Introduction:

An *operating system* is software that manages a computer's hardware. It also provides a basis for application programs and acts as an intermediary between the computer user and the computer hardware. An amazing aspect of operating systems is how they vary in accomplishing these tasks in a wide variety of computing environments. Operating systems are everywhere, from cars and home appliances that include "Internet of Things" devices, to smart phones, personal computers, enterprise computers, and cloud computing environments.

What Operating Systems Do?

A computer system can be divided roughly into four components: the hardware, the operating system, the application programs, and a user (Figure 1.1). The hardware— the central processing unit (CPU), the memory, and the input/output (I/O) devices—provides the basic computing resources for the system. The application programs—such as word processors, spreadsheets, compilers, and web browsers—define the ways in which these resources are used to solve users' computing problems. The operating system controls the hardware and coordinates its use among the various application programs for the various users.

A computer system can be divided roughly into four components: the hardware, the operating system, the application programs, and a user (Figure 1.1). The hardware— the central processing unit (CPU), the memory, and the input/output (I/O) devices—provides the basic computing resources for the system. The application programs—such as word processors, spreadsheets, compilers, and web browsers—define the ways in which these resources are used to solve users' computing problems. The operating system controls the hardware and coordinates its use among the various application programs for the various users.

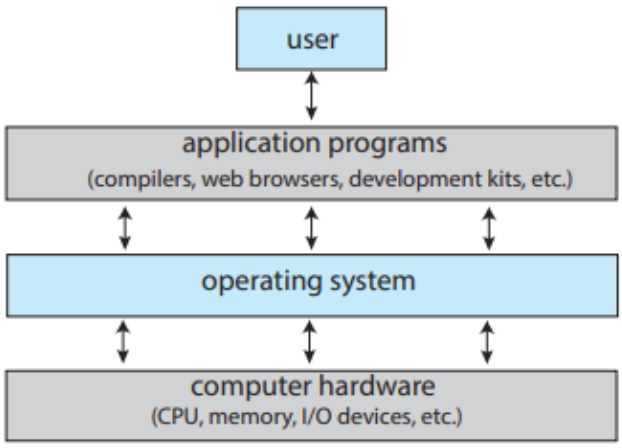


Figure 1.1 Abstract view of the components of a computer system.

User View

The user's view of the computer varies according to the interface being used. Many computer users sit with a laptop or in front of a PC consisting of a monitor, keyboard, and mouse. 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 security and none paid to resource utilization—how various hardware and software resources are shared.

The user interface for mobile computers generally features a touch screen, where the user interacts with the system by pressing and swiping fingers across the screen rather than using a physical keyboard and mouse. Many mobile devices also allow users to interact through a voice recognition interface, such as Apple's Siri. 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 and applications are designed primarily to run without user intervention.

System View

A computer system has many resources that may be required to solve a problem: CPU time, memory space, storage space, I/O devices, and so on. The operating system acts as the manager of these resources. Facing numerous and possibly conflicting requests for resources, the operating system must decide how to allocate them to specific programs and users so that it can operate the computer system efficiently and fairly. A slightly different view of an operating system emphasizes the need to control the various I/O devices and user programs. An operating system is a control program. A control program manages the execution of user programs to prevent errors and improper use of the computer. It is especially concerned with the operation and control of I/O devices.

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.

Single-Processor Systems

Many years ago, most computer systems used a single processor containing one CPU with a single processing core. The core is the component that executes instructions and registers for storing data locally. The one main CPU with its core is capable of executing a general-purpose instruction set, including instructions from processes. All of these special-purpose processors run a limited instruction set and do not run processes. Sometimes, they are managed by the operating system, in that the operating system sends them information about their next task and monitors their status. For example, a disk-controller microprocessor receives a sequence of requests from the main CPU core

and implements its own disk queue and scheduling algorithm. This arrangement relieves the main CPU of the overhead of disk scheduling. PCs contain a microprocessor in the keyboard to convert the keystrokes into codes to be sent to the CPU. In other systems or circumstances, special-purpose processors are low-level components built into the hardware. The operating system cannot communicate with these processors; they do their jobs autonomously. The use of special-purpose microprocessors is common and does not turn a single-processor system into a multiprocessor. If there is only one general-purpose CPU with a single processing core, then the system is a single-processor system. According to this definition, however, very few contemporary computer systems are single-processor systems.

Multiprocessor Systems:

On modern computers, from mobile devices to servers, multiprocessor systems now dominate the landscape of computing. Traditionally, such systems have two (or more) processors, each with a single-core CPU. The processors share the computer bus and sometimes the clock, memory, and peripheral devices. The primary advantage of multiprocessor systems is increased throughput. 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.

The most common multiprocessor systems use symmetric multiprocessing (SMP), in which each peer CPU processor performs all tasks, including operating-system functions and user processes. Figure 1.8 illustrates a typical SMP architecture with two processors, each with its own CPU. Notice that each CPU processor has its own set of registers, as well as a private—or local— cache. However, all processors share physical memory over the system bus.

The benefit of this model is that many processes can run simultaneously —N processes can run if there are N CPUs—without causing performance to deteriorate significantly. However, 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. A multiprocessor system of this form will allow processes and resources—such as memory— to be shared dynamically among the various processors and can lower the workload variance among the processors.

The definition of multiprocessor has evolved over time and now includes multicore systems, in which multiple computing cores reside on a single chip. Multicore systems can be more efficient than multiple chips with single cores because on-chip communication is faster than between-chip communication.

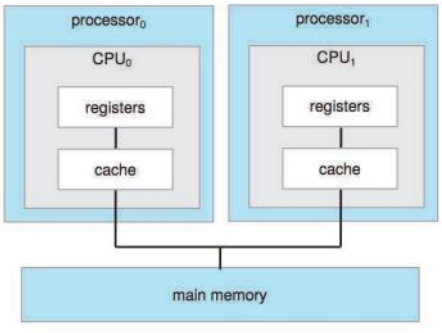


Figure 1.8 Symmetric multiprocessing architecture.

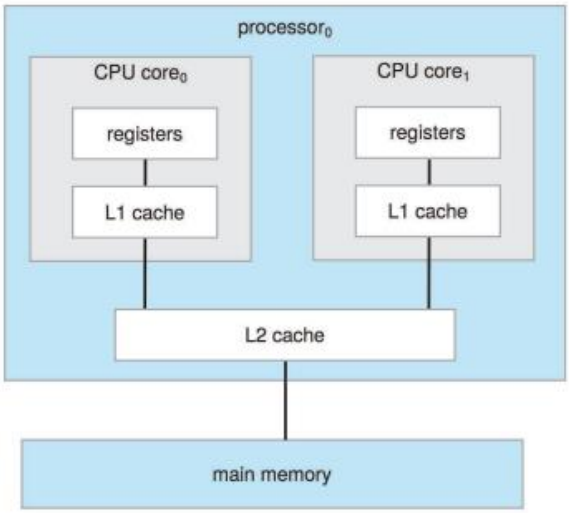


Figure 1.9 A dual-core design with two cores on the same chip.

DEFINITIONS OF COMPUTER SYSTEM COMPONENTS

- CPU—The hardware that executes instructions.
- Processor—A physical chip that contains one or more CPUs.
- Core—The basic computation unit of the CPU.
- Multicore—Including multiple computing cores on the same CPU.
- Multiprocessor—Including multiple processors.

Although virtually all systems are now multicore, we use the general term ***CPU*** when referring to a single computational unit of a computer system and ***core*** as well as ***multicore*** when specifically referring to one or more cores on a CPU.

NUMA systems:

Adding additional CPUs to a multiprocessor system will increase computing power; however, as suggested earlier, the concept does not scale very well, and once we add too many CPUs, contention for the system bus becomes a bottleneck and performance begins to degrade. An alternative approach is instead to provide each CPU (or group of CPUs) with its own local memory that is accessed via a small, fast local bus. The CPUs are connected by a shared system interconnect, so that all CPUs share one physical address space. This approach—known as non-uniform memory access, or NUMA—is illustrated in Figure 1.10.

The advantage is that, when a CPU accesses its local memory, not only is it fast, but there is also no contention over the system interconnect. Thus, NUMA systems can scale more effectively as more processors are added.

A potential drawback with a NUMA system is increased latency when a CPU must access remote memory across the system interconnect, creating a possible performance penalty. In other words, for example, CPU0 cannot access the local memory of CPU3 as quickly as it can access its own local memory, slowing down performance. Operating systems can minimize this NUMA penalty through careful CPU scheduling and memory management. Because NUMA systems can scale to accommodate a large number of processors, they are becoming increasingly popular on servers as well as high-performance computing systems.

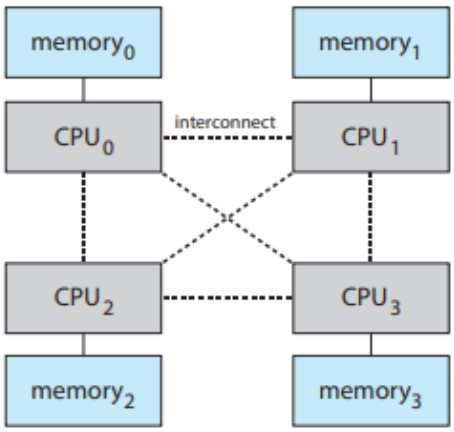


Figure 1.10 NUMA multiprocessing architecture.

Blade Servers:

Blade servers are systems 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 bladeprocessor 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.

Clustered Systems:

Another type of multiprocessor system is a clustered system, which gathers multiple CPUs. Clustered systems differ from the multiprocessor systems that they are composed of two or more individual systems—or nodes—joined together; each node is typically a multicore system. Such systems are considered loosely coupled.

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 is usually used to provide high-availability service— that is, service that will continue even if one or more systems in the cluster fail. Generally, we obtain high availability by adding a level of redundancy in the system.

The users and clients of the applications see only a brief interruption of service. High availability provides increased reliability, which is crucial in many applications. The ability to continue providing service proportional to the level of surviving hardware is called graceful degradation.

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 the available hardware. However, it does require that more than one application be available to run.

Since a cluster consists of several computer systems connected via a network, clusters can also be used to provide high-performance computing environments. Such systems can supply significantly greater computational power than single-processor or even SMP systems because they can run an application concurrently on all computers in the cluster.

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

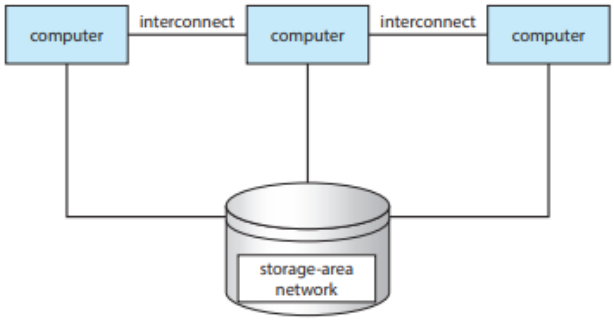


Figure 1.11 General structure of a clustered system.

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

Each machine has full access to all data in the database. 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 thousands 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.

Operating-System Operations:

For a computer to start running— for instance, when it is powered up or rebooted—it needs to have an initial program to run. As noted earlier, this initial program, or bootstrap program, tends to be simple. Typically, it is stored within the computer hardware in firmware. It initializes all aspects of the

system, from CPU registers to device controllers to memory contents. The bootstrap program must know how to load the operating system and how to start executing that system. To accomplish this goal, the bootstrap program must locate the operating-system kernel and load it into memory.

