Notes for Operating System Concepts

CHAPTER 1: Introduction

An operating system acts as an intermediary between the user of a computer and the computer hardware. The purpose of an operating system is to provide an environment in which a user can execute programs in a convenient and efficient manner.
1.1 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).

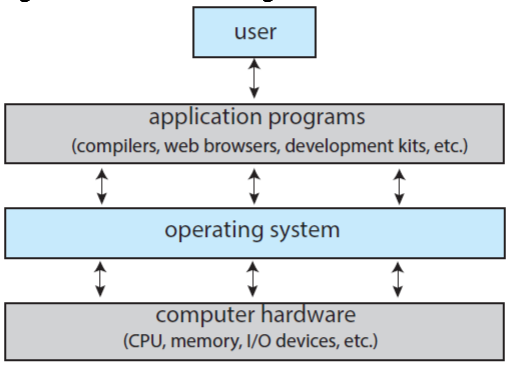

user
application programs
(compilers, web browsers, development kits, etc.)
operating system
computer hardware
(CPU, memory, I/O devices, etc.)

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.
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.
System View
From the computer's point of view, the operating system is the **program** most intimately involved with the hardware. In this context, we can view an operating system as a resource allocator.
Defining Operating Systems
In addition, we have no universally accepted definition of what is part of the operating system. A more common definition, and the one that we usually follow, is that the operating system is the **one program** running at **all times** on the computer—usually called the **kernel**. **Along with** the kernel, there are **two other types** of programs: system programs, which are associated with the operating system but are **not necessarily** part of the kernel, and application programs, which include **all** programs **not** associated with the operation of the system.
Mobile operating systems often include **not only** a core kernel but also middleware—a set of software frameworks that provide additional services to application developers.
In summary, for our purposes, the operating system includes the **always** running kernel, middleware frameworks that ease application development and provide features, and system programs that aid in managing the system while it is running.
1.2 Computer-System Organization
A modern general-purpose computer system consists of one or more CPUs and a number of device controllers connected through a common bus that provides access between components and shared memory (Figure 1.2).

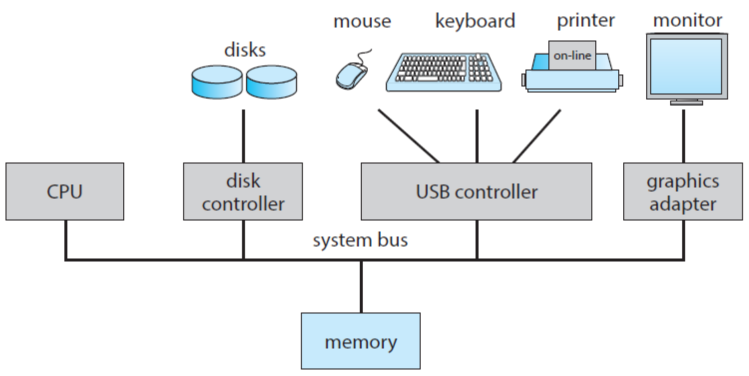


Figure 1. 2 A typical PC computer system.

Typically, operating systems have **a** device driver for **each device controller**. This device driver **understands** the device controller and provides the rest of the operating system with a uniform interface to the device.

Interrupts

Consider a typical computer operation: a program performing I/O. To start an I/O operation, the device driver loads the appropriate registers in the device controller. The device controller, in turn, examines the contents of these registers to determine what action to take (such as "read a character from the keyboard"). The **controller** starts the transfer of data from the **device** to its local buffer. Once the transfer of data is complete, the device controller **informs** the device driver that it has finished its operation. The device driver then **gives control** to other parts of the operating system, possibly returning the data or a pointer to the data if the operation was a read. For other operations, the device driver returns status information such as "write completed successfully" or "device busy". But how does the controller inform the device driver that it has finished its operation? This is accomplished via an interrupt.

Overview

Interrupts are used for many other purposes as well and are a **key part** of how operating systems and hardware **interact**.

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

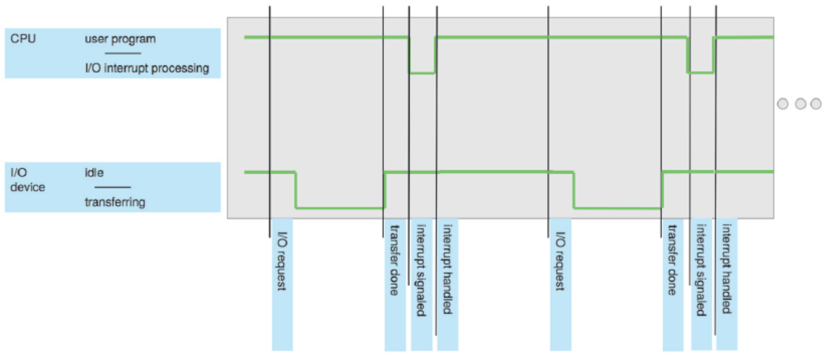


Figure 1. 3 Interrupt timeline for a single program doing output.

