**OPERATING SYSTEMS**

(Common CSE(AI&ML) & CSE(DS))

**UNIT-1**

**Syllabus:**  **Introduction**: Operating system, functions of operating system, types of operating system, computer system organization, computer system architecture, operating system structure, operating system operations, computing environments, open source operating systems.

**Operating System Structures:** operating system services, system calls, types of system calls, system programs, operating system structure, operating system debugging and system boot

**INTRODUCTION AND SYSTEM STRUCTURE**

**Introduction:**

An **operating system** is a program that manages a computer’s hardware. It also provides a basis for application programs and acts as an intermediary between the computer user and the computer hardware. operating systems are designed to be ***convenient,*** others to be ***efficient,*** and others to be some combination of the two.

**Operating system goals**

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

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

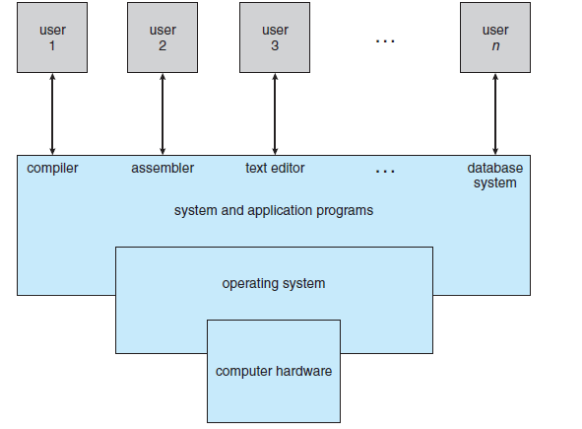


Figure 1: Abstract view of the components of the computer system

The **hardware**—the **central processing unit (CPU)**, the **memory**, and the **input/output (I/O) devices**—provides the basic computing resources for the system. The **application programs**—such as word processors, spreadsheets, compilers, and Web browsers—define the ways in which these resources are used to solve users’ computing problems. The operating system controls the hardware and coordinates its use among the various application programs for the various users. The operating system provides the means for proper use of these resources in the operation of the computer system. operating systems can be explored from two viewpoints: that of the user and that of the system.

**User View**

The user’s view of the computer varies according to the interface being used. Most computer users sit in front of a PC, consisting of a monitor, keyboard, mouse, and system unit. Such a system is designed for one user to monopolize its resources.

The goal is to maximize the work (or play) that the user is performing. In this case, the operating system is designed mostly for **ease of use**, with some attention paid to performance and none paid to **resource utilization**—how various hardware and software resources are

shared.

In other cases, a user sits at a terminal connected to a **mainframe** or a **minicomputer**. Other users are accessing the same computer through other terminals. These users share resources and may exchange information. The operating system in such cases is designed to maximize resource utilization

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

The user interface for mobile computers generally features a **touch screen**, 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.

**Functions of operating system**

**Process Management**

A *process* is a program in execution. A process needs certain resources, including CPU time, memory, files, and I/O devices, to accomplish its task. The operating system is responsible for the following activities in connection with process management.

✦ Process creation and deletion.

✦ Process suspension and resumption.

✦ Provision of mechanisms for:

• process synchronization

• process communication

**Main-Memory Management**

Memory is a large array of words or bytes, each with its own address. It is a repository of quickly accessible data shared by the CPU and I/O devices. Main memory is a volatile storage device. It loses its contents in the case of system failure. The operating system is responsible for the following activities in connections with memory management:

♦ Keep track of which parts of memory are currently being used and by whom.

♦ Decide which processes to load when memory space becomes available.

♦ Allocate and de-allocate memory space as needed.

**File Management**

A file is a collection of related information defined by its creator. Commonly, files represent programs (both source and object forms) and data. The operating system is responsible for the following activities in connections with file management:

✦ File creation and deletion.

✦ Directory creation and deletion.

✦ Support of primitives for manipulating files and directories.

✦ Mapping files onto secondary storage.

✦ File backup on stable (nonvolatile) storage media.

**I/O System Management**

The I/O system consists of:

✦ A buffer-caching system

✦ A general device-driver interface

✦ Drivers for specific hardware devices

**Secondary-Storage Management**

Since main memory (*primary storage*) is volatile and too small to accommodate all data and programs permanently, the computer system must provide *secondary storage* to back up main memory. Most modern computer systems use disks as the principle on-line storage medium, for both programs and data. The operating system is responsible for the following activities in connection with disk management:

✦ Free space management

✦ Storage allocation

✦ Disk scheduling

**Networking (Distributed Systems)**

♦ A *distributed* system is a collection processors that do not share memory or a clock. Each processor has its own local memory.

♦ The processors in the system are connected through a communication network.

♦ Communication takes place using a *protocol.*

♦ A distributed system provides user access to various system resources.

♦ Access to a shared resource allows:

✦ Computation speed-up

✦ Increased data availability

✦ Enhanced reliability

**Protection System**

♦ *Protection* refers to a mechanism for controlling access by programs, processes, or users to both system and user resources.

♦ The protection mechanism must:

✦ distinguish between authorized and unauthorized usage.

✦ specify the controls to be imposed.

✦ provide a means of enforcement.

**Command-Interpreter System**

• Many commands are given to the operating system by control statements which deal with:

✦ process creation and management

✦ I/O handling

✦ secondary-storage management

✦ main-memory management

✦ file-system access

✦ protection

✦ networking

• The program that reads and interprets control statements is called variously:

✦ command-line interpreter

✦ shell (in UNIX)

• Its function is to get and execute the next command statement.

**Operating-System Structures**

• System Components

• Operating System Services

• System Calls

• System Programs

• System Structure

• Virtual Machines

• System Design and Implementation

• System Generation

**Common System Components**

• Process Management

• Main Memory Management

• File Management

• I/O System Management

• Secondary Management

• Networking

• Protection System

• Command-Interpreter System

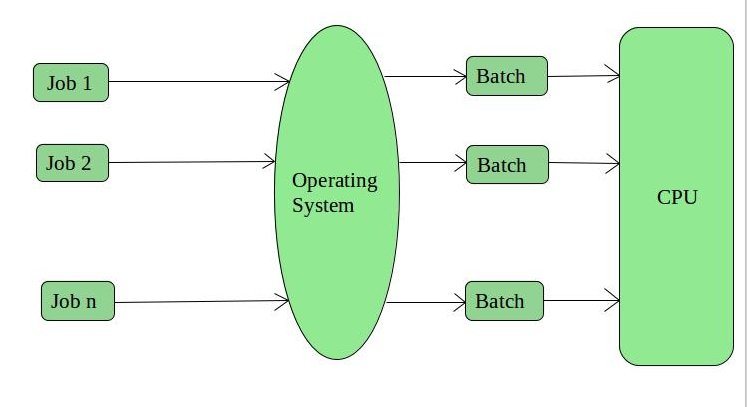
Types of Operating Systems

An Operating System performs all the basic tasks like managing files, processes, and memory. Thus operating system acts as the manager of all the resources, i.e. **resource manager**. Thus, the operating system becomes an interface between user and machine.

Following are the popular types of OS (Operating System):

1. **Batch Operating System –**

This type of operating system does not interact with the computer directly. There is an operator which takes similar jobs having the same requirement and group them into batches. It is the responsibility of the operator to sort jobs with similar needs.

* 

**Advantages of Batch Operating System:**

* It is very difficult to guess or know the time required for any job to complete. Processors of the batch systems know how long the job would be when it is in queue
* Multiple users can share the batch systems
* The idle time for the batch system is very less
* It is easy to manage large work repeatedly in batch systems

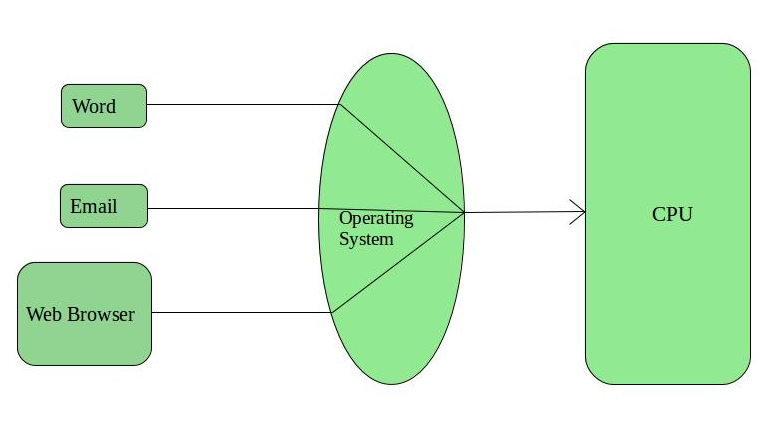
**Disadvantages of Batch Operating System:**

* The computer operators should be well known with batch systems
* Batch systems are hard to debug
* It is sometimes costly
* The other jobs will have to wait for an unknown time if any job fails

**Examples of Batch based Operating System:** Payroll System, Bank Statements, etc.

**2. Time-Sharing Operating Systems –**

Each task is given some time to execute so that all the tasks work smoothly. Each user gets the time of CPU as they use a single system. These systems are also known as Multitasking Systems. The task can be from a single user or different users also. The time that each task gets to execute is called quantum. After this time interval is over OS switches over to the next task.



