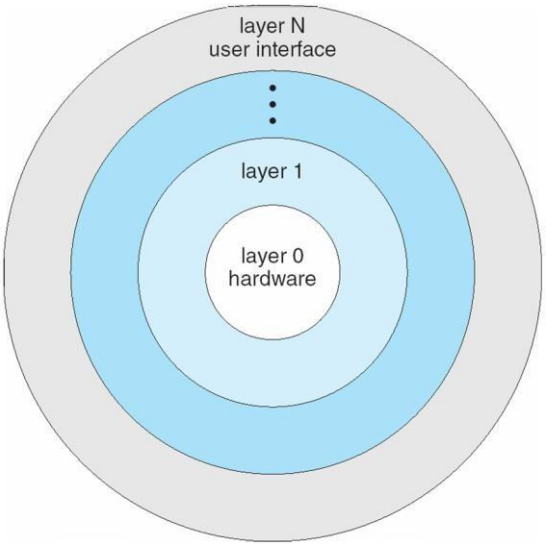


Figure 1.19A layered operating system.

- An operating-system layer is an implementation of an abstract object made up of data and the operations that can manipulate those data. A typical operating-system layer—say, layer M—consists of data structures and a set of routines that can be invoked by higher-level layers. Layer *M*, in turn, can invoke operations on lower-level layers.
- The main **advantage** of the layered approach is simplicity of construction and debugging. The first layer can be debugged without any concern for the rest of the system, it uses only the basic hardware (which is assumed correct) to implement its functions. Once the first layer is debugged, its correct functioning can be assumed while the second layer is debugged, and so on. If an error is found during the debugging of a particular layer, the error must be on that layer, because the layers below it are already debugged. Thus, the design and implementation of the system is simplified.
- Each layer is implemented with only those operations provided by lower level layers.
- The **majordifficulty** with the layered approach involves appropriately defining the various layers.A final problem is that they tend to be less efficient than other types.

3. Microkernels

- **Microkernel** 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, microkernels provide minimal **process** and **memory** management, in addition to a **communication** facility.
- **Advantages** of the microkernel approach are; ease of **extending the operating system**. All new services are added to user space and consequently do not require modification of the kernel.
- When the kernel needs to be modified, the changes tend to be fewer, because the microkernel is a smaller kernel.
- The resulting operating system is easier to port from one hardware design to another.
- The microkernel also provides more security and reliability, since most services are running as user—rather than kernel—processes.
- If a service fails, the rest of the operating system remains untouched.
- Example: Tru64 UNIX, Mach, QNX.

- **Disadvantage**: microkernels can suffer from **decreasedperformance** due to increased system function overhead. Example: Windows NT.

4. Modules

- Current methodology for operating-system design involves using object-oriented programming techniques to create a modular kernel. Such a strategy uses dynamically loadable modules and is common in modern implementations of UNIX, such as Solaris, Linux, and Mac OS X.

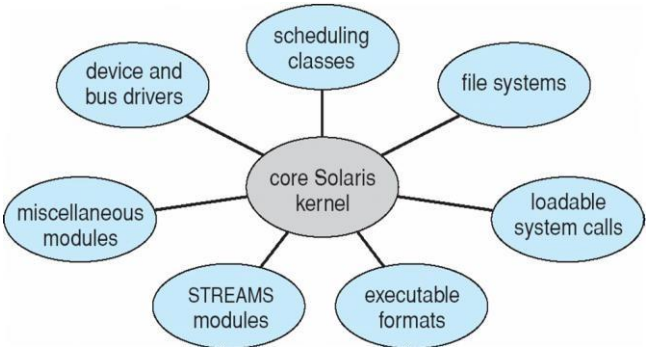


Figure 1.20Solaris loadable modules.