Each computer design has its own interrupt mechanism, but several functions are **common**. The interrupt must transfer control to the **appropriate interrupt service routine**. The straightforward method for managing this transfer would be to invoke a generic **routine** to examine the **interrupt information**. The routine, in turn, would call the **interrupt-specific handler**. However, interrupts must be handled **quickly**, as they occur very frequently. A table of pointers to interrupt routines can be used instead to provide the necessary speed. The interrupt routine is called **indirectly** through the table, with no intermediate routine needed.

Generally, the table of pointers is stored in low memory (the first hundred or so locations). These locations hold the addresses of the interrupt service routines for the various devices. This array, or interrupt vector, of addresses is then indexed by a unique number, given with the interrupt request, to provide the address of the interrupt service routine for the interrupting device.
The interrupt architecture must also save the state information of whatever was interrupted, so that it can restore this information after servicing the interrupt.

Implementation

The **basic interrupt mechanism** works as follows. The CPU hardware has a wire called the **interrupt-request line** that the CPU **senses after** executing every **instruction**. When the CPU detects that a controller has asserted a signal on the interrupt-request line, it reads the interrupt number and jumps to the interrupt-handler routine by using that interrupt number as an index into the interrupt vector.
We say that the device controller ***raises*** an interrupt by asserting a signal on the interrupt request line, the CPU ***catches*** the interrupt and ***dispatches*** it to the interrupt handler, and the handler ***clears*** the interrupt by servicing the device. Figure 1.4 summarizes the interrupt-driven I/O cycle.

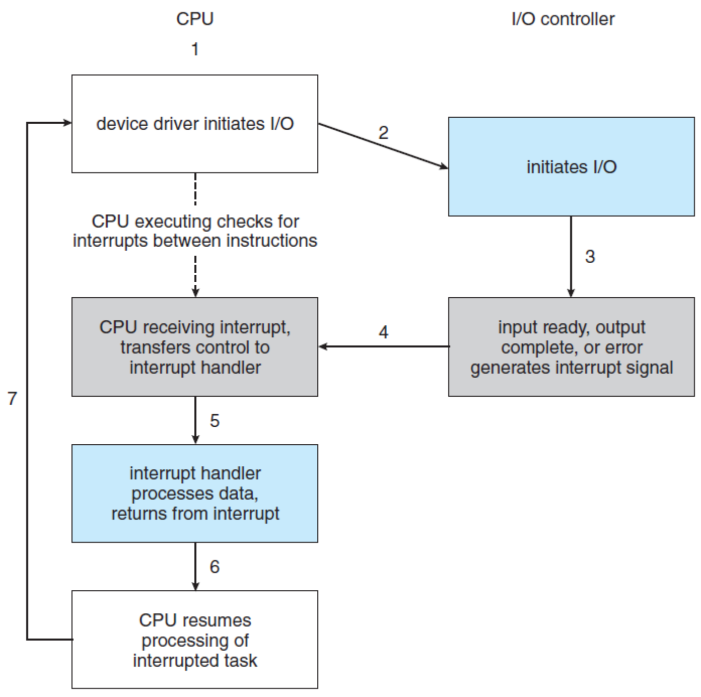
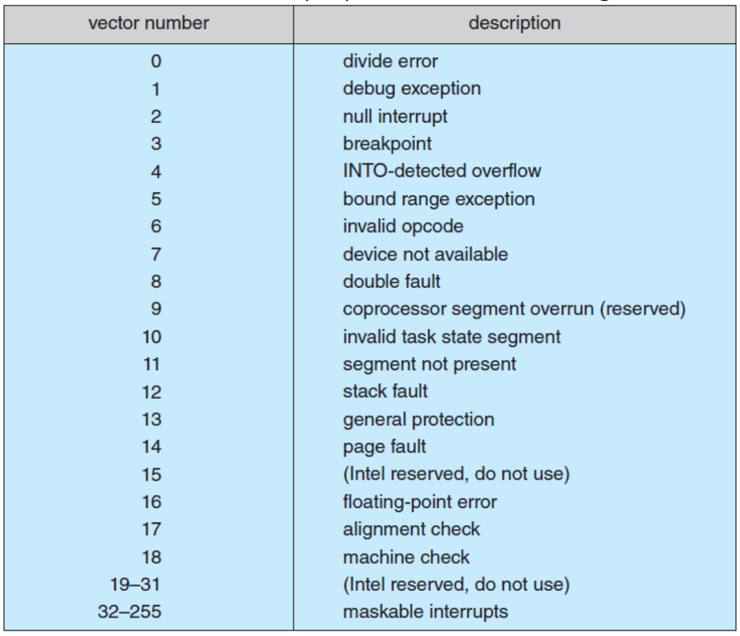

CPU
I/O controller
1
device driver initiates I/O
2
initiates I/O
CPU executing checks for interrupts between instructions
3
CPU receiving interrupt, transfers control to interrupt handler
4
input ready, output complete, or error generates interrupt signal
7
5
interrupt handler processes data, returns from interrupt
6
CPU resumes processing of interrupted task

Figure 1. 4 Interrupt-driven I/O cycle.