**Advantages of Time-Sharing OS:**

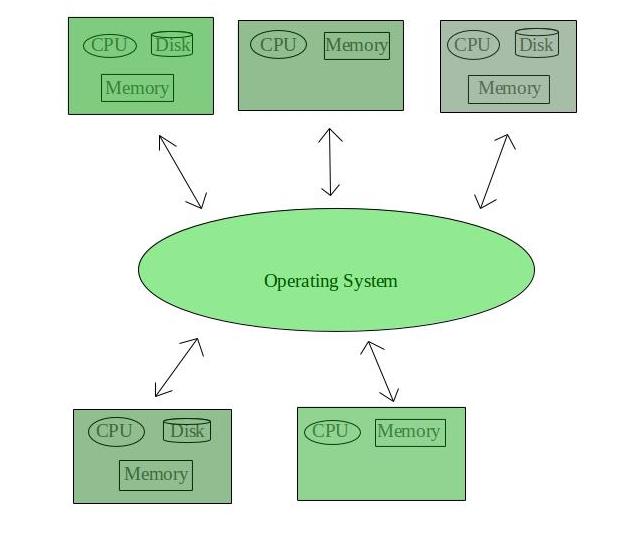
* Each task gets an equal opportunity
* Fewer chances of duplication of software
* CPU idle time can be reduced

**Disadvantages of Time-Sharing OS:**

* Reliability problem
* One must have to take care of the security and integrity of user programs and data
* Data communication problem

**Examples of Time-Sharing OSs are:** Multics, Unix, etc.

**3. Distributed Operating System –**   
These types of the operating system is a recent advancement in the world of computer technology and are being widely accepted all over the world and, that too, with a great pace. Various autonomous interconnected computers communicate with each other using a shared communication network. Independent systems possess their own memory unit and CPU. These are referred to as **loosely coupled systems** or distributed systems. These system’s processors differ in size and function. The major benefit of working with these types of the operating system is that it is always possible that one user can access the files or software which are not actually present on his system but some other system connected within this network i.e., remote access is enabled within the devices connected in that network. 



**Advantages of Distributed Operating System:**

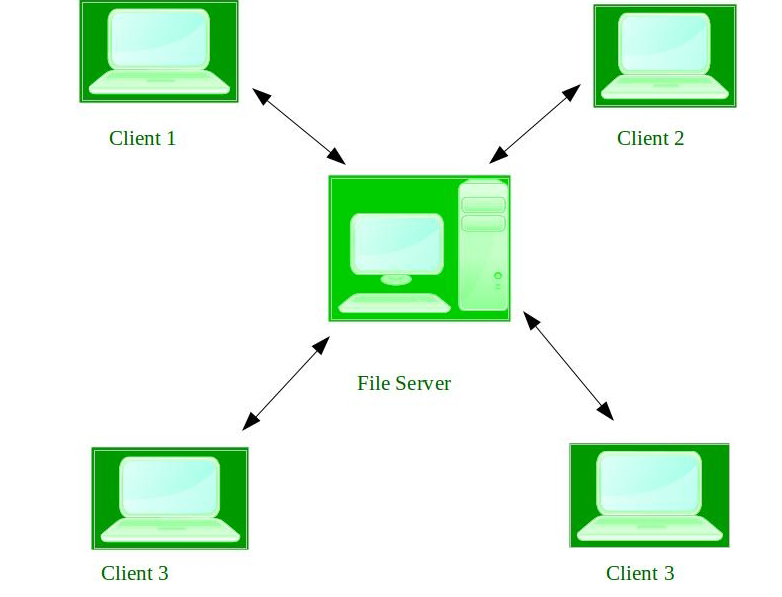
* Failure of one will not affect the other network communication, as all systems are independent from each other
* Electronic mail increases the data exchange speed
* Since resources are being shared, computation is highly fast and durable
* Load on host computer reduces
* These systems are easily scalable as many systems can be easily added to the network
* Delay in data processing reduces

**Disadvantages of Distributed Operating System:**

* Failure of the main network will stop the entire communication
* To establish distributed systems the language which is used are not well defined yet
* These types of systems are not readily available as they are very expensive. Not only that the underlying software is highly complex and not understood well yet

**Examples of Distributed Operating System are-** LOCUS, etc.

**4. Network Operating System –**   
These systems run on a server and provide the capability to manage data, users, groups, security, applications, and other networking functions. These types of operating systems allow shared access of files, printers, security, applications, and other networking functions over a small private network. One more important aspect of Network Operating Systems is that all the users are well aware of the underlying configuration, of all other users within the network, their individual connections, etc. and that’s why these computers are popularly known as **tightly coupled systems**.



**Advantages of Network Operating System:**

* Highly stable centralized servers
* Security concerns are handled through servers
* New technologies and hardware up-gradation are easily integrated into the system
* Server access is possible remotely from different locations and types of systems

**Disadvantages of Network Operating System:**

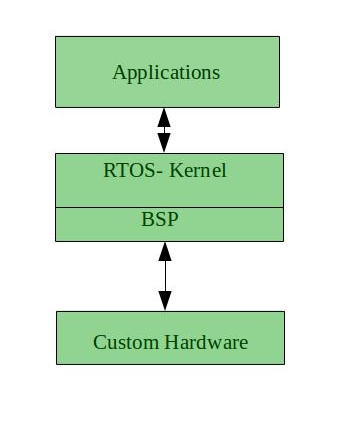
* Servers are costly
* User has to depend on a central location for most operations
* Maintenance and updates are required regularly

**Examples of Network Operating System are:** Microsoft Windows Server 2003, Microsoft Windows Server 2008, UNIX, Linux, Mac OS X, Novell NetWare, and BSD, etc.

**5. Real-Time Operating System –**   
These types of OSs serve real-time systems. The time interval required to process and respond to inputs is very small. This time interval is called **response time**.  **Real-time systems** are used when there are time requirements that are very strict like missile systems, air traffic control systems, robots, etc.

**Two types of Real-Time Operating System which are as follows:**

* **Hard Real-Time Systems:**   
  These OSs are meant for applications where time constraints are very strict and even the shortest possible delay is not acceptable. These systems are built for saving life like automatic parachutes or airbags which are required to be readily available in case of any accident. Virtual memory is rarely found in these systems.
* **Soft Real-Time Systems:**  These OSs are for applications where for time-constraint is less strict.



**Advantages of RTOS:**

* **Maximum Consumption:** Maximum utilization of devices and system, thus more output from all the resources
* **Task Shifting:** The time assigned for shifting tasks in these systems are very less. For example, in older systems, it takes about 10 microseconds in shifting one task to another, and in the latest systems, it takes 3 microseconds.
* **Focus on Application:** Focus on running applications and less importance to applications which are in the queue.
* Real-time**operating system in**the **embedded system:** Since the size of programs are small, RTOS can also be used in embedded systems like in transport and others.
* **Error Free:** These types of systems are error-free.
* **Memory Allocation:** Memory allocation is best managed in these types of systems.

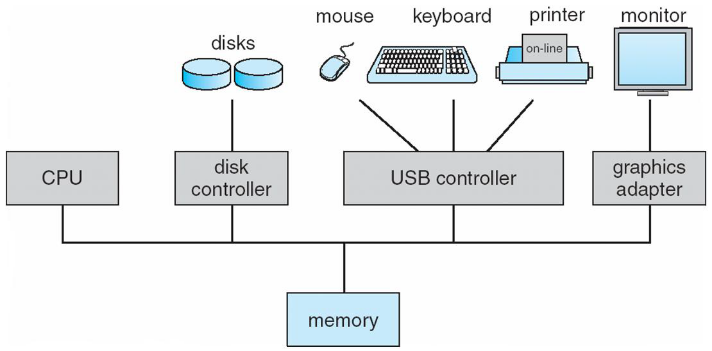
**Disadvantages of RTOS:**

* **Limited Tasks:** Very few tasks run at the same time and their concentration is very less on few applications to avoid errors.
* **Use heavy system resources:** Sometimes the system resources are not so good and they are expensive as well.
* **Complex Algorithms:** The algorithms are very complex and difficult for the designer to write on.
* **Device driver and interrupt signals:** It needs specific device drivers and interrupts signals to respond earliest to interrupts.
* **Thread Priority:** It is not good to set thread priority as these systems are very less prone to switching tasks.

**Examples of Real-Time Operating Systems are:** Scientific experiments, medical imaging systems, industrial control systems, weapon systems, robots, air traffic control systems, etc.

#### Computer System Organization

A modern general-purpose computer system consists of one or more CPUs and a number of device controllers connected through a common bus that provides access to shared memory (Figure 1.2). Each device controller is in charge of a specific type of device (for example, disk drives, audio devices, or video displays). The CPU and the device controllers can execute in parallel, competing for memory cycles. To ensure orderly access to the shared memory, a memory controller synchronizes access to the memory. For a computer to start running—for instance, when it is powered up or rebooted—it needs to have an initial program to run. This initial program, or **bootstrap program**, tends to be simple. Typically, it is stored within the computer hardware in read-only memory (**ROM**) or electrically erasable programmable read-only memory (**EEPROM**), known by the general term **firmware**. It initializes all aspects of the system, from CPU registers to device controllers to memory contents. The bootstrap program must know how to load the operating system and how to start executing that system. To accomplish this goal, the bootstrap program must locate the operating-system kernel and load it into memory.