- For example, the Solaris operating system structure, shown in Figure 1.20, is organized around a core kernel with seven types of loadable kernel modules:
  - Scheduling classes
  - File systems
  - Loadable system calls
  - Executable formats
  - STREAMS modules
  - Miscellaneous
  - Device and bus drivers
- Such a design allows the kernel to provide core services yet also allows certain features to be implemented dynamically.
- The overall result resembles a layered system in that each kernel section has defined, protected interfaces; but it is more flexible than a layered system in that any module can call any other module. Furthermore, the approach is like the microkernel approach in that the primary module has only core functions and knowledge of how to load and communicate with other modules; but it is more efficient, because modules do not need to invoke message passing in order to communicate.
- The Apple Macintosh Mac OS X operating system uses a hybrid structure. Mac OS X structures the operating system using a layered technique where one layer consists of the Mach microkernel.

Virtual Machines

- The fundamental idea behind a virtual machine is to abstract the hardware of a single computer (the CPU, memory, disk drives, network interface cards, and so forth) into several different execution environments, thereby creating the illusion that each separate execution environment is running its own private computer.
- The virtual machine provides an interface that is *identical* to the underlying bare hardware.
- Each **guest** process is provided with a (virtual) copy of the underlying computer (Figure 1.21).
- Usually, the guest process is in fact an operating system, and that is how a single physical

machine can run multiple operating systems concurrently, each in its own virtual machine.

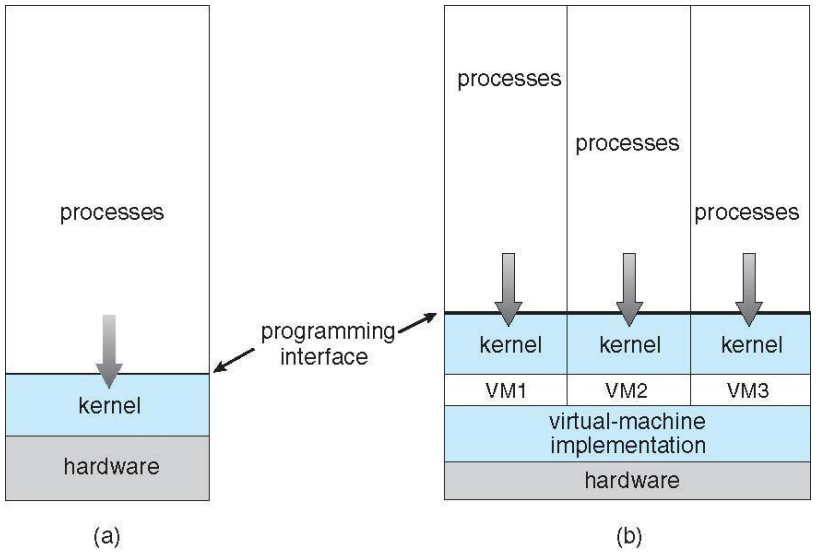


Figure 1.21 System models. (a) Nonvirtual machine. (b) Virtual machine.

1. **History**

- Virtual machines (VM) first appeared commercially on IBM mainframes via the VM operating system in 1972. VM has evolved and is still available, and many of the original concepts are found in other systems, making this facility worth exploring.
- Once these virtual machines were created, users could run any of the operating systems or software packages that were available on the underlying machine. For the IBM VM system, a user normally ran CMS-a single-user interactive operating system.