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

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. In Section 1.2.1 we described hardware interrupts. Another form of interrupt is a trap (or an exception), which 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 by executing a special operation called a system call.

Multiprogramming and Multitasking:

One of the most important aspects of operating systems is the ability to run multiple programs, as a single program cannot, in general, always keep either the CPU or the I/O devices busy. Furthermore, users typically want to run more than one program at a time as well. Multiprogramming increases CPU utilization, as well as keeping users satisfied, by organizing programs so that the CPU always has one to execute. In a multiprogrammed system, a program in execution is termed a process. The idea is as follows: The operating system keeps several processes in memory simultaneously (Figure 1.12). The operating system picks and begins to execute one of these processes. Eventually, the process may have to wait for some tasks, 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 process. When that process needs to wait, the CPU switches to another process, and so on. Eventually, the first process finishes waiting and gets the CPU back. As long as at least one process needs to execute, the CPU is never idle.

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

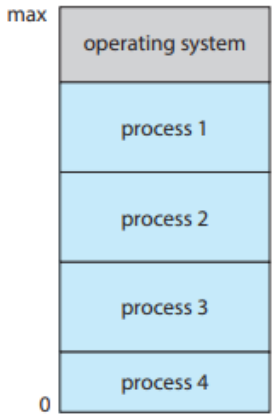


Figure 1.12 Memory layout for a multiprogramming system.

Multitasking is a logical extension of multiprogramming. In multitasking systems, the CPU executes multiple processes by switching among them, but the switches occur frequently, providing the user with a fast response time. Consider that when a process executes, it typically executes for only a short time before it either finishes or needs to perform I/O. I/O may be interactive; that is, output goes to a display for the user, and input comes from a user keyboard, mouse, or touch screen. Since interactive I/O typically runs at "people speeds," it may take a long time to complete. Input, for example, may be bounded by the user's typing speed; seven characters per second is fast for people but incredibly slow for computers. Rather than let the CPU sit idle as this interactive input takes place, the operating system will rapidly switch the CPU to another process.

Having several processes in memory at the same time requires some form of memory management, In addition, if several processes are ready to run at the same time, the system must choose which process will run next. Making this decision is CPU scheduling. Finally, running multiple processes concurrently requires that their ability to affect one another be limited in all phases of the operating system, including process scheduling, disk storage, and memory management.

In a multitasking system, the operating system must ensure reasonable response time. A common method for doing so is virtual memory, a technique that allows the execution of a process that is not completely in memory.

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

Dual-Mode and Multimode Operation:

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 fulfil the request. This is shown in Figure 1.13. As we shall see, this architectural enhancement is useful for many other aspects of system operation as well.

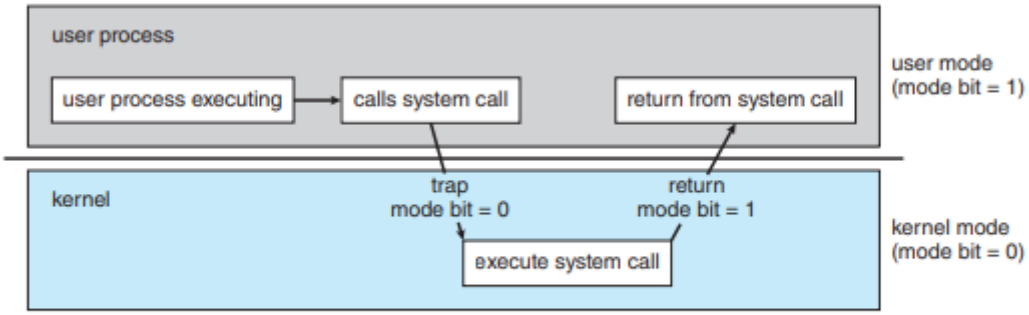


Figure 1.13 Transition from user to kernel mode.

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

The concept of modes can be extended beyond two modes. For example, Intel processors have four separate protection rings, where ring 0 is kernel mode and ring 3 is user mode. (Although rings 1 and 2 could be used for various operating-system services, in practice they are rarely used.) ARMv8 systems have seven modes. CPUs that support virtualization (Section 18.1) frequently have a separate mode to indicate when the virtual machine manager (VMM) 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.

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.

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 drive. Others make users specifically invoke network functions. Generally, systems contain a mix of the two modes— for example FTP and NFS. The protocols that create a distributed system can greatly affect that system's utility and popularity.

A network, in the simplest terms, is a communication path between two or more systems. Distributed systems depend on networking for their functionality. Networks vary by the protocols used, the distances between nodes, and the transport media. TCP/IP is the most common network protocol, and it provides the fundamental architecture of the Internet.

Networks are characterized based on the distances between their nodes. A local-area network (LAN) connects computers within a room, a building, or a campus. A wide-area network (WAN) usually links buildings, cities, or countries. A global company may have a WAN to connect its offices worldwide, for

example. These networks may run one protocol or several protocols. The continuing advent of new technologies brings about new forms of networks. For example, a metropolitan-area network (MAN) could link buildings within a city. Bluetooth and 802.11 devices use wireless technology to communicate over a distance of several feet, in essence creating a personal-area network (PAN) between a phone and a headset or a smartphone and a desktop computer.

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 can communicate with other networked computers. A distributed operating system provides a less autonomous environment.

Computing Environments:

1.Traditional Computing:

Today, web technologies and increasing WAN bandwidth are stretching the boundaries of traditional computing. Companies establish portals, which provide web accessibility to their internal servers. Network computers (or thin clients)—which are essentially terminals that understand web-based computing—are used in place of traditional workstations where more security or easier maintenance is desired. Mobile computers can synchronize with PCs to allow very portable use of company information. Mobile devices can also connect to wireless networks and cellular data networks to use the company's web portal (as well as the myriad other web resources).

At home, most users once had a single computer with a slow modem connection to the office, the Internet, or both. Today, network-connection speeds once available only at great cost are relatively inexpensive in many places, giving home users more access to more data. These fast data connections are allowing home computers to serve up web pages and to run networks that include printers, client PCs, and servers. Many homes use firewall to protect their networks from security breaches. Firewalls limit the communications between devices on a network.

Traditional time-sharing systems are rare today. 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).

Mobile computing:

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

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 and editing high-definition video.

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.

In several computer games that employ accelerometers, players interface with the system not by using a mouse or a keyboard but rather by tilting, rotating, and shaking the mobile device! Perhaps more a practical use of these features is found in augmented-reality applications, which overlay information on a display of the current environment. It is difficult to imagine how equivalent applications could be developed on traditional laptop or desktop computer systems.

To provide access to on-line services, mobile devices typically use either IEEE standard 802.11 wireless or cellular data networks. The memory capacity and processing speed of mobile devices, however, are more limited than those of PCs. Whereas a smartphone or tablet may have 256 GB in storage, it is not uncommon to find 8 TB in storage on a desktop computer.

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

Client –Server Computing:

Contemporary network architecture features arrangements in which server systems satisfy requests generated by client systems. This form of specialized distributed system, called a client–server system, has the general structure depicted in Figure 1.22.

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 serve 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. The actual contents of the files can vary greatly, ranging from traditional web pages to rich multimedia content such as high-definition video.

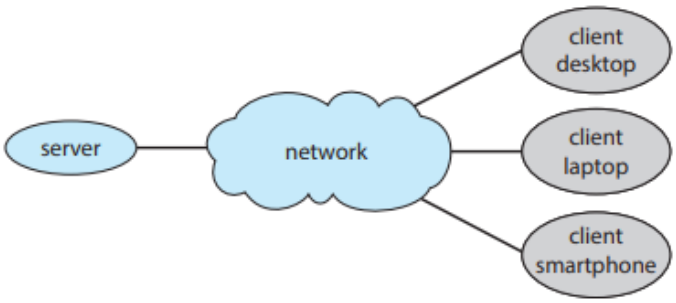


Figure 1.22 General structure of a client–server system.

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.

To participate in a peer-to-peer system, a node must first join the network of peers. Once a node has joined the network, it can begin providing services to—and requesting services from—other nodes in the network.

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

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

• 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 1.23 illustrates such a scenario.

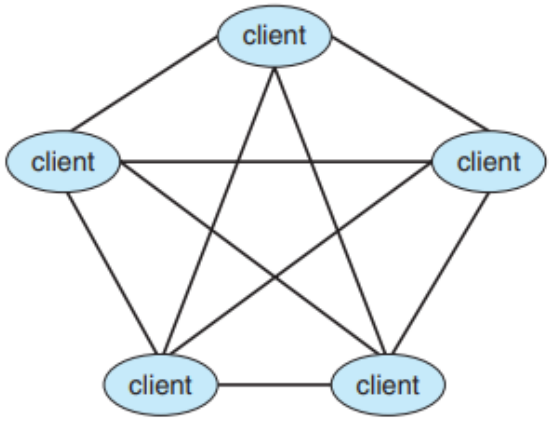


Figure 1.23 Peer-to-peer system with no centralized service.

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 peerto-peer approach. It includes a centralized login server, but it also incorporates decentralized peers and allows two peers to communicate.

Cloud Computing:

Cloud computing is a type of computing that delivers computing, storage, and even applications as a service across a network. In some ways, it's a logical extension of virtualization, because it uses virtualization as a base for its functionality. For example, the Amazon Elastic Compute Cloud (ec2) facility has thousands of servers, millions of virtual machines, and petabytes of storage available for use by anyone on the Internet. Users pay per month based on how much of those resources they use.

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

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

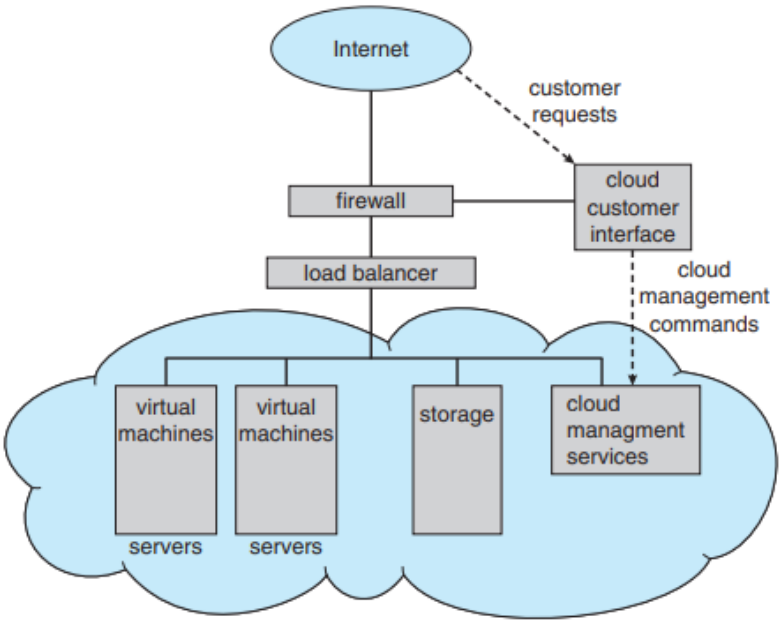


Figure 1.24 Cloud computing.