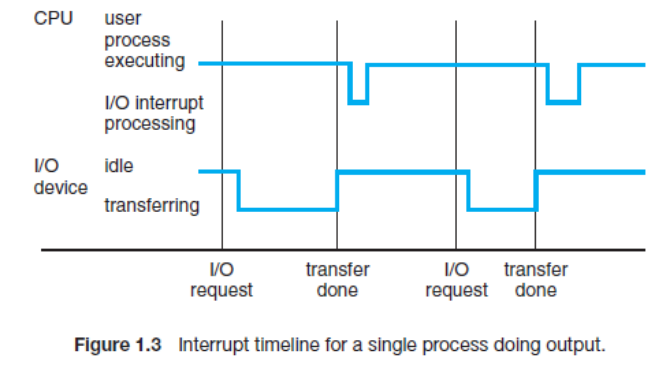


**Figure 1.2** A modern computer system.

Once the kernel is loaded and executing, it can start providing services to the system and its users. Some services are provided outside of the kernel, by system programs that are loaded into memory at boot time to become **system processes**, or **system daemons** that run the entire time the kernel is running

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

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 timeline of this operation is shown in Figure 1.3.



**1.2.2 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 (also called **random-access memory**, or **RAM**). Main memory commonly is implemented in a semiconductor technology called **dynamic random-access memory (DRAM)**. Computers use other forms of memory as well. We have already mentioned read-only memory, ROM) and electrically erasable programmable read-only memory, EEPROM).

A typical instruction–execution cycle, as executed on a system with a **vonNeumann architecture**, 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 the instruction on the operands has been executed, the result may be stored back in memory

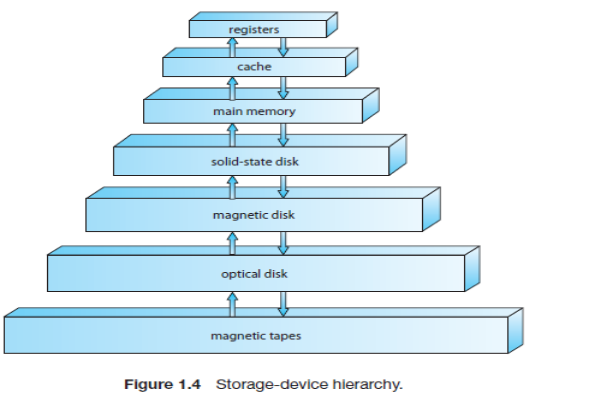
Ideally, we want the programs and data to reside in main memory permanently. This arrangement usually is not possible for the following two reasons:

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

**2.** Main memory is a **volatile** storage 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 main requirement for secondary storage is that it be able to hold large quantities of data permanently. The most common secondary-storage device is a **magnetic disk**, which provides storage for both programs and data. Most programs (system and application) are stored on a disk until they are loaded into memory

The wide variety of storage systems can be organized in a hierarchy (Figure 1.4) according to speed and cost. The higher levels are expensive, but they are fast. As we move down the hierarchy, the cost per bit generally decreases, whereas the access time generally increases. This trade-off is reasonable; if a given storage system were both faster and less expensive than another—other properties being the same—then there would be no reason to use the slower, more expensive memory.



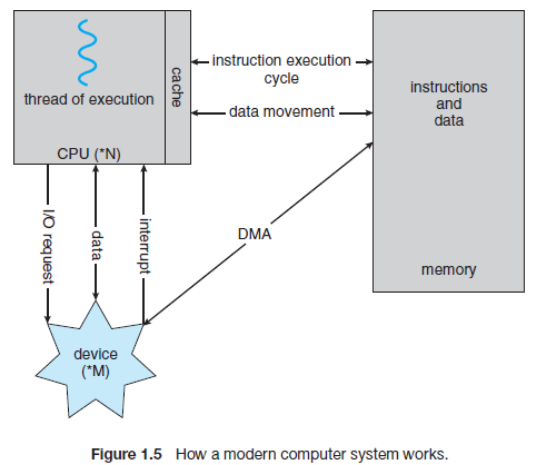
In addition to differing in speed and cost, the various storage systems are either volatile or nonvolatile. As mentioned earlier, **volatile storage** loses its contents when the power to the device is removed. In the absence of expensive battery and generator backup systems, data must be written to **nonvolatile storage** for safekeeping. In the hierarchy shown in Figure 1.4, the storage systems above the solid-state disk are volatile, whereas those including the solid-state disk and below are nonvolatile

* + 1. 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. Depending on the controller, more than one device may be attached. For instance, seven or more devices can be attached to the **small computer-systems interface (SCSI)** controller. A device controller maintains some local buffer storage and a set of special-purpose registers. The device controller is responsible for moving the data between the peripheral devices that it controls and its local buffer storage. Typically, operating systems have a **device driver** for each device controller. This device driver understands the device controller and provides the rest of the operating system with a uniform interface to the device.

interrupt-driven I/O is fine for moving small amounts of data but can produce high overhead when used for bulk data movement such as disk I/O. To solve this problem, **direct memory access (DMA)** is used. After setting up 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. Figure 1.5 shows the interplay of all components of a computer system.



**Computer system architecture**

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

1.3.1 Single-Processor Systems

most computer systems used a single processor. On a single processor system, there is one main CPU capable of executing a general-purpose instruction set, including instructions from user processes. Almost all single processor systems have other special-purpose processors as well. If there is only one general-purpose CPU, then the system is a single-processor system.

1.3.2 Multiprocessor Systems

**multiprocessor systems** (also known as **parallel systems** or **multicore systems**) have begun to dominate the landscape of computing. Such systems have two or more processors in close communication, sharing the computer bus and sometimes the clock, memory, and peripheral

devices.

Multiprocessor systems have three main advantages

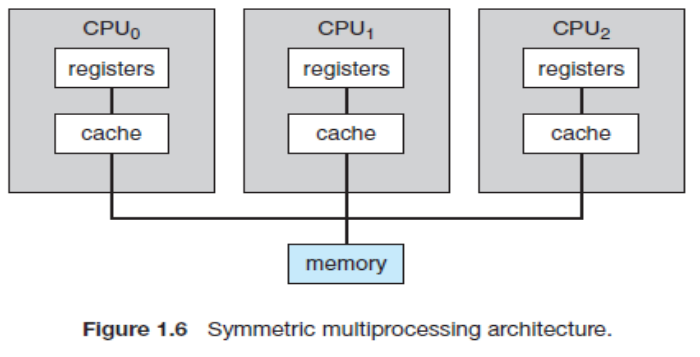
**Increased throughput**. By increasing the number of processors, we expect to get more work done in less time. .

**Economy of scale**.. If several programs operate on the same set of data, it is cheaper to store those data on one disk and to have all the processors share them than to have many computers with local disks and many copies of the data.

**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.

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

The most common systems use **symmetric multiprocessing (SMP)**, in which each processor performs all tasks within the operating system. SMP means that all processors are peers; no boss–worker relationship exists between processors. Figure 1.6 illustrates a typical SMP architecture.



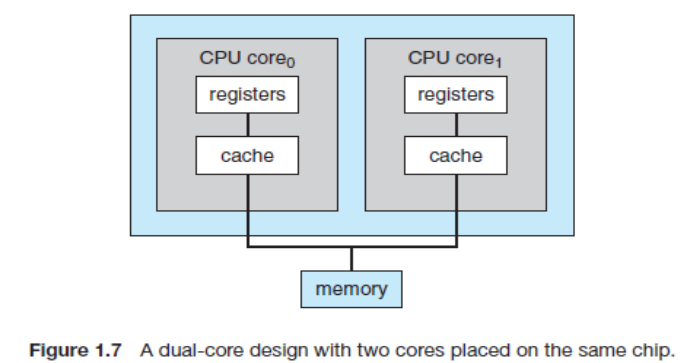
The difference between symmetric and asymmetric multiprocessing may result from either hardware or software. Special hardware can differentiate the multiple processors, or the software can be written to allow only one boss and multiple workers.

Multiprocessing adds CPUs to increase computing power. If the CPU has an integrated memory controller, then adding CPUs can also increase the amount of memory addressable in the system. Either way, multiprocessing can cause a system to change its memory access model from uniform memory access (**UMA**) to non-uniform memory access (**NUMA**). UMA is defined as the situation in which access to any RAM from any CPU takes the same amount of time. With NUMA, some parts of memory may take longer to access than other parts,

creating a performance penalty.

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

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



Finally, **blade servers** are a relatively recent development in which multiple processor boards, I/O boards, and networking boards are placed in the same chassis. The difference between these and traditional multiprocessor systems is that each blade-processor board boots independently and runs its own operating system. Some blade-server boards are multiprocessor as well, which blurs the lines between types of computers. In essence, these servers consist of multiple independent multiprocessor systems.