2. **Benefits**

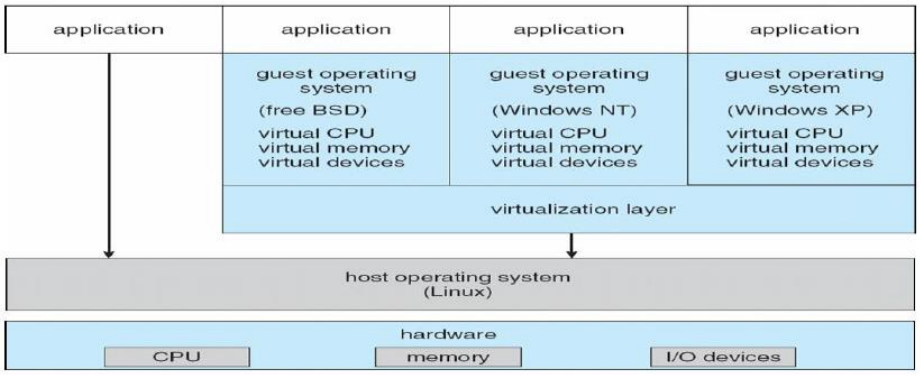
- Using VM we are able to share the same hardware yet run several different execution environments (that is, different operating systems) concurrently.
- **Advantages**: **First**, The host system is protected from the virtual machines, just as the virtual machines are protected from each other. A virus inside a guest operating system might damage that operating system but is unlikely to affect the host or the other guests.
- **Second,**A virtual-machine system is a perfect vehicle for operating-systems research and development.
- **Third** advantage of virtual machines for developers is that multiple operating systems can be running on the developer's workstation concurrently.
- **Four**, Virtualized workstation allows for rapid porting and testing of programs in varying environments. Similarly, quality-assurance engineers can test their applications in multiple environments without buying, powering, and maintaining a computer for each environment.
- **Fifth** advantage of virtual machines in production data-center use is system **consolidation**, which involves taking two or more separate systems and running them in virtual machines on one system.

3. Implementation

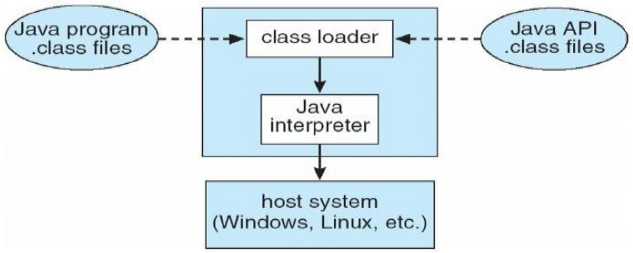
- Although the virtual-machine concept is useful it is difficult to implement. Much work is required to provide an *exact* duplicate of the underlying machine.
- Underlying machine typically has two modes: user mode and kernel mode.
- The virtual-machine software can run in kernel mode, since it is the operating system.
- The virtual machine itself can execute in only user mode. Just as the physical machine has two modes, however, so must the virtual machine.
- Consequently, we must have a virtual user mode and a virtual kernel mode, both of which run in a physical user mode. Those actions that cause a transfer from user mode to kernel mode on a real machine must also cause a transfer from virtual user mode to virtual kernel mode on a virtual machine.
- Such a transfer can be accomplished as follows.
- When a system call for example, is made by a program running on a virtual machine in virtual user mode, it will cause a transfer to the virtual-machine monitor in the real machine.
- When the virtual-machine monitor gains control it can change the register contents and program counter for the virtual machine to simulate the effect of the system call. It can then restart the virtual machine, noting that it is now in virtual kernel mode.
- The major difference, of course, is time. Whereas the real I/O might have taken 100 milliseconds, the virtual I/O might take less time (because it is spooled) or more time (because it is interpreted).
- Without some level of hardware support, virtualization would be impossible.

4. Examples

- VM-Ware :



- JVM:



Operating System Generation

- Operating systems are designed to run on any of a class of machines; the system must be configured for each specific computer site.
- SYSGEN program obtains information concerning the specific configuration of the hardware system.
- *Booting* – starting a computer by loading the kernel.
- *Bootstrap program* – code stored in ROM that is able to locate the kernel, load it into memory, and start its execution.

System Boot

- Operating system must be made available to hardware so hardware can start it.
- Small piece of code – **bootstrap loader**, locates the kernel, loads it into memory, and starts it.
- Sometimes two-step process where **boot block** at fixed location loads bootstrap loader.
- When power initialized on system, execution starts at a fixed memory location Firmware used to hold initial boot code.

Process Management

Process is a program in execution. Process contains,

- **Text section** – this contains program code and **Current activity** – as represented by the value of the **program counter** and the contents of the processor's **registers**.
- **Stack section** – contains temporary data (such as function parameters, return addresses, and local variables).
- **Data section** – contains global variables.
- **Heap** – This is a memory that is dynamically allocated during process run time.

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),** whereas 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.)