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

Certainly, there are traditional operating systems within many of the types of cloud infrastructure. Beyond those are the VMMs that manage the virtual machines in which the user processes run. At a higher level, the VMMs themselves are managed by cloud management tools, such as VMware vCloud Director and the open-source Eucalyptus toolset. These tools manage the resources within a given cloud and provide interfaces to the cloud components, making a good argument for considering them a new type of operating system.

Figure 1.24 illustrates a public cloud providing IaaS. Notice that both the cloud services and the cloud user interface are protected by a firewall.

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 optical drives 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.

These embedded systems vary considerably. Some are general-purpose computers, running standard operating systems—such as Linux—with special-purpose applications to implement the functionality. Others are hardware devices with a special-purpose embedded operating system providing just the functionality desired. Yet others are hardware devices with application-specific integrated circuits (ASICs) that perform their tasks without an operating system.

Embedded systems almost always run real-time operating systems. A realtime 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 analyse the data and possibly adjust controls to modify the sensor inputs. Systems that control scientific experiments, medical imaging systems, industrial control systems, and certain display systems are real-time systems. Some automobile-engine fuel-injection systems, home-appliance controllers, and weapon systems are also real-time systems.

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.

Operating-System Services:

An operating system provides an environment for the execution of programs. It makes certain services available to programs and to the users of those programs. The specific services provided, of course, differ from one operation system to another, but we can identify common classes. Figure 2.1 shows one view of the various operating-system services and how they interrelate. Note that these services also make the programming task easier for the programmer.

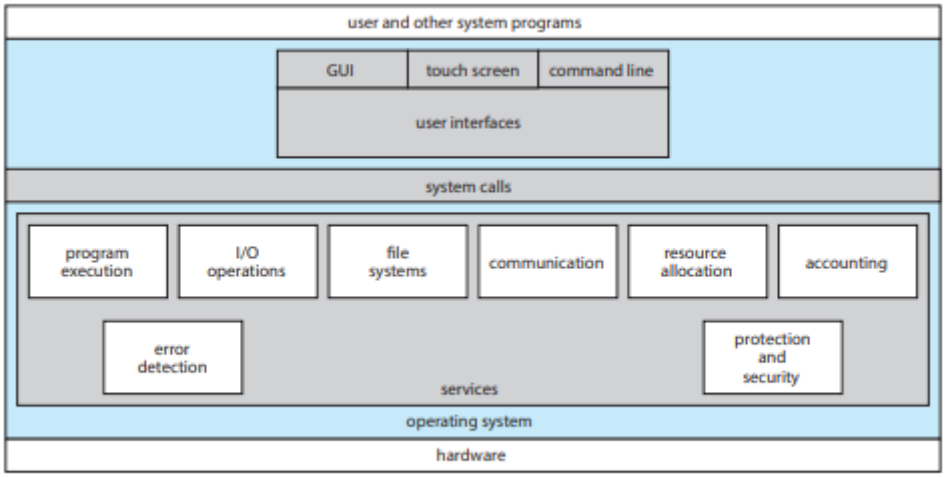


Figure 2.1 A view of operating system services.

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). This interface can take several forms. Most commonly, a graphical user interface (GUI) is used. Here, the interface is a window system with a mouse that serves as a pointing device to direct I/O, choose from menus, and make selections and a keyboard to enter text. Mobile systems such as phones and tablets provide a touch-screen interface, enabling users to slide their fingers across the screen or press buttons on the screen to select choices. Another option 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). Some systems provide two or all three of these variations.

• Program execution. The system must be able to load a program into memory and to run that program. The program must be able to end its execution, either normally or abnormally (indicating error). • I/O operations. A running program may require I/O, which may involve a file or an I/O device. For specific devices, special functions may be desired (such as reading from a network interface or writing to a file system). For efficiency and protection, users usually cannot control I/O devices directly. Therefore, the operating system must provide a means to do I/O.

• File-system manipulation. The file system is of particular interest. Obviously, programs need to read and write files and directories. They also need to create and delete them by name, search for a given file, and list file information. Finally, some operating systems include permissions management to allow or deny access to files or directories based on file ownership. Many operating systems provide a variety of file systems, sometimes to allow personal choice and sometimes to provide specific features or performance characteristics.

• Communications. There are many circumstances in which one process needs to exchange information with another process. Such communication may occur between processes that are executing on the same computer or between processes that are executing on different computer systems tied together by a network. Communications may be implemented via shared memory, in which two or more processes read and write to a shared section of memory, or message passing, in

which packets of information in predefined formats are moved between processes by the operating system.

• Error detection. The operating system needs to be detecting and correcting errors constantly. Errors may occur in the CPU and memory hardware (such as a memory error or a power failure), in I/O devices (such as a parity error on disk, a connection failure on a network, or lack of paper in the printer), and in the user program (such as an arithmetic overflow or 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. Sometimes, it has no choice but to halt the system. At other times, it might terminate an error-causing process or return an error code to a process for the process to detect and possibly correct.

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

• Resource allocation. When there are multiple processes running at the same time, resources must be allocated to each of them. The operating system manages many different types of resources. Some (such as CPU cycles, main memory, and file storage) may have special allocation code, whereas others (such as I/O devices) may have much more general request and release code. For instance, in determining how best to use the CPU, operating systems have CPU-scheduling routines that consider the speed of the CPU, the process that must be executed, the number of processing cores on the CPU, and other factors. There may also be routines to allocate printers, USB storage drives, and other peripheral devices.

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

• Protection and security. The owners of information stored in a multiuser or networked computer system may want to control use of that information. When several separate processes execute concurrently, it should not be possible for one process to interfere with the others or with the operating system itself. Protection involves ensuring that all access to system resources is controlled. Security of the system from outsiders is also important. Such security starts with requiring each user to authenticate himself or herself to the system, usually by means of a password, to gain access to system resources. It extends to defending external I/O devices, including network adapters, from invalid access attempts and recording all such connections for detection of break-ins. If a system is to be protected and secure, precautions must be instituted throughout it. A chain is only as strong as its weakest link.

System Calls:

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

Example:

Before we discuss how an operating system makes system calls available, let's first use an example to illustrate how system calls are used: writing a simple program to read data from one file and copy them to another file. One approach is to pass the names of the two files as part of the command— for example, the UNIX cp command:

cp in.txt out.txt

This command copies the input file in.txt to the output file out.txt. A second approach is for the program to ask the user for the names. In an interactive system, this approach will require a sequence of system calls, first to write a prompting message on the screen and then to read from the keyboard the characters that define the two files.

When both files are set up, we enter a loop that reads from the input file (a system call) and writes to the output file (another system call). Each read and write must return status information regarding various possible error conditions. On input, the program may find that the end of the file has been reached or that there was a hardware failure in the read (such as a parity error). The write operation may encounter various errors, depending on the output device (for example, no more available disk space). Finally, after the entire file is copied, the program may close both files (two system calls), write a message to the console or window (more system calls), and finally terminate normally (the final system call). This system-call sequence is shown in Figure 2.5.

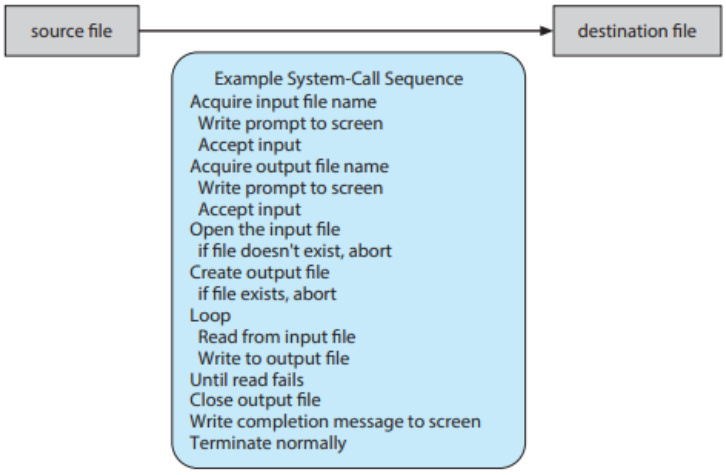


Figure 2.5 Example of how system calls are used.

Application Programming Interface:

Frequently, systems execute thousands of system calls per second. Most programmers never see this level of detail, however. Typically, application developers design programs according to an application programming interface (API). The API specifies a set of functions that are available to an application programmer, including the parameters that are passed to each function and the return values the programmer can expect. 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 macOS), 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.