1.3.3 Clustered Systems

Another type of multiprocessor system is a **clustered system**, which gathers together multiple CPUs. Clustered systems differ from the multiprocessor systems in that they are composed of two or more individual systems—or nodes—joined together. Such systems are considered **loosely coupled**. Each node may be a single processor system or a multicore system. We should note that the definition of ***clustered*** is not concrete; many commercial packages wrestle to define a clustered system and why one form is better than another. The generally accepted definition is that clustered computers share storage and are closely linked via a local-area network LAN or a faster interconnect, such as InfiniBand.

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

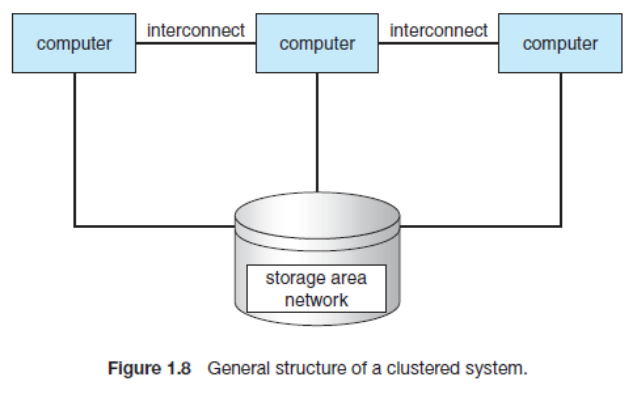
Since a cluster consists of several computer systems connected via a network, clusters can also be used to provide **high-performance computing** environments. The application must

have been written specifically to take advantage of the cluster, however. This involves a technique known as **parallelization**, which divides a program into separate components that run in parallel on individual computers in the cluster.

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 shared storage. To provide this shared access, 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)**,

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

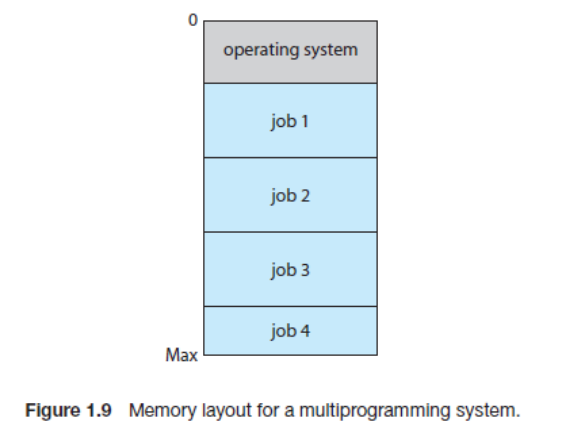
increasing performance and reliability. Figure 1.8 depicts the general structure of a clustered system.



Operating-System Structure

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

The idea is as follows: The operating system keeps several jobs in memory simultaneously (Figure 1.9). Since, in general, main memory is too small to accommodate all jobs, the jobs are kept initially on the disk in the **job pool**. This pool consists of all processes residing on disk awaiting allocation of main Memory.



The set of jobs in memory can be a subset of the jobs kept in the job pool. The operating system picks and begins to execute one of the jobs in memory. Eventually, the job may have to wait for some task, such as an I/O operation, to complete. In a non-multiprogrammed system, the CPU would sit idle. In a multiprogrammed system, the operating system simply switches to, and executes, another job. When ***that*** job needs to wait, the CPU switches to ***another*** job, and so on. Eventually, the first job finishes waiting and gets the CPU back. As long as at least one job needs to execute, the CPU is never idle.

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

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

be short—typically less than one second.

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

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

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

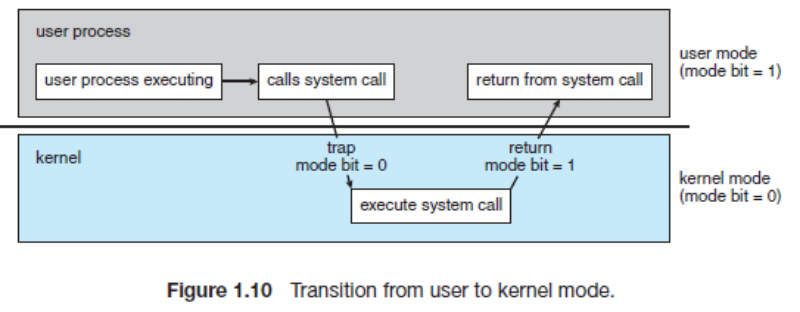
Operating-System Operations

Modern operating systems are **interrupt driven**. If there are no processes to execute, no I/O devices to service, and no users to whom to respond, an operating system will sit quietly, waiting for something to happen. Events are almost always signaled by the occurrence of an interrupt or a trap. A **trap** (or an **exception**) is a software-generated interrupt caused either by an error (for example, division by zero or invalid memory access) or by a specific request from a user program that an operating-system service be performed. The interrupt-driven nature of an operating system defines that system’s general structure. More subtle errors can occur in a multiprogramming system, where one erroneous program might modify another program, the data of another program, or even the operating system itself.

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

Dual-Mode and Multimode Operation

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



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

executing on behalf of a user application, the system is in user mode. However, when a user application requests a service from the operating system (via a system call), the system must transition from user to kernel mode to fulfill the request. This is shown in Figure 1.10. As we shall see, this architectural enhancement is useful for many other aspects of system operation as well.

At system boot time, the hardware starts in kernel mode. The operating system is then loaded and starts user applications in user mode. Whenever a trap or interrupt occurs, the hardware switches from user mode to kernel mode (that is, changes the state of the mode bit to 0). Thus, whenever the operating system gains control of the computer, it is in kernel mode. The system always switches to user mode (by setting the mode bit to 1) before passing control to

a user program. The dual mode of operation provides us with the means for protecting the

operating system from errant users—and errant users from one another.

The concept of modes can be extended beyond two modes (in which case the CPU uses more than one bit to set and test the mode). CPUs that support virtualization (Section 16.1) frequently have a separate mode to indicate when the **virtual machine manager (VMM)**—and the virtualization management software—is in control of the system. In this mode, the VMM has more privileges than user processes but fewer than the kernel. It needs that level

of privilege so it can create and manage virtual machines, changing the CPU state to do so. Sometimes, too, different modes are used by various kernel components. We should note that, as an alternative to modes, the CPU designer may use other methods to differentiate operational privileges. The Intel 64 family of CPUs supports four ***privilege levels,*** for example, and supports virtualization but does not have a separate mode for virtualization.

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. A user program running awry can wipe out the operating system by writing over it with data; and multiple programs are able to write to a device at the same time, with potentially disastrous results. Modern versions of the Intel CPU

do provide dual-mode operation.

Timer

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

1/60 second) or variable (for example, from 1 millisecond to 1 second). A **variable timer** is generally implemented by a fixed-rate clock and a counter.

The operating system sets the counter. Every time the clock ticks, the counter is decremented. When the counter reaches 0, an interrupt occurs. For instance, a 10-bit counter with a 1-millisecond clock allows interrupts at intervals from 1 millisecond to 1,024 milliseconds, in steps of 1 millisecond.

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

Computing Environments

operating systems are used in a variety of computing environments

Traditional Computing

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

Today, traditional time-sharing systems are uncommon. The same scheduling technique is still in use on desktop computers, laptops, servers, and even mobile computers, but frequently all the processes are owned by the same user (or a single user and the operating system). User processes, and system processes that provide services to the user, are managed so that each frequently gets a slice of computer time.

Mobile Computing

**Mobile computing** refers to computing on handheld smart phones and tablet computers. These devices share the distinguishing physical features of being portable and lightweight. Today, mobile systems are used not only for e-mail and web browsing but also for playing music and video, reading digital books, taking photos, and recording high-definition video. Accordingly, tremendous growth continues in the wide range of applications that run on such devices. Many developers are now designing applications that take advantage of the unique features of mobile devices, such as global positioning system (GPS) chips, accelerometers,

and gyroscopes. An embedded GPS chip allows a mobile device to use satellites to determine its precise location on earth.

Two operating systems currently dominate mobile computing: **Apple iOS** and **Google Android**. iOS was designed to run on Apple iPhone and iPad mobile devices. Android powers smartphones and tablet computers available from many manufacturers

Distributed Systems

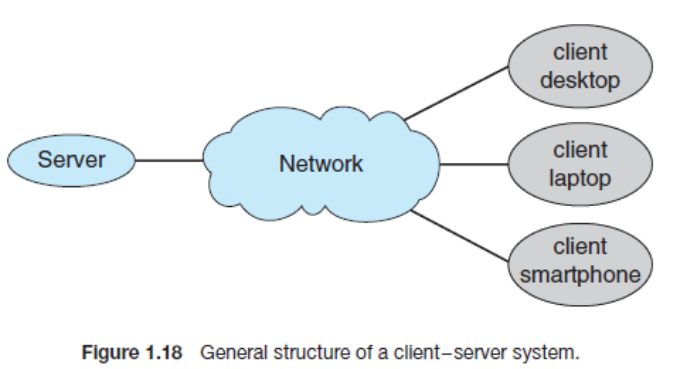
A distributed system is a collection of physically separate, possibly heterogeneous, computer systems that are networked to provide users with access to the various resources that the system maintains. Access to a shared resource increases computation speed, functionality, data availability, and reliability.