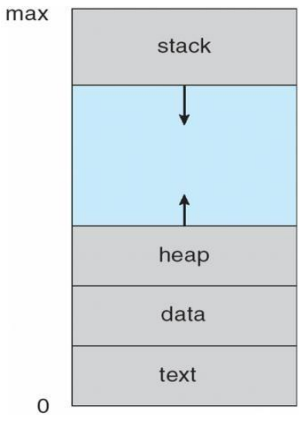


Figure 2.1 Process in memory

Process State

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

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

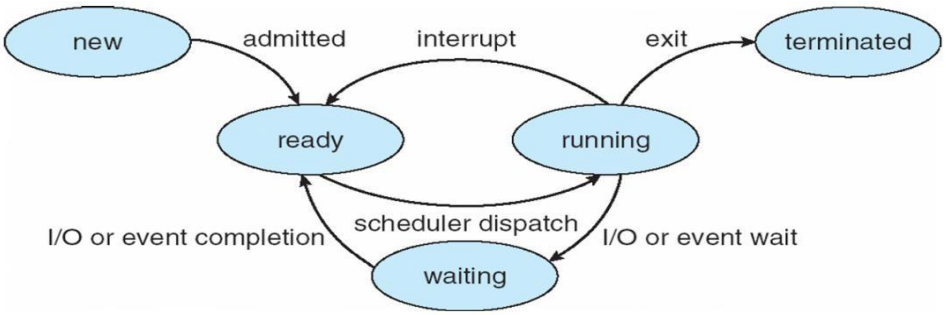


Figure 2.2 Diagram of process state.

Process (task) Control Block

Each process is represented in the operating system by a **process control block (PCB)**—also called a **task control block**. A PCB is shown in Figure 2.3.

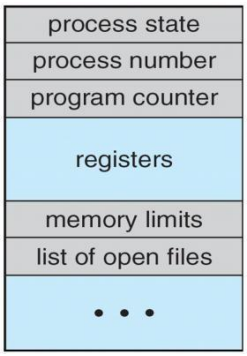


Figure 2.3 Process control block (PCB).

- **Process state.** The state may be new, ready, running, waiting, halted, and so on.
- **Program counter.** The counter indicates the address of the next instruction to be executed for this process.
- **CPU registers.** The registers vary in number and type, depending on the computer architecture. They include accumulators, index registers, stack pointers, and general-purpose registers, plus any condition-code information. Along with the program counter, this state information must be saved when an interrupt occurs, to allow the process to be continued correctly afterward (Figure 2.4).
- **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 information as the value of the base and limit registers, 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 members, 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.

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.

1. Scheduling Queues

- As processes enter the system, they are put into a **job queue,** which consists of all processes in the system.
- The processes that are residing in main memory and are ready and waiting to execute are kept on a list called the **ready queue.** This queue is generally stored as a linked list. A ready-queue header contains pointers to the first and final PCBs in the list. Each PCB includes a pointer field that points to the next PCB in the ready queue.
- When a process is allocated the CPU, it executes for a while and eventually quits, is interrupted, or waits for the occurrence of a particular event, such as the completion of an I/O request. The list of processes waiting for a particular I/O device is called a **device queue**.
- A common representation of process scheduling is a **queuing diagram**, such as that in Figure 2.6.
- Each **rectangular box** represents a **queue**. The **circles** represent the **resources** that serve the queues, and the **arrows** indicate the **flow of processes** in the system.