EXAMPLE OF STANDARD API

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

```
man read
```

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

```
#include <unistd.h>

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

return value — function name — parameters

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

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

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

Another important factor in handling system calls is the run-time environment (RTE)— the full suite of software needed to execute applications written in each programming language, including its compilers or interpreters as well as other software, such as libraries and loaders.

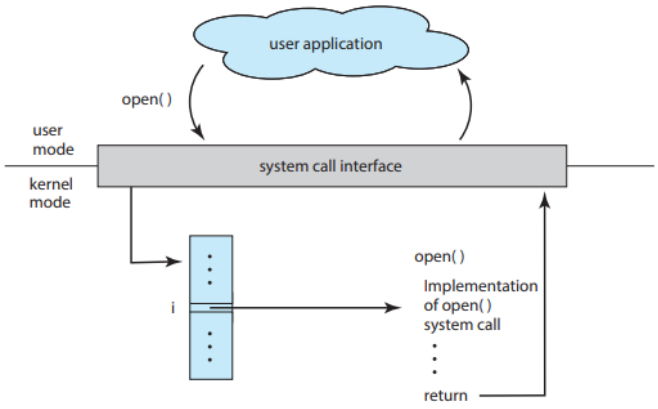


Figure 2.6 The handling of a user application invoking the `open()` system call.

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. The caller need know nothing about how the system call is implemented or what it does during execution. Rather, the caller need only obey the API and understand what the operating system will

do because of the execution of that system call. Thus, most of the details of the operating-system interface are hidden from the programmer by the API and are managed by the RTE. The relationship among 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.

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). Linux uses a combination of these approaches. If there are five or fewer parameters, registers are used. If there are more than five parameters, the block method is used. Parameters also can be placed, or pushed, onto a stack by the program and popped off the stack by the operating system. Some operating systems prefer the block or stack method because those approaches do not limit the number or length of parameters being passed.

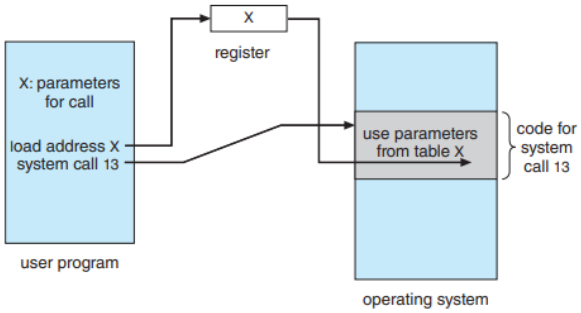


Figure 2.7 Passing of parameters as a table.

Types of System Calls:

System calls can be grouped roughly into six major categories: process control, fil management, device management, information maintenance, communications, and protection. Below, we briefly discuss the types of system calls that may be provided by an operating system. Figure 2.8 summarizes the types of system calls normally provided by an operating system. As mentioned, in this text, we normally refer to the system calls by generic names. Throughout the text, however, we provide examples of the actual counterparts to the system calls for UNIX, Linux, and Windows systems.

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 a special log file on disk and may be examined by a debugger—a system program designed to aid the programmer in finding and correcting errors, or bugs— to determine the cause of the problem. Under either normal or abnormal circumstances, the operating system must transfer control to the invoking command interpreter. The command interpreter then reads the next command.

- **Process control**
  - create process, terminate process
  - load, execute

- get process attributes, set process attributes
- wait event, signal event
- allocate and free memory

- **File management**
  - create file, delete file
  - open, close
  - read, write, reposition
  - get file attributes, set file attributes
- **Device management**
  - request device, release device
  - read, write, reposition
  - get device attributes, set device attributes
  - logically attach or detach devices
- **Information maintenance**
  - get time or date, set time or date
  - get system data, set system data
  - get process, file, or device attributes
  - set process, file, or device attributes
- **Communications**
  - create, delete communication connection
  - send, receive messages
  - transfer status information
  - attach or detach remote devices
- **Protection**
  - get file permissions
  - set file permissions

EXAMPLES OF WINDOWS AND UNIX SYSTEM CALLS

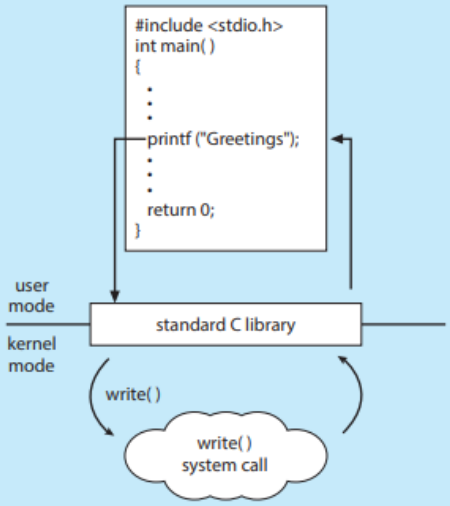
The following illustrates various equivalent system calls for Windows and UNIX operating systems.

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

A process 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 or the click of a mouse. An interesting question is where to return control when the loaded program terminates. This question is related to whether the existing program is lost, saved, or allowed to continue execution concurrently with the new program. If control returns to the existing program when the new program terminates, we must save the memory image of the existing program; thus, we have effectively created a mechanism for one program to call another program. If both programs continue concurrently, we have created a new process to be multiprogrammed. Often, there is a system call specifically for this purpose (create process ()).

THE STANDARD C LIBRARY

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



File Management:

We first need to be able to create () and delete () files. Either system call requires the name of the file and perhaps some of the file's attributes. Once the file is created, we need to open () it and to use it. We may also read (), write (), or reposition () (rewind or skip to the end of the file, for example). Finally, we need to close () the file, indicating that we are no longer using it. We may need these same sets of operations for directories if we have a directory structure for organizing files in the file system. In addition, for either files or directories, we need to be able to determine the values of various attributes and perhaps to set them if necessary. File attributes include the file name, file type, protection codes, accounting information, and so on. At least two system calls, get file attributes () and set file attributes (), are required for this function. Some operating systems provide many more calls, such as calls for file move () and copy (). Others might provide an API that performs those operations using code and other system calls, and others might provide system programs to perform the tasks. If the system

programs are callable by other programs, then each can be considered an API by other system programs.

Device Management:

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

The various resources controlled by the operating system can be thought of as devices. Some of these devices are physical devices (for example, disk drives), while others can be thought of as abstract or virtual devices (for example, files). A system with multiple users may require us to first request () a device, to ensure exclusive use of it. After we are finished with the device, we release () it. These functions are like the open () and close () system calls for files. Other operating systems allow unmanaged access to devices.

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

The user interface can also make files and devices appear to be similar, even though the underlying system calls are dissimilar. This is another example of the many design decisions that go into building an operating system and user interface.

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 version number of the operating system, the amount of free memory or disk space, and so on.

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

Communication:

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

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

In the shared-memory model, processes use shared memory create () and shared memory attach () system calls to create and gain access to regions of memory owned by other processes. Recall that, normally, the operating system tries to prevent one process from accessing another process's memory. Shared memory requires that two or more processes agree to remove this restriction. They can then exchange information by reading and writing data in the shared areas. The form of the data is determined by the processes and is not under the operating system's control. The processes are also responsible for ensuring that they are not writing to the same location simultaneously.

Protection:

Protection provides a mechanism for controlling access to the resources provided by a computer system. Historically, protection was a concern only on multiprogrammed computer systems with several users. However, with the advent of networking and the Internet, all computer systems, from servers to mobile handheld devices, must be concerned with protection. Typically, system calls providing protection include set permission () and get permission (), which manipulate the permission settings of resources such as files and disks. The allow user () and deny user () system calls specify whether particular users can—or cannot—be allowed access to certain resources.

System Services:

System services, also known as system utilities, provide a convenient environment for program development and execution. Some of them are simply user interfaces to system calls. Others are considerably more complex.

They can be divided into these categories:

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

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

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

files or perform transformations of the text.

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

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

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

• **Background services.** All general-purpose systems have methods for launching certain system-program processes at boot time. Some of these processes terminate after completing their tasks, while others continue to n until the system is halted. Constantly running system-program processes are known as services, subsystems, or daemons. One example is the network daemon discussed in Section 2.3.3.5. In that example, a system needed a service to listen for network connections to connect those requests to the correct processes.

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

Linkers and Loaders:

Source files are compiled into object files that are designed to be loaded into any physical memory location, a format known as a relocatable object fil. Next, the linker combines these relocatable object files into a single binary executable file. During the linking phase, other object files or libraries may be included as well, such as the standard C or math library (specified with the flag -lm).

A loader is used to load the binary executable file into memory, where it is eligible to run on a CPU core. An activity associated with linking and loading is relocation, which assigns final addresses to the program parts and adjusts code and data in the program to match those addresses so that, for example, the code can call library functions and access its variables as it executes. In Figure 2.11, we see that to run the loader, all that is necessary is to enter the name of the executable file on the command line. When a program name is entered on the command line on UNIX systems— for example,./ main— the shell first creates a new process to run the program using the fork() system call. The shell then invokes the loader with the exec () system call, passing exec () the name of the executable file. The loader then loads the specified program into memory using the address space of the newly created process.