Some operating systems generalize network access as a form of file access, with the details of networking contained in the network interface’s device driver. Others make users specifically invoke network functions. Some operating systems have taken the concept of networks and distributed systems further than the notion of providing network connectivity

A **network operating system** is an operating system that provides features such as file sharing across the network, along with 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. A distributed operating system provides a less autonomous environment. The different computers communicate closely enough to provide the illusion that only a single operating system controls the network.

Client–Server Computing

Terminals connected to centralized systems are now being supplanted by PCs and mobile devices. Correspondingly, user-interface functionality once handled directly by centralized systems is increasingly being handled by PCs, quite often through a web interface. As a result, many of today’s systems act as **server systems** to satisfy requests generated by **client systems**. This form of specialized distributed system, called a **client–server** system, has the general structure depicted in Figure 1.18.



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

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

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

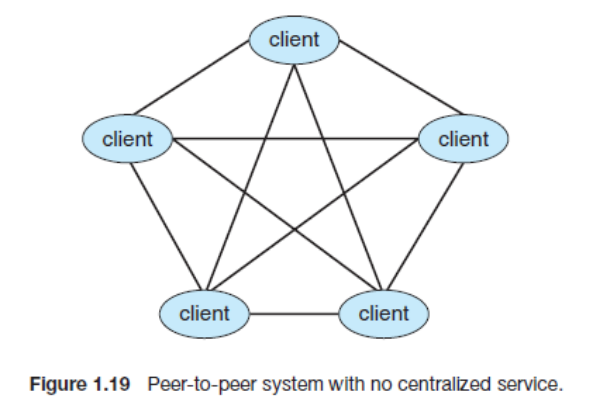
Peer-to-Peer Computing

Another structure for a distributed system is the peer-to-peer (P2P) system model. In this model, clients and servers are not distinguished from one another. Instead, all nodes within the system are considered peers, and each may act as either a client or a server, depending on whether it is requesting or providing a service. Peer-to-peer systems offer an advantage over traditional client-server systems. In a client-server system, the server is a bottleneck; but in a peer-to-peer system, services can be provided by several nodes distributed throughout the network.

Determining what services are available is accomplished in one of two general ways:

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

• An alternative scheme uses no centralized lookup service. Instead, a peer acting as a client must 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. Figure 1.19 illustrates such a scenario.



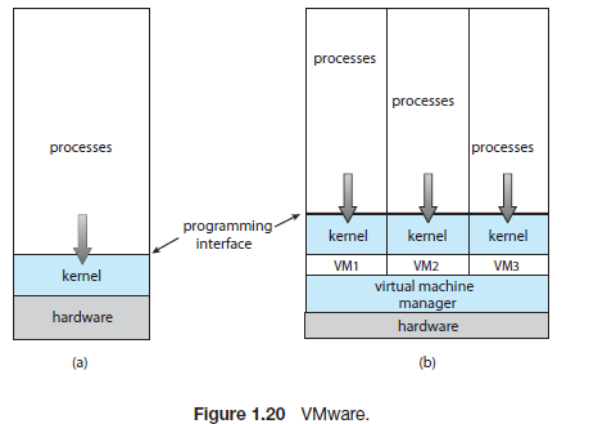
Peer-to-peer networks gained widespread popularity in the late 1990s with several file-sharing services, such as Napster and Gnutella, that enabled peers to exchange files with one another. The Napster system used an approach similar to the first type described above: a centralized server maintained an index of all files stored on peer nodes in the Napster network, and the actual exchange of files took place between the peer nodes. The Gnutella system used a technique similar to the second type: a client broadcasted file requests to other nodes in the system, and nodes that could service the request responded directly to the client.

Skype is another example of peer-to-peer computing. It allows clients to make voice calls and video calls and to send text messages over the Internet using a technology known as **voice over IP (VoIP)**. Skype uses a hybrid peer-to- peer approach.

Virtualization

Virtualization is a technology that allows operating systems to run as applications within other operating systems. At first blush, there seems to be little reason for such functionality. But the virtualization industry is vast and growing, which is a testament to its utility and importance. Broadly speaking, virtualization is one member of a class of software that also includes emulation. **Emulation** is used when the source CPU type is different from the target CPU type.

With **virtualization**, in contrast, an operating system that is natively compiled for a particular CPU architecture runs within another operating system also native to that CPU. Microsoft Windows XP applications on the Intel x86 CPU, VMware created a new virtualization technology in the form of an application that ran on XP. That application ran one or more **guest** copies of Windows or other native x86 operating systems, each running its own applications. (See Figure 1.20.)



Windows was the **host** operating system, and the VMware application was the virtual machine manager VMM. The VMM runs the guest operating systems, manages their resource use, and protects each guest from the others

Cloud Computing

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

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

• **Public cloud**—a cloud available via the Internet to anyone willing to pay for the service

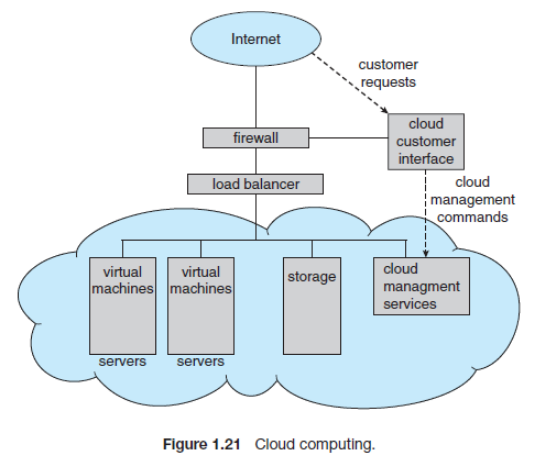
• **Private cloud**—a cloud run by a company for that company’s own use

• **Hybrid cloud**—a cloud that includes both public and private cloud components

• Software as a service **(SaaS)**—one or more applications (such as word processors or spreadsheets) available via the Internet

• Platform as a service **(PaaS)**—a software stack ready for application use via the Internet (for example, a database server)

• Infrastructure as a service **(IaaS)**—servers or storage available over the Internet (for example, storage available for making backup copies of production data) Figure 1.21 illustrates a public cloud providing IaaS.

:

Real-Time Embedded Systems

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

automobile engines and robotic arms.

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

A real-time system has well-defined, fixed time constraints. Processing ***must*** be done within the defined constraints, or the system will fail. For instance, it would not do for a robot arm to be instructed to halt ***after*** it 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

Open-Source Operating Systems

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

There are many benefits to open-source operating systems, including a community of interested (and usually unpaid) programmers who contribute to the code by helping to debug it, analyze it, provide support, and suggest changes. Arguably, open-source code is more secure than closed-source code because many more eyes are viewing the code. Companies that earn revenue from selling their programs often hesitate to open-source their code, but Red Hat and a myriad of other companies are doing just that and showing that commercial companies benefit, rather than suffer, when they open-source their code. Revenue can be generated through support contracts and the sale of hardware on which the software runs, for example.

History

In the early days of modern computing (that is, the 1950s), a great deal of software was available in open-source format.

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

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

To counter the move to limit software use and redistribution, Richard Stallman in 1983 started the GNU project to create a free, open-source, UNIX compatible operating system. In 1985, he published the GNU Manifesto, which argues that all software should be free and open-sourced. He also formed the **Free Software Foundation (FSF)** with the goal of encouraging the free exchange of software source code and the free use of that software. Rather than copyright its software, the FSF “copy lefts” the software to encourage sharing and improvement. The **GNU General Public License (GPL)** codifies copy lefting and is a common license under which free software is released.

Linux

As an example of an open-source operating system, consider **GNU/Linux**. The GNU project produced many UNIX-compatible tools, including compilers, editors, and utilities, but never released a kernel. In 1991, a student in Finland, Linus Torvalds, released a rudimentary UNIX-like kernel using the GNU compilers and tools and invited contributions worldwide.

The resulting GNU/Linux operating system has spawned hundreds of unique **distributions**, or custom builds, of the system.

Major distributions include RedHat, SUSE, Fedora, Debian, Slackware, and Ubuntu. Distributions vary in function, utility, installed applications, hardware support, user interface,

and purpose. For example, RedHat Enterprise Linux is geared to large commercial use. PCLinuxOS is a **LiveCD**—an operating system that can be booted and run from a CD-ROM without being installed on a system’s hard disk. One variant of PCLinuxOS—called “PCLinuxOS Supergamer DVD”—is a **LiveDVD** that includes graphics drivers and games. A gamer can run it on any compatible system simply by booting from the DVD. When the gamer is finished, a reboot of the system resets it to its installed operating system.

BSD UNIX

**BSD UNIX** has a longer and more complicated history than Linux. It started in 1978 as a derivative of AT&T’s UNIX. BSD UNIX’s development was slowed by a lawsuit by AT&T, but eventually a fully functional, open-source version, 4.4BSD-lite, was released in 1994.

Just as with Linux, there are many distributions of BSD UNIX, including FreeBSD, NetBSD, OpenBSD, and DragonflyBSD. To explore the source code of FreeBSD, simply download the virtual machine image of the version of interest and boot it within VMware, as described above for Linux.

Solaris

**Solaris** is the commercial UNIX-based operating system of Sun Microsystems. In 2005, Sun open-sourced most of the Solaris code as the OpenSolaris project. Several groups interested in using OpenSolaris have started from that base and expanded its features. Their working set is Project Illumos, which has expanded from the OpenSolaris base to include more features and to be the basis for several products.