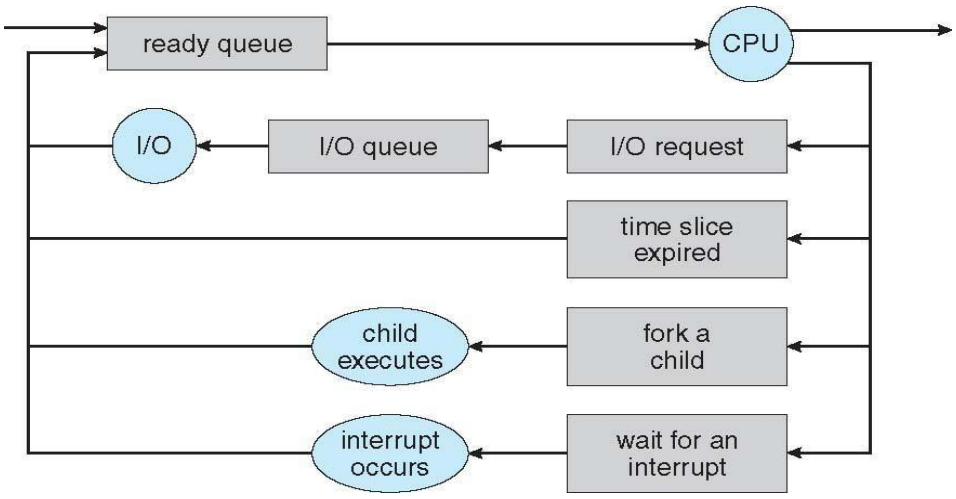


Figure 2.6 Queuing-diagram representation of process scheduling.

- A new process is initially put in the ready queue. It waits there until it is selected for execution, or is 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 sub process and wait for the sub process'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.

2. Schedulers

- A process migrates among the various scheduling queues throughout its lifetime. The selection process is carried out by the OS by an appropriate scheduler algorithm.
- The processes are spooled to a mass-storage device, where they are kept for later execution.
- The **long-term scheduler**, or job scheduler, selects processes from this pool and loads them into memory for execution. The **short-term scheduler**, or CPU scheduler, selects from among the processes that are ready to execute and allocates the CPU to one of them.
- The primary distinction between these two schedulers lies in frequency of execution.
- The short-term scheduler must select a new process for the CPU **frequently**. A process may execute for only a few milliseconds before waiting for an I/O request.
- The long-term scheduler executes much **less frequently**; minutes may separate the creation of one new process and the next. The long-term scheduler controls the degree of multiprogramming (the number of processes in memory).
- An I/O bound process is one that spends more of its time doing I/O than it spends doing computations. A CPU-bound process, in contrast, generates I/O requests infrequently, using more of its time doing computations.
- Some operating systems, such as **time-sharing systems**, may introduce an additional, intermediate level of scheduling. This **medium-term scheduler** is shown in Figure 2.7.
- The key idea behind a medium-term scheduler is that sometimes it can be advantageous to remove processes from memory 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.

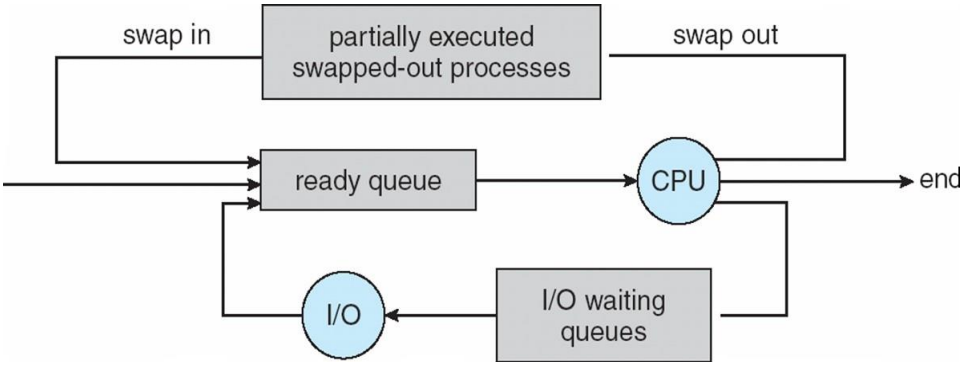


Figure 2.7 Addition of medium-term scheduling to the queuing diagram.

3. 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 2.4 shows the context switch from one process to other.