Windows, for instance, supports dynamically linked libraries (DLLs). The benefit of this approach is that it avoids linking and loading libraries that may end up not being used into an executable file.

Object files and executable files typically have standard formats that include the compiled machine code and a symbol table containing metadata about functions and variables that are referenced in the program. For UNIX and Linux systems, this standard format is known as ELF (for Executable and Linkable Format). There are separate ELF formats for relocatable and executable files. One piece of information in the ELF file for executable files is the program's entry point, which contains the address of the first instruction to be executed when the program runs. Windows systems use the Portable Executable (PE) format, and macOS uses the Mach-O format.

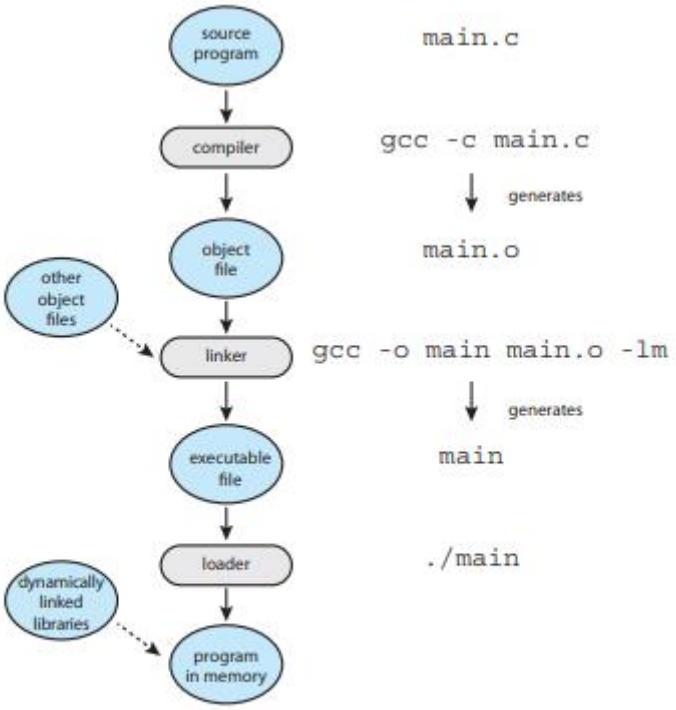


Figure 2.11 The role of the linker and loader.

Operating-System Design and Implementation:

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: traditional desktop/laptop, mobile, distributed, or real time.

Beyond this highest design level, the requirements may be much harder to specify. The requirements can, however, be divided into two basic groups: **user goals and system goals.**

Users want certain obvious properties in a system. The system should be convenient to use, easy to learn and to use, reliable, safe, and fast. Of course, these specifications are not particularly useful in the system design since there is no general agreement on how to achieve them.

A similar set of requirements can be defined by the developers who must design, create, maintain, and operate the system. The system should be easy to design, implement, and maintain; and it should

be flexible, reliable, error free, and efficient. Again, these requirements are vague and may be interpreted in various ways.

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**.

Mechanisms and Policies:

One important principle is the separation of policy from mechanism. Mechanisms determine how to do something; policies determine what will be done. For example, the timer construct (see Section 1.4.3) 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 flexible enough to work across a range of policies is preferable. A change in policy would then require redefinition of only certain parameters of the system. For instance, consider a mechanism for giving priority to certain types of programs over others. If the mechanism is properly separated from policy, it can be used either to support a policy decision that I/O-intensive programs should have priority over CPU-intensive ones or to support the opposite policy.

Policy decisions are important for all resource allocation. Whenever it is necessary to decide whether to allocate a resource, a policy decision must be made. Whenever the question is how rather than what, it is a mechanism that must be determined.

Implementation:

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

Early operating systems were written in assembly language. Now, most are written in higher-level languages such as C or C++, with small amount of the system written in assembly language. In fact, more than one higher level language is often used. The lowest levels of the kernel might be written in assembly language and C. Higher-level routines might be written in C and C++, and system libraries might be written in C++ or even higher-level languages. Android provides a nice example: its kernel is written mostly in C with some assembly language. Most Android system libraries are written in C or C++, and its application frameworks—which provide the developer interface to the system—are written mostly in Java.

The advantages of using a higher-level language, or at least a systems implementation language, for implementing operating systems are the same as those gained when the language is used for application programs: the code can be written faster, is more compact, and is easier to understand and debug. In addition, 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 other hardware if it is written in a higher-level language.

The only possible disadvantages of implementing an operating system in a higher-level language are reduced speed and increased storage requirements. This, however, is not a major issue in today's systems.

Operating-System Structure:

A system as large and complex as a modern operating system must be engineered carefully if it is to function properly and be modified easily. A common approach is to partition the task into small components, or modules, rather than have one single system. Each of these modules should be a well-defined portion of the system, with carefully defined interfaces and functions. You may use a similar approach when you structure your programs: rather than placing all of your code in the main() function, you instead separate logic into a number of functions, clearly articulate parameters and return values, and then call those functions from main().

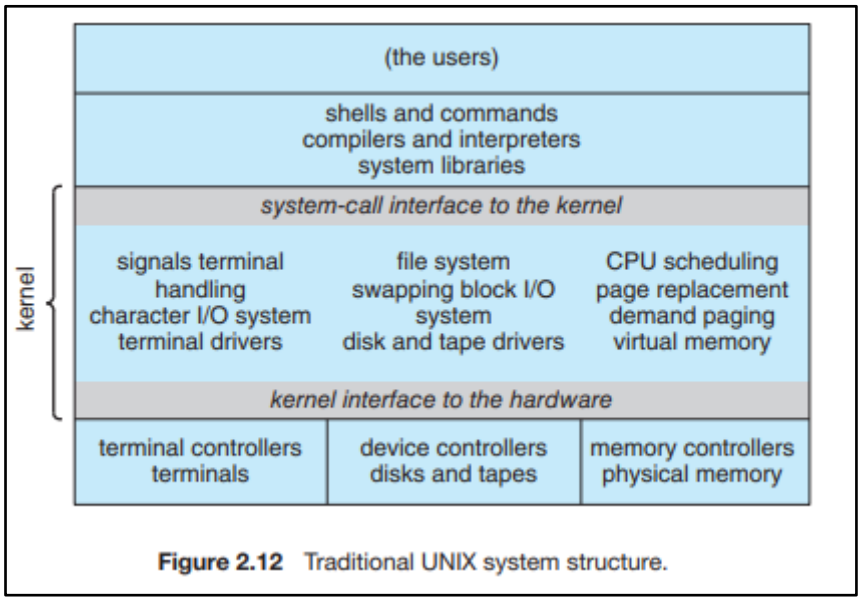


Figure 2.12 Traditional UNIX system structure.

Monolithic Structure:

The simplest structure for organizing an operating system is no structure at all. That is, place all the functionality of the kernel into a single, static binary file that runs in a single address space. This approach—known as a monolithic structure—is a common technique for designing operating systems.

An example of such limited structuring is the original UNIX operating system, which consists of two separable parts: the kernel and the system programs. The kernel is further separated into a series of interfaces and device drivers, which have been added and expanded over the years as UNIX has evolved. We can view the traditional UNIX operating system as being layered to some extent, as shown in Figure 2.12. Everything below the system-call interface and above the physical hardware is the kernel. The kernel provides the file system, CPU scheduling, memory management, and other operatingsystem functions through system calls. Taken in sum, that is an enormous amount of functionality to be combined into one single address space.

The Linux operating system is based on UNIX and is structured similarly, as shown in Figure 2.13.

Despite the apparent simplicity of monolithic kernels, they are difficult to implement and extend. Monolithic kernels do have a distinct performance advantage, however: there is very little overhead

in the system-call interface, and communication within the kernel is fast. Therefore, despite the drawbacks of monolithic kernels, their speed and efficiency explains why we still see evidence of this structure in the UNIX, Linux, and Windows operating systems.

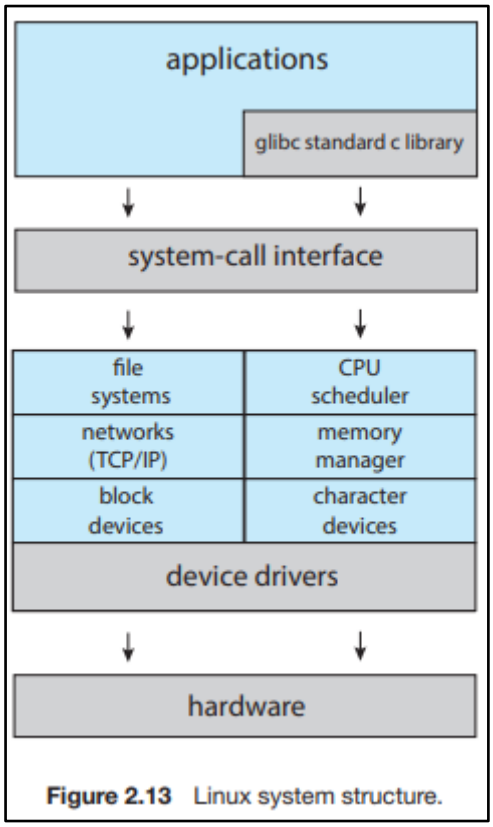


Figure 2.13 Linux system structure.

Layered Approach:

The monolithic approach is often known as a tightly coupled system because changes to one part of the system can have wide-ranging effects on other parts. Alternatively, we could design a loosely coupled system. Such a system is divided into separate, smaller components that have specific and limited functionality. All these components together comprise the kernel. The advantage of this modular approach is that changes in one component affect only that component, and no others, allowing system implementers more freedom in creating and changing the inner workings of the system.

A system can be made modular in many ways. One method is the layered approach, in which the operating system is broken into several layers (levels). The bottom layer (layer 0) is the hardware; the highest (layer N) is the user interface. This layering structure is depicted in Figure 2.14.

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.

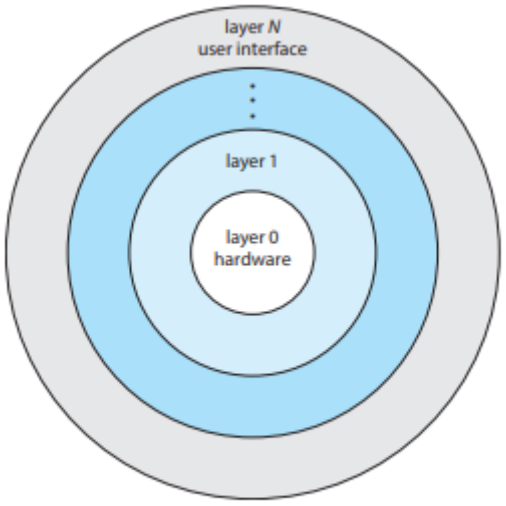


Figure 2.14 A layered operating system.

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

Layered systems have been successfully used in computer networks (such as TCP/IP) and web applications. Generally, these systems have fewer layers with more functionality, providing most of the advantages of modularized code while avoiding the problems of layer definition and interaction.

Microkernels We have already seen that the original UNIX system had a monolithic structure. As UNIX expanded, the kernel became large and difficult to manage. In the mid-1980s, researchers at Carnegie Mellon University developed an operating system called Mach that modularized the kernel using the micro kernel approach. This method structures the operating system by removing

Microkernels:

UNIX expanded, the kernel became large and difficult to manage. In the mid-1980s, researchers at Carnegie Mellon University developed an operating system called Mach that modularized the kernel using the microkernel approach. This method structures the operating system by removing all nonessential components from the kernel and implementing them as userlevel programs that reside in separate address spaces. The result is a smaller kernel. Figure 2.15 illustrates the architecture of a typical microkernel.

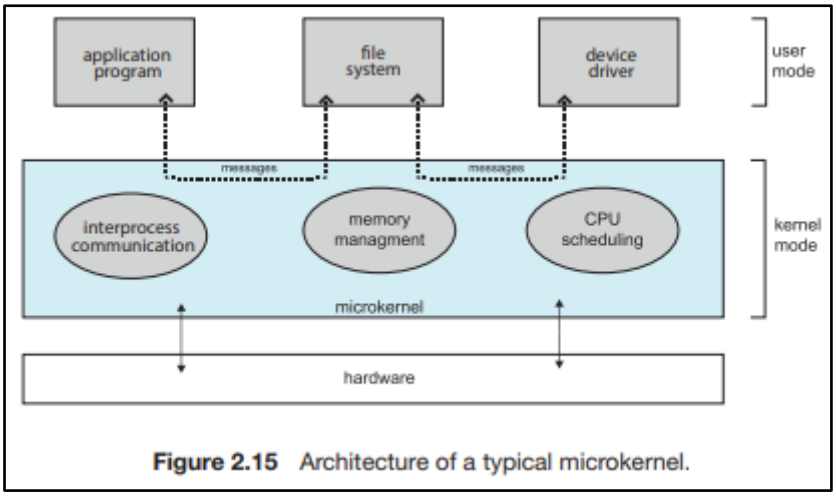


Figure 2.15 Architecture of a typical microkernel.

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

One benefit of the microkernel approach is that it makes extending the operating system easier. All new services are added to user space and consequently do not require modification of the kernel. When the kernel does have to be modified, the changes tend to be fewer, because the microkernel is a smaller kernel. The resulting operating system is easier to port from one hardware design to another. The microkernel also provides more security and reliability since most services are running as user—rather than kernel—processes. If a service fails, the rest of the operating system remains untouched.

Perhaps the best-known illustration of a microkernel operating system is Darwin, the kernel component of the macOS and iOS operating systems. Darwin, in fact, consists of two kernels, one of which is the Mach microkernel.

Another example is QNX, a real-time operating system for embedded systems. The QNX Neutrino microkernel provides services for message passing and process scheduling. It also handles low-level network communication and hardware interrupts. All other services in QNX are provided by standard processes that run outside the kernel in user mode.

Modules:

Perhaps the best current methodology for operating-system design involves using loadable kernel modules (LKMs). Here, the kernel has a set of core components and can link 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 Linux, macOS, and Solaris, 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.

Linux uses loadable kernel modules, primarily for supporting device drivers and file systems. LKMs can be “inserted” into the kernel as the system is started (or booted) or during run time, such as when a USB device is plugged into a running machine.

Hybrid Systems:

In practice, very few operating systems adopt a single, strictly defined structure. Instead, they combine different structures, resulting in hybrid systems that address performance, security, and usability issues. For example, Linux is monolithic, because having the operating system in a single address space provides very efficient performance. However, it also modular, so that new functionality can be dynamically added to the kernel. Windows is largely monolithic as well (again primarily for performance reasons), but it retains some behaviour typical of microkernel systems, including providing support for separate subsystems (known as operating-system personalities) that run as user-mode processes.

macOS and iOS:

Apple's macOS operating system is designed to run primarily on desktop and laptop computer systems, whereas iOS is a mobile operating system designed for the iPhone smartphone and iPad tablet computer. Architecturally, macOS and iOS have much in common, and so we present them together, highlighting what they share as well as how they differ from each other. The general architecture of these two systems is shown in Figure 2.16. Highlights of the various layers include the following:

- **User experience layer:** This layer defines the software interface that allows users to interact with the computing devices. macOS uses the Aqua user interface, which is designed for a mouse or trackpad, whereas iOS uses the Springboard user interface, which is designed for touch devices.
- **Application frameworks layer:** This layer includes the Cocoa and Cocoa Touch frameworks, which provide an API for the Objective-C and Swift programming languages. The primary difference between Cocoa and Cocoa Touch is that the former is used for developing macOS applications, and the latter by iOS to provide support for hardware features unique to mobile devices, such as touch screens.
- **Core frameworks:** This layer defines frameworks that support graphics and media including, QuickTime and OpenGL
- **Kernel environment:** This environment, also known as Darwin, includes the Mach microkernel and the BSD UNIX kernel. We will elaborate on Darwin shortly.

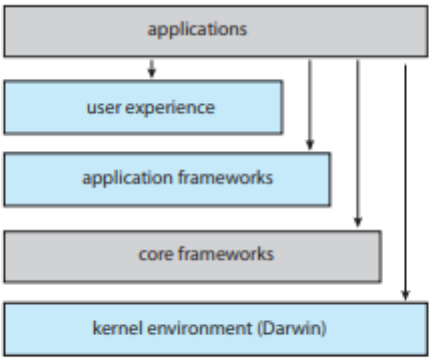


Figure 2.16 Architecture of Apple's macOS and iOS operating systems.

As shown in Figure 2.16, applications can be designed to take advantage of user-experience features or to bypass them and interact directly with either the application framework or the core framework.

Some significant distinctions between macOS and iOS include the following:

•Because macOS is intended for desktop and laptop computer systems, it is compiled to run on Intel architectures. iOS is designed for mobile devices and thus is compiled for ARM-based architectures. Similarly, the iOS kernel has been modified somewhat to address specific features and needs of mobile systems, such as power management and aggressive memory management. Additionally, iOS has more stringent security settings than macOS.

• The iOS operating system is generally much more restricted to developers than macOS and may even be closed to developers. For example, iOS restricts access to POSIX and BSD APIs on iOS, whereas they are openly available to developers on macOS.

We now focus on Darwin, which uses a hybrid structure. Darwin is a layered system that consists primarily of the Mach microkernel and the BSD UNIX kernel. Darwin's structure is shown in Figure 2.17.

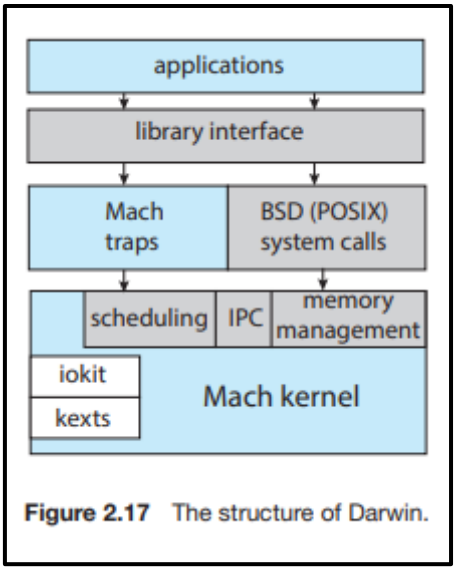


Figure 2.17 The structure of Darwin.

Apple has released the Darwin operating system as open source. As a result, various projects have added extra functionality to Darwin, such as the X-11 windowing system and support for additional file systems. Unlike Darwin, however, the Cocoa interface, as well as other proprietary Apple frameworks available for developing macOS applications, are closed.

Android:

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

Android is similar to iOS in that it is a layered stack of software that provides a rich set of frameworks supporting graphics, audio, and hardware features. These features, in turn, provide a platform for developing mobile applications that run on a multitude of Android-enabled devices. Software designers for Android devices develop applications in the Java language, but they do not generally use the standard Java API. Google has designed a separate Android API for Java development. Java applications are compiled into a form that can execute on the Android Runtime ART, a virtual machine designed for Android and optimized for mobile devices with limited memory and CPU processing capabilities. Java programs are first compiled to a Java bytecode .class file and then translated into an executable .dex file. Whereas many Java virtual machines perform just-in-time (JIT) compilation to improve application efficiency, ART performs ahead-of-time (AOT) compilation. Here, .dex files are compiled into native machine code when they are installed on a device, from which they can execute on the ART. AOT compilation allows more efficient application execution as well as reduced power consumption, features that are crucial for mobile systems.

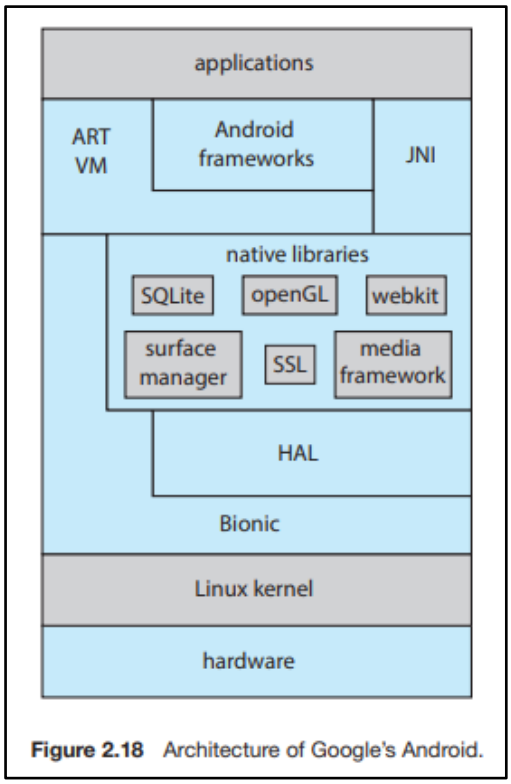


Figure 2.18 Architecture of Google's Android.

Android developers can also write Java programs that use the Java native interface—or JNI—which allows developers to bypass the virtual machine and instead write Java programs that can access specific hardware features. Programs written using JNI are generally not portable from one hardware device to another.

The set of native libraries available for Android applications includes frameworks for developing web browsers (webkit), database support (SQLite), and network support, such as secure sockets (SSLs).

Because Android can run on an almost unlimited number of hardware devices, Google has chosen to abstract the physical hardware through the hardware abstraction layer, or HAL. By abstracting all hardware, such as the camera, GPS chip, and other sensors, the HAL provides applications with a

consistent view independent of specific hardware. This feature, of course, allows developers to write programs that are portable across different hardware platforms.

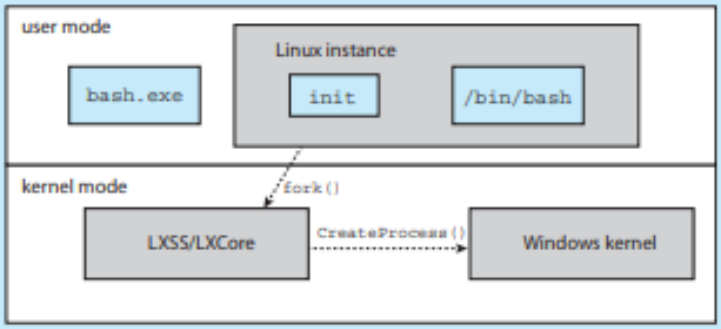
The standard C library used by Linux systems is the GNU C library (glibc). Google instead developed the Bionic standard C library for Android. Not only does Bionic have a smaller memory footprint than glibc, but it also has been designed for the slower CPUs that characterize mobile devices. (In addition, Bionic allows Google to bypass GPL licensing of glibc.)

At the bottom of Android's software stack is the Linux kernel. Google has modified the Linux kernel used in Android in a variety of areas to support the special needs of mobile systems, such as power management. It has also made changes in memory management and allocation and has added a new form of IPC known as Binder.

WINDOWS SUBSYSTEM FOR LINUX

Windows uses a hybrid architecture that provides subsystems to emulate different operating-system environments. These user-mode subsystems communicate with the Windows kernel to provide actual services. Windows 10 adds a Windows subsystem for Linux (WSL), which allows native Linux applications (specified as ELF binaries) to run on Windows 10. The typical operation is for a user to start the Windows application `bash.exe`, which presents the user with a bash shell running Linux. Internally, the WSL creates a Linux instance consisting of the `init` process, which in turn creates the `bash` shell running the native Linux application `/bin/bash`. Each of these processes runs in a Windows Pico process. This special process loads the native Linux binary into the process's own address space, thus providing an environment in which a Linux application can execute.

Pico processes communicate with the kernel services LXCore and LXSS to translate Linux system calls, if possible using native Windows system calls. When the Linux application makes a system call that has no Windows equivalent, the LXSS service must provide the equivalent functionality. When there is a one-to-one relationship between the Linux and Windows system calls, LXSS forwards the Linux system call directly to the equivalent call in the Windows kernel. In some situations, Linux and Windows have system calls that are similar but not identical. When this occurs, LXSS will provide some of the functionality and will invoke the similar Windows system call to provide the remainder of the functionality. The Linux `fork()` provides an illustration of this: The Windows `CreateProcess()` system call is similar to `fork()` but does not provide exactly the same functionality. When `fork()` is invoked in WSL, the LXSS service does some of the initial work of `fork()` and then calls `CreateProcess()` to do the remainder of the work. The figure below illustrates the basic behavior of WSL.



Process Concept:

A question that arises in discussing operating systems involves what to call all the CPU activities. Early computers were batch systems that executed jobs, followed by the emergence of time-shared systems that ran user programs, or tasks. Even on a single-user system, a user may be able to run several programs at one time: a word processor, a web browser, and an e-mail package. And even if a computer can execute only one program at a time, such as on an embedded device that does not support multitasking, the operating system may need to support its own internal programmed activities, such as memory management. In many respects, all these activities are similar, so we call all of them **processes.**

The Process:

Informally, as mentioned earlier, a process is a program in execution. The status of the current activity of a process is represented by the value of the program counter and the contents of the processor's registers. The memory layout of a process is typically divided into multiple sections and is shown in Figure 3.1. These sections include:

- Text section— the executable code
- Data section—global variables
- Heap section—memory that is dynamically allocated during program run time
- Stack section— temporary data storage when invoking functions (such as function parameters, return addresses, and local variables).

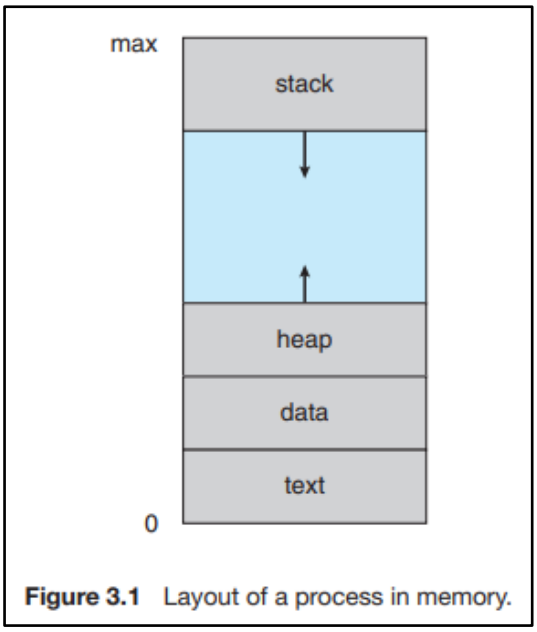
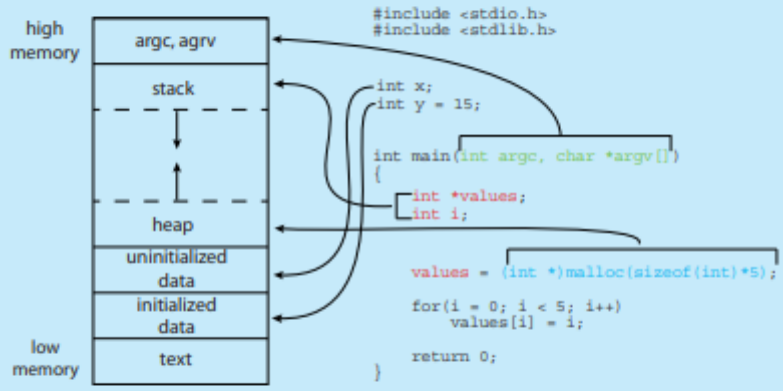


Figure 3.1 Layout of a process in memory.

MEMORY LAYOUT OF A C PROGRAM

The figure shown below illustrates the layout of a C program in memory, highlighting how the different sections of a process relate to an actual C program. This figure is similar to the general concept of a process in memory as shown in Figure 3.1, with a few differences:

- The global data section is divided into different sections for (a) initialized data and (b) uninitialized data.
- A separate section is provided for the `argc` and `argv` parameters passed to the `main()` function.



The GNU `size` command can be used to determine the size (in bytes) of some of these sections. Assuming the name of the executable file of the above C program is `memory`, the following is the output generated by entering the command `size memory`:

```
text    data    bss     dec     hex     filename
1158    284     8       1450    5aa     memory
```

The `data` field refers to uninitialized data, and `bss` refers to initialized data. (`bss` is a historical term referring to ***block started by symbol***.) The `dec` and `hex` values are the sum of the three sections represented in decimal and hexadecimal, respectively.

Process State:

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

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

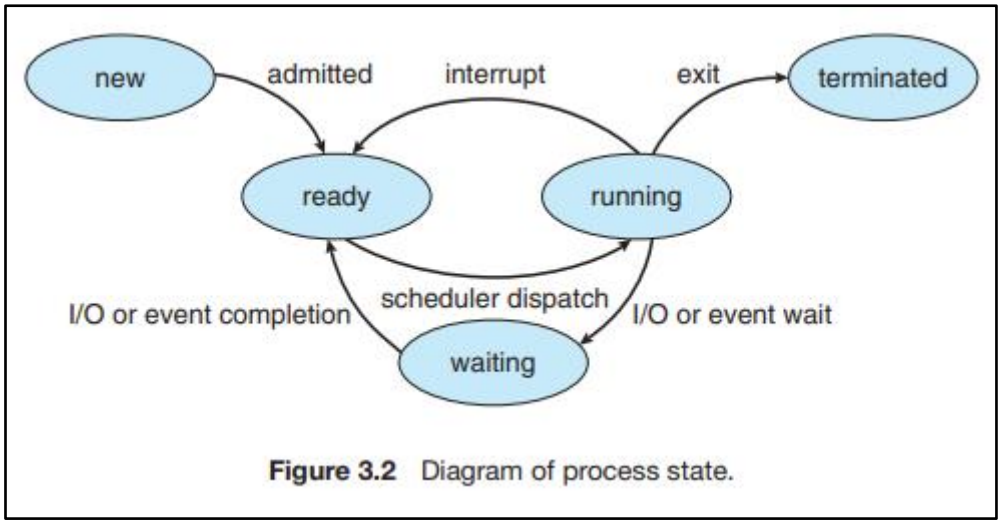


Figure 3.2 Diagram of process state.

Process Control Block:

Each process is represented in the operating system by a process control block (PCB)—also called a task control block. A PCB is shown in Figure 3.3. It contains many pieces of information associated with a specific process, including these:

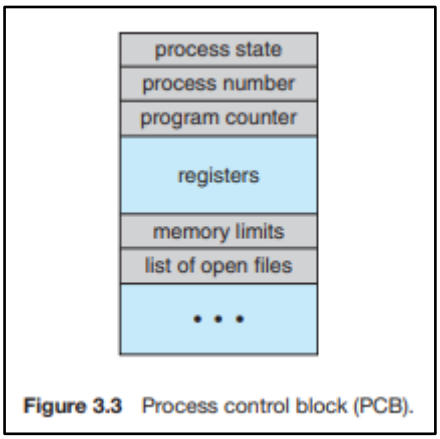


Figure 3.3 Process control block (PCB).

- **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 when it is rescheduled to run.

- **CPU-scheduling information:** This information includes a process priority, pointers to scheduling queues, and any other scheduling parameters. (Chapter 5 describes process scheduling.)
- **Memory-management information:** This information may include such items as the value of the base and limit registers and the page tables, or the segment tables, depending on the memory system used by the operating system .
- **Accounting information:** This information includes the amount of CPU and real time used, time limits, account numbers, job, or process numbers, and so on.
- **I/O status information:** This information includes the list of I/O devices allocated to the process, a list of open files, and so on.

Context Switch:

Interrupts cause the operating system to change a CPU core from its current task and to run a kernel routine. Such operations happen frequently on general-purpose systems. When an interrupt occurs, the system needs to save the current context of the process running on the CPU core so that it can restore that context when its processing is done, essentially suspending the process, and then resuming it. The context is represented in the PCB of the process. It includes the value of the CPU registers, the process state (see Figure 3.2), and memory-management information. Generically, we perform a state save of the current state of the CPU core, be it in kernel or user mode, and then a state restores to resume operations. Switching the CPU core 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 and is illustrated in Figure 3.6. When a context switch occurs, the kernel saves the context of the old process in its PCB and loads the saved context of the new process scheduled to run. Context switch time is pure overhead because the system does no useful work while switching. Switching speed varies from machine to machine, depending on the memory speed, the number of registers that must be copied, and the existence of special instructions (such as a single instruction to load or store all registers). A typical speed is a several microseconds.

Context-switch times are highly dependent on hardware support. For instance, some processors provide multiple sets of registers. A context switch here simply requires changing the pointer to the current register set. Of course, if there are more active processes than there are register sets, the system resorts to copying register data to and from memory, as before. Also, the more complex the operating system, the greater the amount of work that must be done during a context switch. As we will see in Chapter 9, advanced memory-management techniques may require that extra data be switched with each context. For instance, the address space of the current process must be preserved as the space of the next task is prepared for use. How the address space is preserved, and what amount of work is needed to preserve it, depend on the memorymanagement method of the operating system.

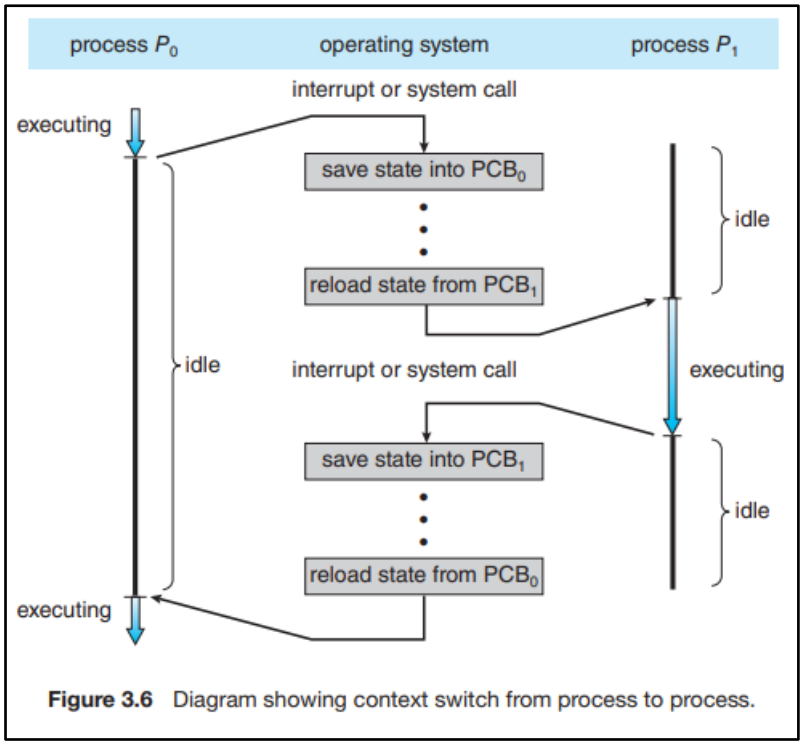


Figure 3.6 Diagram showing context switch from process to process.

Operations on Processes:

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

Process Creation:

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

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

Figure 3.7 illustrates a typical process tree for the Linux operating system, showing the name of each process and its pid. (We use the term process rather loosely in this situation, as Linux prefers the term task instead.) The systemd process (which always has a pid of 1) serves as the root parent process for all user processes and is the first user process created when the system boots. Once the system has booted, the systemd process creates processes which provide additional services such as a web or print server, an ssh server, and the like. In Figure 3.7, we see two children of systemd—logind and sshd. The logind process is responsible for managing clients that directly log onto the system. In this example, a client has logged on and is using the bash shell, which has been assigned pid 8416. Using the bash command-line interface, this user has created the process ps as well as the vim editor. The

sshd process is responsible for managing clients that connect to the system by using ssh (which is short for secure shell).

On UNIX and Linux systems, we can obtain a listing of processes by using the ps command. For example, the command

ps -el

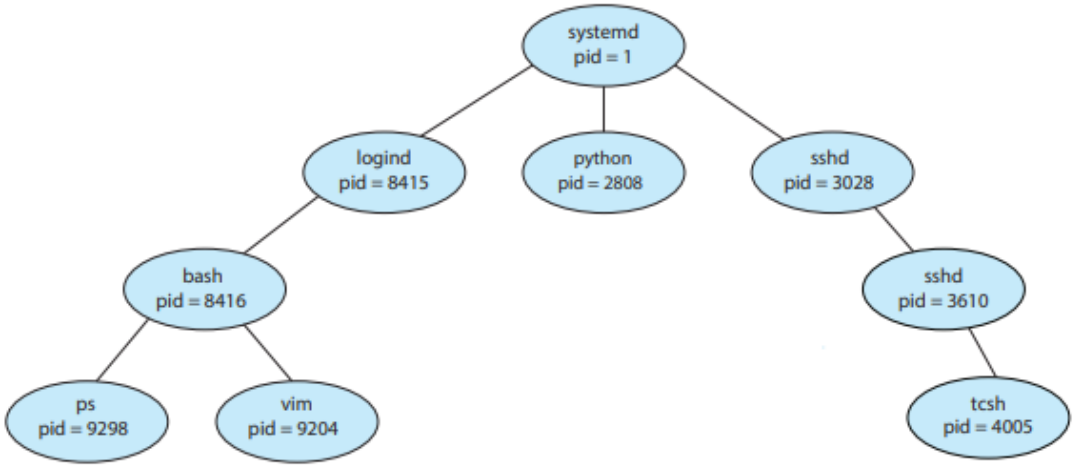


Figure 3.7 A tree of processes on a typical Linux system.

THE init AND systemd PROCESSES

Traditional UNIX systems identify the process `init` as the root of all child processes. `init` (also known as System V `init`) is assigned a pid of 1, and is the first process created when the system is booted. On a process tree similar to what is shown in Figure 3.7, `init` is at the root.

Linux systems initially adopted the System V `init` approach, but recent distributions have replaced it with `systemd`. As described in Section 3.3.1, `systemd` serves as the system's initial process, much the same as System V `init`; however it is much more flexible, and can provide more services, than `init`.

On such a system, the new process may get two open files, hw1.c and the terminal device, and may simply transfer the datum between the two.

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

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

2. The parent waits until some or all its children have terminated.

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

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

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

To illustrate these differences, let's first consider the UNIX operating system. In UNIX, as we've seen, each process is identified by its process identifier, which is a unique integer. A new process is created by the fork() system call. The new process consists of a copy of the address space of the original process. This mechanism allows the parent process to communicate easily with its child process. Both processes (the parent and the child) continue execution at the instruction after the fork(), with one difference: the return code for the fork() is zero for the new (child) process, whereas the (nonzero) process identifier of the child is returned to the parent.

After a fork() system call, one of the two processes typically uses the exec() system call to replace the process's memory space with a new program. The exec() system call loads a binary file into memory (destroying the memory image of the program containing the exec() system call) and starts its execution. In this manner, the two processes can communicate and then go their separate ways. 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. Because the call to exec() overlays the process's address space with a new program, exec() does not return control unless an error occurs.

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

int main()
{
pid_t pid;

    /* fork a child process */
    pid = fork();

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

    return 0;
}
```