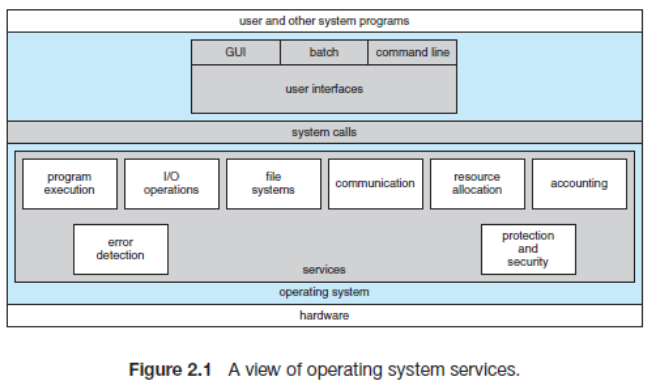
Open-Source Systems as Learning Tools

The free software movement is driving legions of programmers to create thousands of open-source projects, including operating systems. Sites like http://freshmeat.net/ and http://distrowatch.com/ provide portals to many of these projects. As we stated earlier, open-source projects enable students to use source code as a learning tool.

GNU/Linux and BSD UNIX are all open-source operating systems, but each has its own goals, utility, licensing, and purpose. several major components of OpenSolaris have been ported to BSD UNIX. The advantages of free software and open sourcing are likely to increase the number and quality of open-source projects, leading to an increase in the number of individuals and companies that use these projects.

Operating-System Services

An operating system provides an environment for the execution of programs. It provides certain services to programs and to the users of those programs. The specific services provided, of course, differ from one operating system to another. Figure 2.1 shows one view of the various operating-system services and how they interrelate.



One set of operating system services provides functions that are helpful to the user.

**User interface**. Almost all operating systems have a **user interface (UI)**. This interface can take several forms. One is a **command-line interface (CLI)**, which uses text commands and a method for entering them (say, a keyboard for typing in commands in a specific format with specific options). Another is a **batch interface**, in which commands and directives to control those commands are entered into files, and those files are executed. Most commonly, a **graphical user interface (GUI)** is used. Here, the interface is a window system with a pointing device to direct I/O, choose from menus, and make selections and a keyboard to enter text. Some systems provide two or all three of these variations.

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

• **I/O operations**. A running program may require I/O, which may involve a file or an I/O device. 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 operating systems include permissions management to allow or deny access to files or directories based on file ownership.

.• **Communications**. Communication may occur between processes that are executing on the same computer or between processes that are executing on different computer systems tied

together by a computer network.

• **Error detection**. The operating system needs to be detecting and correcting errors constantly.

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

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

• **Accounting**. Keep track of which users use how much and what kinds of computer resources. This record keeping may be used for accounting or simply for accumulating usage

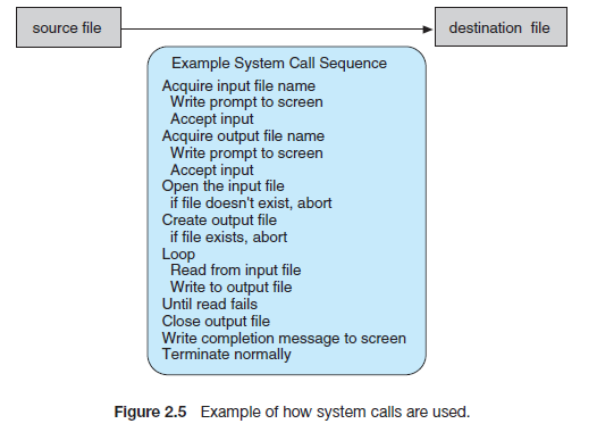
statistics.

• **Protection and security**. Protection involves ensuring that all access to system resources is controlled. Security of the system from outsiders is also important. Such security starts with requiring each user to authenticate himself or herself to the system, usually by means of a password, to gain access to system resources. If a system is to be protected and secure, precautions must be instituted throughout it. A chain is only as strong as its weakest link.

System Calls

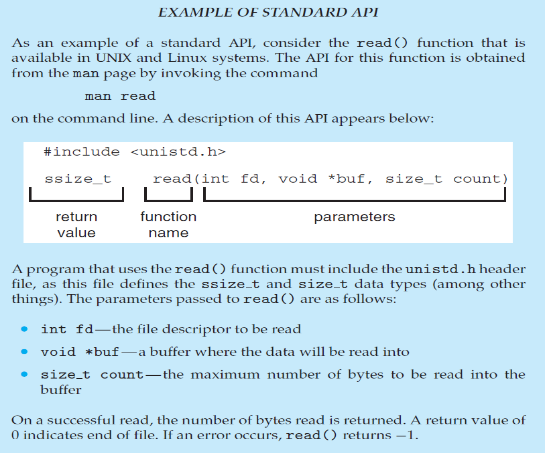
**System calls** provide an interface to the services made available by an operating system. These calls are generally available as routines written in C and C++, although certain low-level tasks may have to be written using assembly-language instructions.

An example to illustrate how system calls are used: writing a simple program to read data from one file and copy them to another file.



Typically, application developers design programs according to an **application programming interface (API)**. The API specifies a set of functions that are available to an application programmer, including the parameters that are passed to each function and the return values the programmer can expect. Three of the most common APIs available to application programmers are the Windows API for Windows systems, the POSIX API for POSIX-based systems (which include virtually all versions of UNIX, Linux, and Mac OSX), and the Java API for programs that run on the Java virtual machine. A programmer accesses

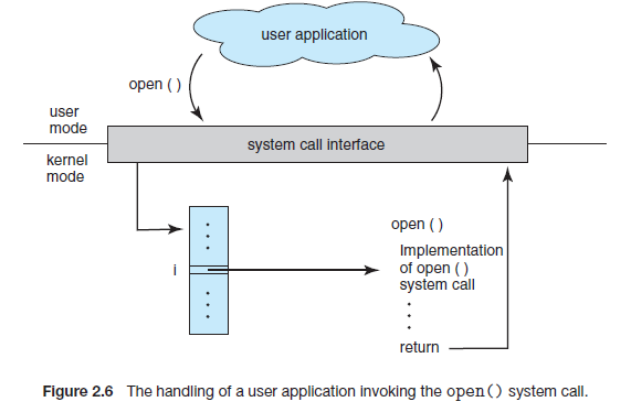
an API via a library of code provided by the operating system. In the case of UNIX and Linux for programs written in the C language, the library is called **libc**.



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.

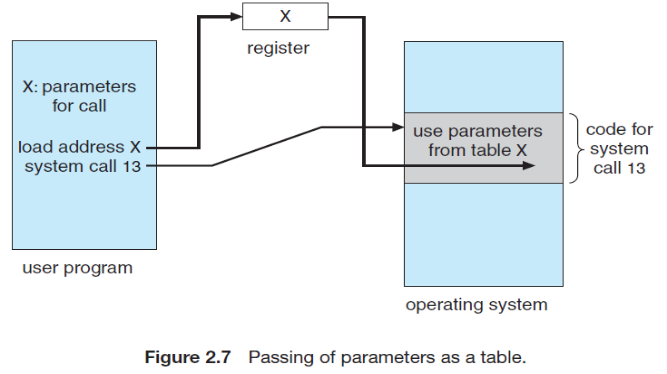
For most programming languages, the run-time support provides a **system call interface** that serves as the link to system calls made available by the operating system. The system-call interface intercepts function calls in the API and invokes the necessary system calls within the operating system. Typically, a number is associated with each system call, and the system-call interface maintains a table indexed according to these numbers. The system call interface then invokes the intended system call in the operating-system kernel and returns the status of the system call and any return values.

The relationship between an API, the system-call interface, and the operating system is shown in Figure 2.6, which illustrates how the operating system handles a user application invoking the open() system call.



System calls occur in different ways, depending on the computer in use. Often, more information is required than simply the identity of the desired system call. The exact type and amount of information vary according to the particular operating system and call.

Three general methods are used to pass parameters to the operating system. The simplest approach is to pass the parameters in registers. In some cases, however, there may be more parameters than registers. In these cases, the parameters are generally stored in a block, or table, in memory, and the address of the block is passed as a parameter in a register (Figure 2.7). This is the approach taken by Linux and Solaris. Parameters also can be placed, or **pushed**, onto the **stack** by the program and **popped** off the stack by the operating system. Some operating systems prefer the block or stack method because those approaches do not limit the number or length of parameters being passed.



Types of system calls

System calls can be grouped roughly into six major categories: **process control**, **file manipulation**, **device manipulation**, **information maintenance**, **communications**, and **protection**.

Process Control

A running program needs to be able to halt its execution either normally (end) or abnormally (abort). If a system call is made to terminate the currently running program abnormally, or if the program runs into a problem and causes an error trap, a dump of memory is sometimes taken and an error message generated. The dump is written to disk and may be examined by a **debugger**—a system program designed to aid the programmer in finding and correcting bugs-—to determine the cause of the problem. Under either normal or abnormal circumstances, the operating system must transfer control to the invoking command interpreter. The command interpreter then reads the next command. In an interactive system, the command interpreter simply continues with the next command; it is assumed that the user will issue an appropriate command to respond to any error.