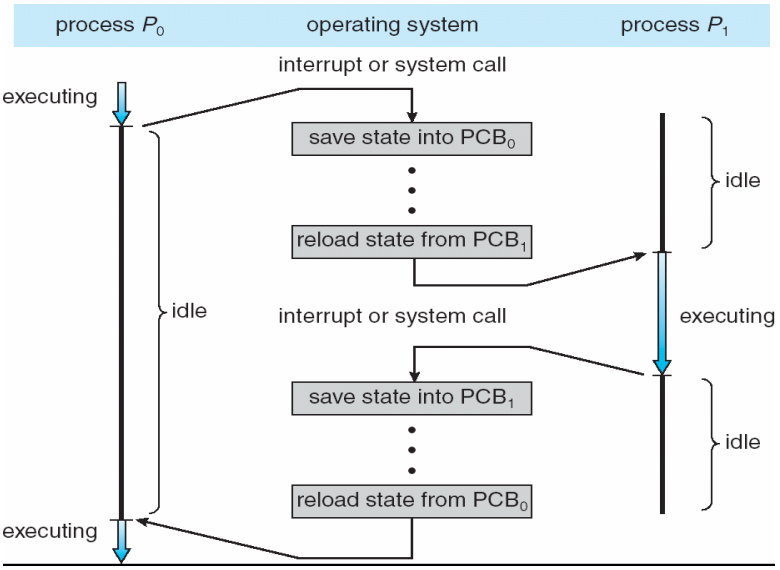


Figure 2.4 Diagram showing CPU switch from process to process.

Operations on Processes

1. Process Creation

- A process may create several new processes, via a create-process system call, during the course of execution. 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.
- Operating systems identify processes according to a unique **process identifier (pid)**, which is typically an **integer** number.
- A process will need certain **resources** (CPU time, memory, files, I/0 devices) to accomplish its task.
- When a process creates a **subprocess**, that **subprocess** may be able to obtain its resources
    1. directly from the operating system,
    2. it may be constrained to a subset of the resources of the parent process.
- In addition to the various physical and logical resources that a process obtains when it is created, initialization data (input) may be passed along by the parent process to the child process.
- When a process creates a new process, two possibilities exist in terms of execution:
    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 possibilities in terms of the address space of the new process:
    1. The child process is a duplicate of the parent process (it has the same program and data as the parent).
    2. The child process has a new program loaded into it.
- In UNIX, a new process is created by the **fork()** system call. The new process consists of a copy of the address space of the original process. This mechanism allows the parent process to communicate easily with its child process. Both processes (the parent and the child) continue execution at the instruction after the fork(), with one difference: the return code for the fork() is zero for the new (child) process, whereas the (nonzero) process identifier of the child is returned to the parent.
- The **exec()** system call is used after a fork() system call by one of the two processes to

replace the process's memory space with a new program.

- The parent can then create more children; or, if it has nothing else to do while the child runs, it can issue a**wait()** system call to move itself off the ready queue until the termination of the child.

```
#include <sys/types.h>
#include <stdio.h>
#include <unistd.h> int
main()
{
        pid_t pid;
        pid =fork();            /* fork a child process */
        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 */
                wait (NULL) ; /* parent will wait for the child to complete */
                printf("Child Complete");
        }
        return 0;
}
```

Figure 2.8 Creating a separate process using the UNIX fork() system call.

- 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.
- The parent waits for the child process to complete with the wait() system call.
- When the child process completes (by either implicitly or explicitly invoking exit ()) the parent process resumes from the call to wait (),where it completes using the exit() system call.
- This is also illustrated in Figure 2.9.

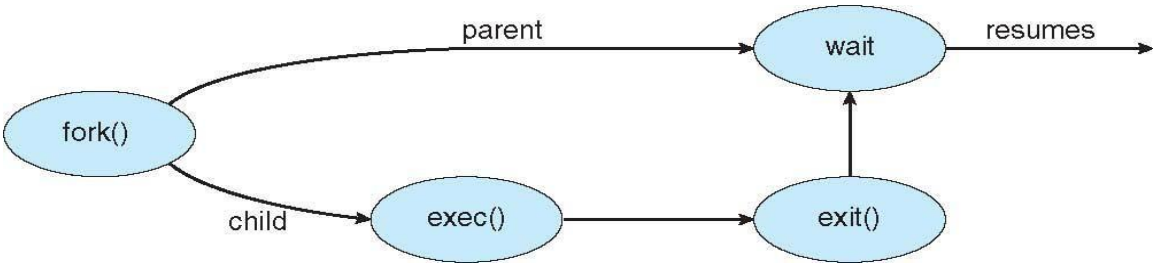


Figure 2.9 Process creation using fork() system call.