Figure 3.8 Creating a separate process using the UNIX `fork()` system call.

The C program shown in Figure 3.10 illustrates the CreateProcess() function, which creates a child process that loads the application mspaint.exe. We opt for many of the default values of the ten parameters passed to CreateProcess(). Readers interested in pursuing the details of process creation and management in the Windows API are encouraged to consult the biblio graphical notes at the end of this chapter.

The two parameters passed to the CreateProcess() function are instances of the STARTUPINFO and PROCESS INFORMATION structures. STARTUPINFO specifies many properties of the new process, such as windows size and appearance and handles to standard input and output files. The PROCESS INFORMATION structure contains a handle and the identifiers to the newly created process and its thread. We invoke the ZeroMemory() function to allocate memory for each of these structures before proceeding with CreateProcess().

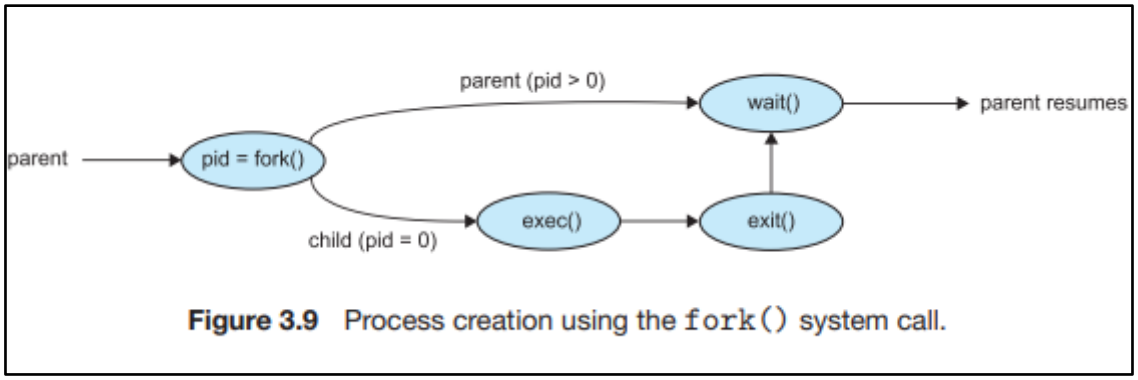


Figure 3.9 Process creation using the `fork()` system call.

```
#include <stdio.h>
#include <windows.h>

int main(VOID)
{
STARTUPINFO si;
PROCESS_INFORMATION pi;

    /* allocate memory */
    ZeroMemory(&si, sizeof(si));
    si.cb = sizeof(si);
    ZeroMemory(&pi, sizeof(pi));

    /* create child process */
    if (!CreateProcess(NULL, /* use command line */
      "C:\\WINDOWS\\system32\\mspaint.exe", /* command */
      NULL, /* don't inherit process handle */
      NULL, /* don't inherit thread handle */
      FALSE, /* disable handle inheritance */
      0, /* no creation flags */
      NULL, /* use parent's environment block */
      NULL, /* use parent's existing directory */
      &si,
      &pi))
    {
       fprintf(stderr, "Create Process Failed");
       return -1;
    }
    /* parent will wait for the child to complete */
    WaitForSingleObject(pi.hProcess, INFINITE);
    printf("Child Complete");

    /* close handles */
    CloseHandle(pi.hProcess);
    CloseHandle(pi.hThread);
}
```

Figure 3.10 Creating a separate process using the Windows API.

Process Termination:

A process terminates when it finishes executing its final statement and asks the operating system to delete it by using the exit() system call. At that point, the process may return a status value (typically an integer) to its waiting parent process (via the wait() system call). All the resources of the process — including physical and virtual memory, open files, and I/O buffers—are deallocated and reclaimed by the operating system.

Termination can occur in other circumstances as well. A process can cause the termination of another process via an appropriate system call (for example, TerminateProcess() in Windows). Usually, such a system call can be invoked only by the parent of the process that is to be terminated. Otherwise, a user— or a misbehaving application—could arbitrarily kill another user's processes. Note that a parent needs to know the identities of its children if it is to terminate them. Thus, when one process creates a new process, the identity of the newly created process is passed to the parent.

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

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

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

Inter process Communication:

Processes executing concurrently in the operating system may be either independent processes or cooperating processes. A process is independent if it does not share data with any 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. Clearly, any process that shares data with other processes is a cooperating process.

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

- **Information sharing:** Since several applications may be interested in the same piece of information (for instance, copying and pasting), we must provide an environment to allow concurrent access to such information.
- **Computation speedup**: If we want a particular task to run faster, we must break it into subtasks, each of which will be executing in parallel with the others. Notice that such a speedup can be achieved only if the computer has multiple processing cores.
- **Modularity**: We may want to construct the system in a modular fashion, dividing the system functions into separate processes or threads.

Cooperating processes require an inter process communication (IPC) mechanism that will allow them to exchange data— that is, send data to and receive data from each other. There are two fundamental models of inter process communication: shared memory and message passing. In the shared-memory model, a region of memory that is shared by the cooperating processes is established. Processes can then exchange information by reading and writing data to the shared region. In the message-passing model, communication takes place by means of messages exchanged between the cooperating processes. The two communications models are contrasted in Figure 3.11.

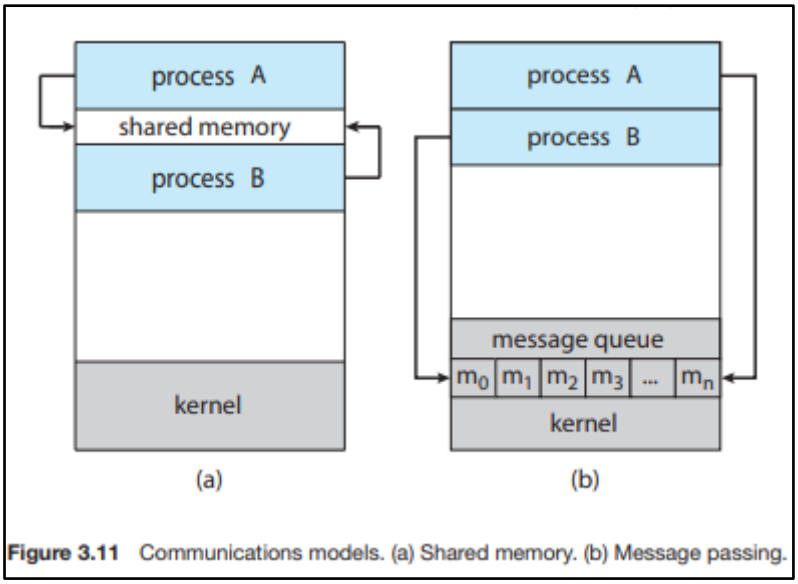


Figure 3.11 Communications models. (a) Shared memory. (b) Message passing.

IPC in Shared-Memory Systems:

Inter process 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. 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 these processes and are not under the operating system's control. The processes are also responsible for ensuring that they are not writing to the same location simultaneously.

To illustrate the concept of cooperating processes, let's consider the **producer–consumer problem,** which is a common paradigm for cooperating processes. A producer process produces information that is consumed by a consumer process. For example, a compiler may produce assembly code that is consumed by an assembler. The assembler, in turn, may produce object modules that are consumed by the loader. The producer–consumer problem also provides a useful metaphor for the client–server paradigm. We generally think of a server as a producer and a client as a consumer. For example, a web server produces (that is, provides) web content such as HTML files and images, which are consumed (that is, read) by the client web browser requesting the resource.

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

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

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.

```
item next_produced;

while (true) {
     /* produce an item in next_produced */

     while (((in + 1) % BUFFER_SIZE) == out)
        ; /* do nothing */

     buffer[in] = next_produced;
     in = (in + 1) % BUFFER_SIZE;
}
```

Figure 3.12 The producer process using shared memory.

```
item next_consumed;

while (true) {
     while (in == out)
        ; /* do nothing */

     next_consumed = buffer[out];
     out = (out + 1) % BUFFER_SIZE;

     /* consume the item in next_consumed */
}
```

Figure 3.13 The consumer process using shared memory.

IPC in Message-Passing Systems:

As we already seen how cooperating processes can communicate in a shared-memory environment. The scheme requires that these processes share a region of memory and that the code for accessing and manipulating the shared memory be written explicitly by the application programmer. Another way to achieve the same effect is for the operating system to provide the means for cooperating processes to communicate with each other via a message-passing facility.

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

another by exchanging messages.

A message-passing facility provides at least two operations:

send(message)

and

receive(message)

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

If processes P and Q want to communicate, they must send messages to and receive messages from each other: a communication link must exist between them. This link can be implemented in a variety of ways. We are concerned here not with the link's physical implementation (such as shared memory, hardware bus, or network, which are covered in Chapter 19) but rather with its logical implementation. Here are several methods for logically implementing a link and the send()/receive() operations:

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

```
message next_produced;

while (true) {
     /* produce an item in next_produced */

     send(next_produced);
}
```

Figure 3.14 The producer process using message passing.

```
message next_consumed;

while (true) {
     receive(next_consumed);

     /* consume the item in next_consumed */
}
```

Figure 3.15 The consumer process using message passing.

Naming:

Processes that want to communicate must have a way to refer to each other. They can use either direct or indirect communication. Under direct communication, each process that wants to communicate must explicitly name the recipient or sender of the communication.

In this scheme, the send() and receive() primitives are defined as:

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

A communication link in this scheme has the following properties:

- A link is established automatically between every pair of processes that want to communicate. The processes need to know only each other's identity to communicate
- A link is associated with exactly two processes.
- Between each pair of processes, there exists exactly one link.

This scheme exhibits symmetry in addressing; that is, both the sender pro cess 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 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. All references to the old identifier must be found, so that they can be modified to the new identifier. In general, any such hard-coding techniques, where identifiers must be explicitly stated, are less desirable than techniques involving indirection, as described next.

With indirect communication, the messages are sent to and received from mailboxes, or ports. A mailbox can be viewed abstractly as an object into which messages can be placed by processes and from which messages can be removed. Each mailbox has a unique identification. For example, POSIX message queues use an integer value to identify a mailbox. A process can communicate with another process via several different mailboxes, but two processes can communicate only if they have a shared mailbox.

The send() and receive() primitives are defined as follows:

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

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

- A link is established between a pair of processes only if both members of the pair have a shared mailbox.
- A link may be associated with more than two processes.

- Between each pair of communicating processes, several different links may exist, with each link corresponding to one mailbox.

Now suppose that processes P1, P2, and P3 all share mailbox A. Process P1 sends a message to A, while both P2 and P3 execute a receive() from A. Which process will receive the message sent by P1? The answer depends on which of the following methods we choose:

- Allow a link to be associated with two processes at most
- Allow at most one process at a time to execute a receive() operation.
- Allow the system to select arbitrarily which process will receive the message (that is, either P2 or P3, but not both, will receive the message). The system may define an algorithm for selecting which process will receive the message (for example, round robin, where processes take turns receiving messages). The system may identify the receiver to the sender.

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

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

Synchronization:

Communication between processes takes place through calls to send () and receive() primitives. There are different design options for implementing each primitive. Message passing may be either blocking or nonblocking— also known as synchronous and asynchronous. (Throughout this text, you will encounter the concepts of synchronous and asynchronous behaviour in relation to various operating-system algorithms.)

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

Different combinations of send() and receive() are possible. When both send() and receive() are blocking, we have a rendezvous between the sender and the receiver. The solution to the producer–consumer problem becomes trivial when we use blocking send() and receive() statements. The producer merely invokes the blocking send() call and waits until the message is delivered to either the receiver or the mailbox. Likewise, when the consumer invokes receive(), it blocks until a message is available. This is illustrated in Figures 3.14 and 3.15.

Buffering:

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

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

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