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

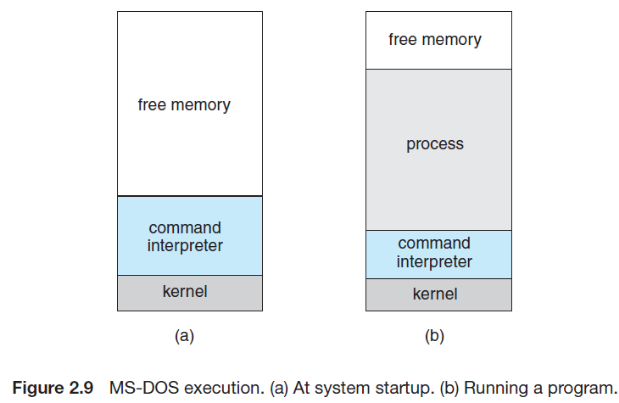
If we create a new job or process, or perhaps even a set of jobs or processes, we should be able to control its execution. This control requires the ability to determine and reset the attributes of a job or process, including the job’s priority, its maximum allowable execution time, and so on (get process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

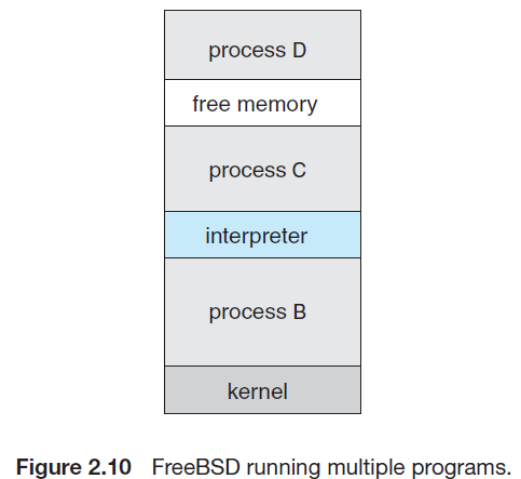
The MS-DOS operating system is an example of a single-tasking system. It has a command interpreter that is invoked when the computer is started (Figure 2.9(a)). Because MS-DOS

is single-tasking, it uses a simple method to run a program and does not create a new process. It loads the program into memory, writing over most of itself to give the program as much memory as possible (Figure 2.9(b)). Next, it sets the instruction pointer to the first instruction of the program. The program then runs, and either an error causes a trap, or the program executes a system call to terminate.

FreeBSD (derived from Berkeley UNIX) is an example of a multitasking system. When a user logs on to the system, the shell of the user’s choice is run. This shell is similar to the MS-DOS shell in that it accepts commands and executes programs that the user requests. However, since FreeBSD is a multitasking system, the command interpreter may continue running while another program is executed (Figure 2.10).

To start a new process, the shell executes a fork() system call. Then, the selected program is loaded into memory via an exec() system call, and the program is executed. Depending on the way the command was issued, the shell then either waits for the process to finish or runs the process “in the background.”





**File Management**

We first need to be able to create and delete files. Either system call requires the name of the file

and perhaps some of the file's attributes. Once the file is created, we need to open it and to use it. We

may also read, write, or reposition (rewinding or skipping to the end of the file, for example). Finally, we need to close the file, indicating that we are no longer using it. We may need these same sets of operations for directories if we have a directory structure for organizing files in the file system. In addition, for either files or directories, we need to be able to determine the values of various attributes and perhaps to reset them if necessary. File attributes include the file name, a file type, protection codes, accounting information, and so on. At least two system calls, get file attribute and set file attribute, are required for this function. Some operating systems provide many more calls, such as calls for file move and copy.

**Device Management**

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

Otherwise, the process will have to wait until sufficient resources are available. The various resources controlled by the operating system can be thought of as devices. Some of these devices are physical devices (for example, tapes), while others can be thought of as abstract or virtual devices (for example, files). If there are multiple users of the system, the system may require us to first request the device, to ensure exclusive use of it. After we are finished with the device, we release it. These functions are similar to the open and close system calls for files.

**Information Maintenance**

Many system calls exist simply for the purpose of transferring information between the user

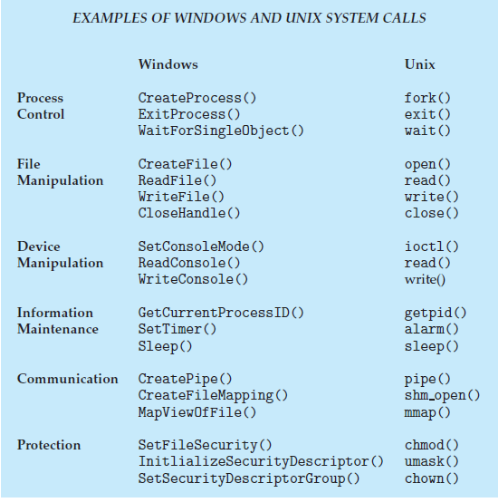
program and the operating system. For example, most systems have a system call to return the current time and date . Other system calls may return information about the system, such as the number of current users, the version number of the operating system, the amount of free memory or disk space, and so on.

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

**Communication**

There are two common models of inter process communication: the message passing model and the shared-memory model. In the message-passing model, the communicating processes exchange messages with one another to transfer information. Messages can be exchanged between the processes either directly or indirectly through a common mailbox. Before communication can take place, a connection must be opened. The name of the other communicator must be known, be it another process on the same system or a process on another computer connected by a communications network. each process has a *process name,* and this name is translated into an identifier by which the operating system can refer to the process. The get host id and get processed system calls do this translation. The identifiers are then passed to the general purpose open and close calls provided by the file system or to specific open connection and close connection system calls, depending on the system's model of communication. In the shared-memory model, processes use shared memory creates and shared memory attaches system calls to create and gain access to regions of memory owned by other processes.

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



System Programs

Modern system is a collection of system programs. At the lowest level is hardware. Next are the operating system, then the system programs, and finally the application programs. System programs provide a convenient environment for program development and execution. Some of them are simply user interfaces to system calls; others are considerably more complex.

They can be divided into these categories:

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

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

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

• **Programming-language support.** Compilers, assemblers, debuggers and interpreters for common programming languages (such as C, C++, Java, 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.

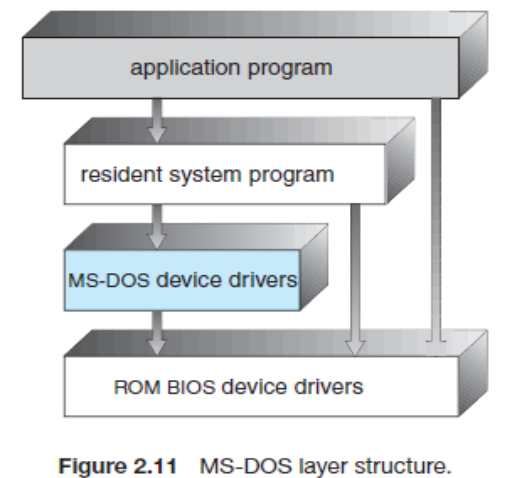
• **Communications.** These programs provide the mechanism for creating virtual connections among processes, users, and computer systems. They allow users to send messages to one another's screens, to browse web pages, to send electronic-mail messages, to log in remotely, or to transfer files from one machine to another. In addition to systems programs, most 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 structure

A system as large and complex as a modern operating system must be engineered carefully if it is to function properly and be modified easily

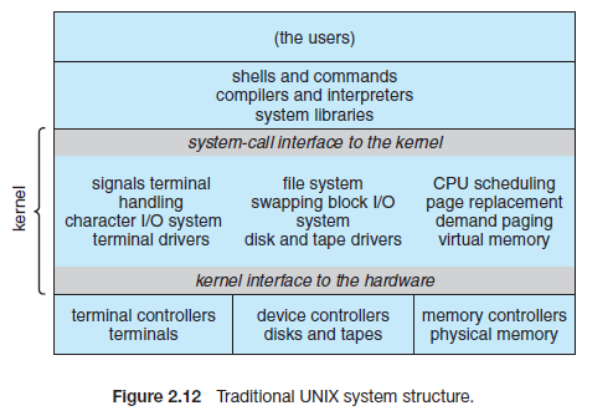
**Simple Structure**

Many commercial systems do not have well-defined structures. Frequently, 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. For instance, application programs are able to access the basic I/O routines to write directly to the display and disk drives. Such freedom leaves MS-DOS vulnerable to errant (or malicious) programs, causing entire system crashes when user programs fail. Of course, MS-DOS was also limited by the hardware of its era. Another example of limited structuring is the original UNIX operating system. UNIX is another system that initially was limited by hardware functionality.