2. 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.

- A parent may terminate the execution of one of its children for a variety of reasons, such as these:
    1. 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.)
    2. The task assigned to the child is no longer required.
    3. The parent is exiting, and the operating system does not allow a child to continue if its parent terminates.
- If a parent 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.
- In UNIX, we can terminate a process by using the exit() system call; its parent process may wait for the termination of a child process by using the wait() system call.
- The wait() system call returns the process identifier of a terminated child so that the parent can tell which of its children has terminated.

Inter-process Communication

There are two types of processes. A process is **independent** if it cannot affect or be affected by the other processes executing in the system. A process is **cooperating** if it can affect or be affected by the other processes executing in the system.

Reasons for providing an environment that allows process cooperation:

- **Information sharing**. Since several users may be interested in the same piece of information (for instance, a shared file), we must provide an environment to allow concurrent access to such information.
- **Computation speedup**. If we want a particular task to run faster, we must break it into subtasks, each of which will be executing in parallel with the others. Notice that such a speedup can be achieved only if the computer has multiple processing elements (such as CPUs or I/O channels).
- **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, printing, and compiling in parallel.

There are **two** fundamental models of interprocess communication in cooperating processes:
(1) **shared memory** and (2) **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.

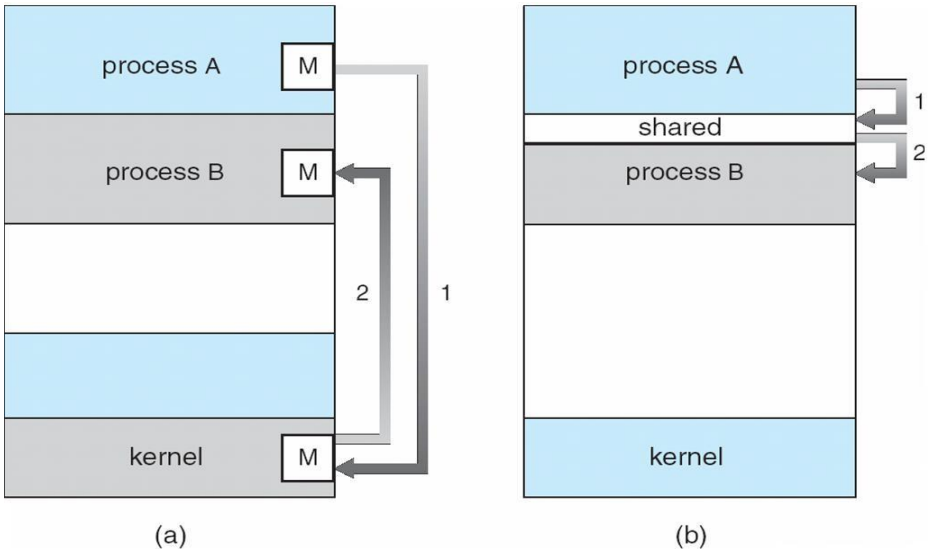


Figure 2.10Communications models. (a) Message passing. (b) Shared memory.