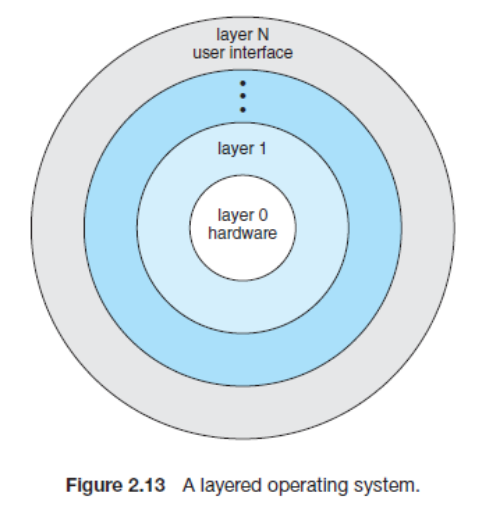
It consists of two separable parts: the kernel and the system programs. The kernel is further separated into a series of interfaces and device drivers, which have been added and expanded over the years as UNIX has evolved

* ****

**Layered Approach**

The operating system can then retain much greater control over the computer and over the applications that make use of that computer. Implementers have more freedom in changing the inner workings of the system and in creating modular operating systems. Under the top down approach, the overall functionality and features are determined and are separated into components. Information hiding is also important, because it leaves programmers free to implement the low-level routines as they see fit, provided that the external interface of the routine stays unchanged and that the routine itself performs the advertised task.

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

****

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

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

layers are selected so that each uses functions (operations) and services of only lower-level layers. This approach simplifies debugging and system verification. The first layer can be debugged without any concern for the rest of the system, because, by definition, it uses only the basic hardware (which is assumed correct) to implement its functions. Once the first layer is debugged, its correct functioning can be assumed while the second layer is debugged, and so on. If an error is found during the debugging of a particular layer, the error must be on that layer, because the layers below it are already debugged. Thus, the design and implementation of the system is simplified.

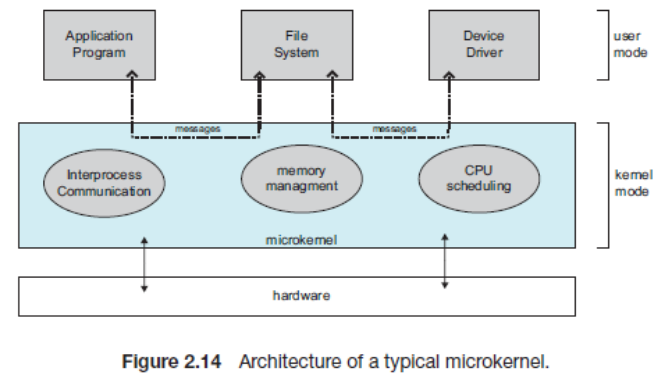
Each layer is implemented with only those operations provided by lower level layers. A layer

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

The major difficulty with the layered approach involves appropriately defining the various layers. The backing-store driver would normally be above the CPU scheduler, because the driver may need to wait for I/O and the CPU can be rescheduled during this time. A final problem with layered implementations is that they tend to be less efficient than other types. For instance, when a user program executes an I/O operation, it executes a system call that is trapped to the I/O layer, which calls the memory-management layer, which in turn calls the CPU-scheduling layer, which is then passed to the hardware.

**Micro kernels**

The kernel became large and difficult to manage. In the mid-1980s, researchers at Carnegie Mellon University developed an operating system called **Mach** that modularized the kernel using the **microkernel** approach. This method structures the operating system by removing all nonessential components from the kernel and implementing them as system and user-level programs. The result is a smaller kernel. microkernels provide minimal process and memory management, in addition to a communication facility. Figure 2.14 illustrates the architecture of a typical microkernel.



The main function of the microkernel is to provide a communication facility between the client program and the various services that are also running in user space. One benefit of the microkernel approach is 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 does have to be modified, the changes tend to be fewer, because the microkernel is a smaller kernel.

The resulting operating system is easier to port from one hardware design to another. The

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

**Modules**

Perhaps the best current methodology for operating-system design involves using **loadable kernel modules**. Here, the kernel has a set of core components and links in additional services via modules, either at boot time or during run time. This type of design is common in modern implementations of UNIX, such as Solaris, Linux, and Mac OS X, as well as Windows.

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

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

**1.** Scheduling classes

**2.** File systems

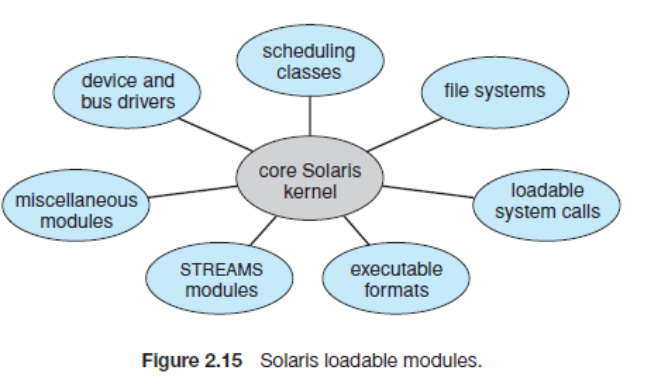
**3.** Loadable system calls

**4.** Executable formats

**5.** STREAMS modules

**6.** Miscellaneous

**7.** Device and bus drivers

****

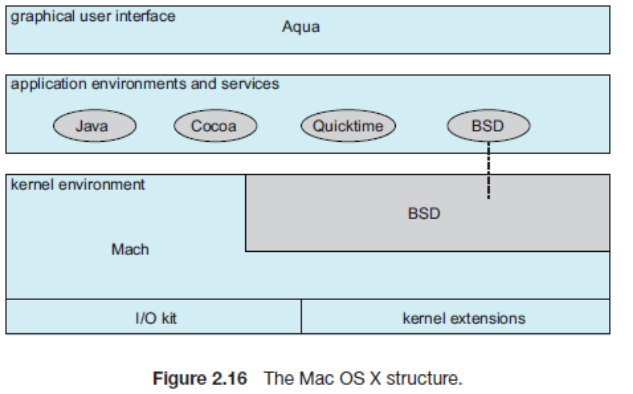
Hybrid Systems

In practice, very few operating systems adopt a single, strictly defined structure. Instead, they combine different structures, resulting in hybrid systems that address performance, security, and usability issues. three hybrid systems: the Apple Mac OS X operating system and the two most prominent mobile operating systems—iOS and Android.

**Mac OS X**

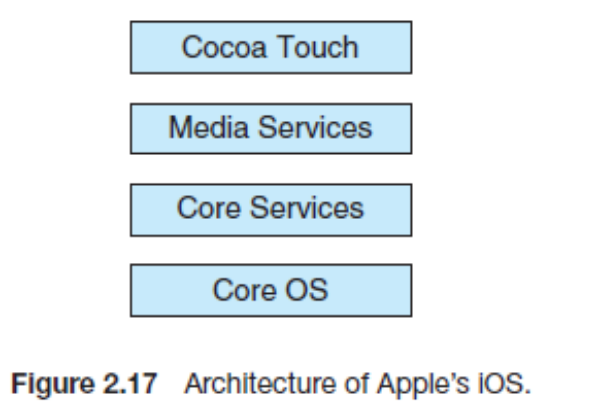
The Apple Mac OS X operating system uses a hybrid structure. As shown in Figure 2.16, it is a layered system. The Apple Mac OS X operating system uses a hybrid structure. As shown in Figure 2.16, it is a layered system. the **Cocoa** environment specifies an API for the Objective-C programming language, which is used for writing Mac OS X applications. Below these layers is the ***kernel environment***, which consists primarily of the Mach microkernel and the BSD UNIX kernel. Mach provides memory management; support for remote procedure calls (RPCs) and interprocess communication (IPC) facilities, including message passing; and thread scheduling. The BSD component provides a BSD command-line interface, support for networking and file systems, and an implementation of POSIX APIs, including Pthreads. In addition to Mach and BSD, the kernel environment provides an I/O kit

for development of device drivers and dynamically loadable modules

****

**iOS**

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

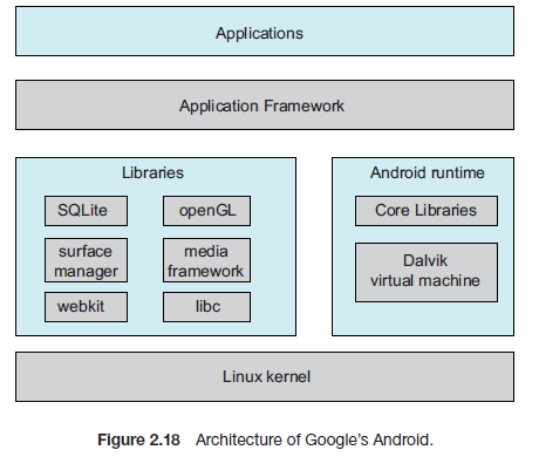


**Cocoa Touch** is an API for Objective-C that provides several frameworks for developing applications that run on iOS devices. The **core services** layer provides a variety of features, including support for cloud computing and databases

**Android**

The Android operating system was designed by the Open Handset Alliance (led primarily by Google) and was developed for Android smartphones and tablet computers. The structure of

Android appears in Figure 2.18.



Linux is used primarily for process, memory, and device-driver support for hardware and has been expanded to include power management. The Android runtime environment includes a core set of libraries as well as the Dalvik virtual machine. The set of libraries available for Android applications includes frameworks for developing web browsers (webkit), database support (SQLite), and multimedia. The libc library is similar to the standard C library but is much smaller and has been designed for the slower CPUs that characterize mobile devices