1. Shared-Memory Systems

- Interprocess communication using shared memory requires communicating processes to establish a region of shared memory.
- Typically, a shared-memory region resides in the address space of the process creating the shared memory segment.
- Other processes that wish to communicate using this shared memory segment must attach it to their address space.
- They can then exchange information by reading and writing data in the shared areas.
- Example: **Producer Consumer Problem**.
- A producer process produces information that is consumed by a consumer process.
- **Example**, a compiler may produce assembly code, which is consumed by an assembler. The assembler, in turn, may produce object modules, which are consumed by the loader.
- **Another example** is, 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.
- 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.
- **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.
- The following variables reside in a region of memory shared by the producer and consumer processes:

```
#define BUFFER_SIZE 10
typedef struct {
        ….
}item;
item buffer[BUFFER_SIZE];
int in = 0;
int out = 0;
```

Figure 2.11 Variable in shared memory

- The shared buffer is implemented as a circular array with two logical pointers: **in and out**. The variable in points to the next free position in the buffer; out points to the first full position in the buffer. The buffer is empty when in==out; the buffer is full when ((in+1)%BUFFER_SIZE) == out.
- The code for the producer and consumer processes is shown in Figures 2.12 and 2.13 respectively.

```
while (true) {
  /* Produce an item */
     while (((in + 1) % BUFFER SIZE) == out)
          ; /* do nothing -- no free buffers */
          buffer[in] = item;
          in = (in + 1) % BUFFER SIZE;
}
```

Figure 2.12 Producer Process code

```
while (true) {
while (in == out)
   ; // do nothing -- nothing to consume
    // remove an item from the buffer
    item = buffer[out];
    out = (out + 1) % BUFFER SIZE;
    return item;
 }
```

Figure 2.13Consumer Process code

2. Message-Passing Systems

- Message passing provides a mechanism to allow processes to communicate and to synchronize their actions without sharing the same address space and is particularly useful in a distributed environment, where the communicating processes may reside on different computers connected by a network.
- For example, a **chat program** used on the World Wide Web 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)** and **receive(message)**. Messages sent by a process can be of either fixed or variable size.
- 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.
  - Direct or indirect communication
  - Synchronous or asynchronous communication
  - Automatic or explicit buffering

2.1 Naming

- Under **direct communication**, each process that wants to communicate must explicitly name the recipient or sender of the communication. In this scheme, the send() and receive() primitives are defined as:
  - send(P, message) -Send a message to process P.
  - receive (Q, message)-Receive a message from process Q.
- A communication link in this scheme has the following properties:
  - A link is established automatically between every pair of processes that want to communicate. The processes need to know only each other's identity to communicate.
  - A link is associated with exactly two processes.
  - Between each pair of processes, there exists exactly one link.
- This scheme exhibits ***symmetry*** in addressing; that is, both the sender process and the receiver process must name the other to communicate.
- A variant of this scheme employs ***asymmetry*** in addressing. Here, only the sender names the recipient; the recipient is not required to name the sender. In this scheme, the send() and receive() primitives are defined as follows:
  - send(P, message) -Send a message to process P.
  - receive (id, message) -Receive a message from any process; the variable *id* is set to the name of the process with which communication has taken place.
- The **disadvantage** in both of these schemes (symmetric and asymmetric) is the **limited modularity** of the resulting process definitions. Changing the identifier of a process may necessitate examining all other process definitions. Such a hard coding is less desirable.
- With **indirect communication**, the messages are sent to and received from **mailboxes**, or **ports**. Each mailbox has a **unique identification**. For example, POSIX message queues use an integer value to identify a mailbox. Two processes can communicate only if the processes have a shared mailbox.
- The send() and receive 0 primitives are defined as follows:
  - send (A, message) -Send a message to mailbox A.
  - receive (A, message)-Receive a message from mailbox A.
- In this scheme, a communication link has the following properties:
  - A link is established between a pair of processes only if both members of the pair have a shared mailbox.
  - A link may be associated with more than two processes.
  - Between each pair of communicating processes, there may be a number of different links, with each link corresponding to one mailbox.

2.2 Synchronization

Message passing may be either blocking or non-blocking also known as synchronous and asynchronous.

- **Blocking send**. The sending process is blocked until the message is received by the receiving process or by the mailbox.
- **Non-blocking send**. The sending process sends the message and resumes operation.

- **Blocking receive**. The receiver blocks until a message is available.
- **Non-blocking receive**. The receiver retrieves either a valid message or a null.

When both send() and receive() are blocking, we have a rendezvous between the sender and the receiver.

2.3 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. 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, and the sender can continue execution without waiting. 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.