The basic interrupt mechanism just described enables the CPU to respond to an **asynchronous** event, as when a device controller becomes ready for service. In a modern operating system, however, we need more **sophisticated** interrupt handling features:
1. We need the ability to defer interrupt handling during critical processing.
2. We need an efficient way to dispatch to the proper interrupt handler for a device.
3. We need multilevel interrupts, so that the operating system can distinguish between high- and low-priority interrupts and can respond with the appropriate degree of urgency.

In modern computer hardware, these three features are provided by the CPU and the interrupt-controller hardware.
Most CPUs have two interrupt request lines. One is the nonmaskable interrupt, which is reserved for events such as unrecoverable memory errors. The second interrupt line is maskable: it can be turned off by the CPU before the execution of critical instruction sequences that must not be interrupted. The maskable interrupt is used by device controllers to request service.
In practice, however, computers have more devices (and, hence, interrupt handlers) **than** they have address elements in the interrupt vector. A **common way** to solve this problem is to use

interrupt chaining, in which each element in the interrupt vector points to the head of a list of interrupt handlers.
Figure 1.5 illustrates the design of the interrupt vector for **Intel processors**. The events from 0 to 31, which are nonmaskable, are used to signal various error conditions. The events from 32 to 255, which are maskable, are used for purposes such as device-generated interrupts.



vector number	description
0	divide error
1	debug exception
2	null interrupt
3	breakpoint
4	INTO-detected overflow
5	bound range exception
6	invalid opcode
7	device not available
8	double fault
9	coprocessor segment overrun (reserved)
10	invalid task state segment
11	segment not present
12	stack fault
13	general protection
14	page fault
15	(Intel reserved, do not use)
16	floating-point error
17	alignment check
18	machine check
19–31	(Intel reserved, do not use)
32–255	maskable interrupts

Figure 1. 5 Intel processor event-vector table.

The interrupt mechanism **also** implements a system of interrupt priority levels. These levels **enable** the CPU to defer the handling of low-priority interrupts without masking all interrupts and makes it possible for a high-priority interrupt to preempt the execution of a low-priority interrupt.

Storage Structure

The CPU can load instructions only from memory, so any programs must **first** be loaded into memory to run. General-purpose computers run most of their programs from rewritable memory, called **main memory** (also called random-access memory, or **RAM**).
Computers use other forms of memory as well. (Electrically erasable programmable read-only memory (EEPROM))
A typical instruction–execution cycle, as executed on a system with a **von Neumann architecture**, first fetches an instruction from memory and stores that instruction in the **instruction register**. The instruction is then decoded and may cause operands to be fetched from memory and stored in some internal register. After the instruction on the operands has been executed, the result may be stored back in memory.
Most computer systems provide secondary storage as an extension of main memory. **The main requirement** for secondary storage is that it be able to hold **large** quantities of data **permanently**.
The most common secondary-storage devices are hard-disk drives (**HDDs**) and nonvolatile memory (**NVM**) devices, which provide storage for both **programs** and **data**.
Other possible components include cache memory, CD-ROM or blu-ray, magnetic tapes, and so on. Those that are slow enough and large enough that they are used only for special purposes —to store backup copies of material stored on other devices, for example— are called **tertiary storage**.

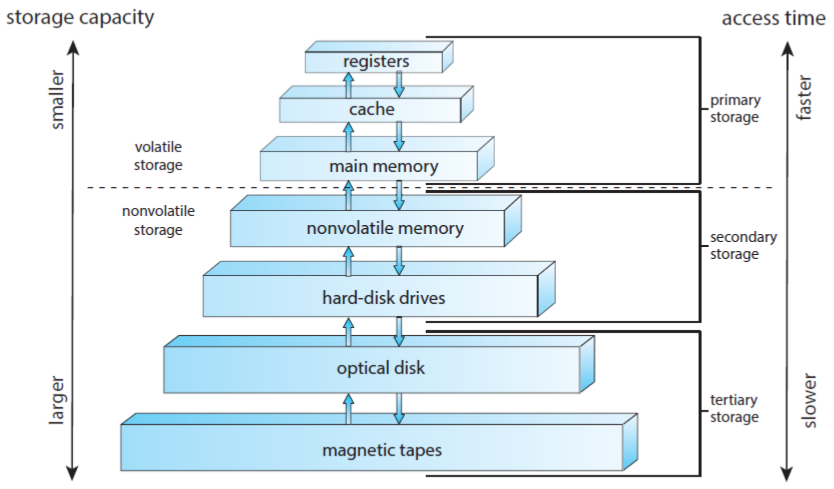


Figure 1. 6 Storage-device hierarchy.

I/O Structure

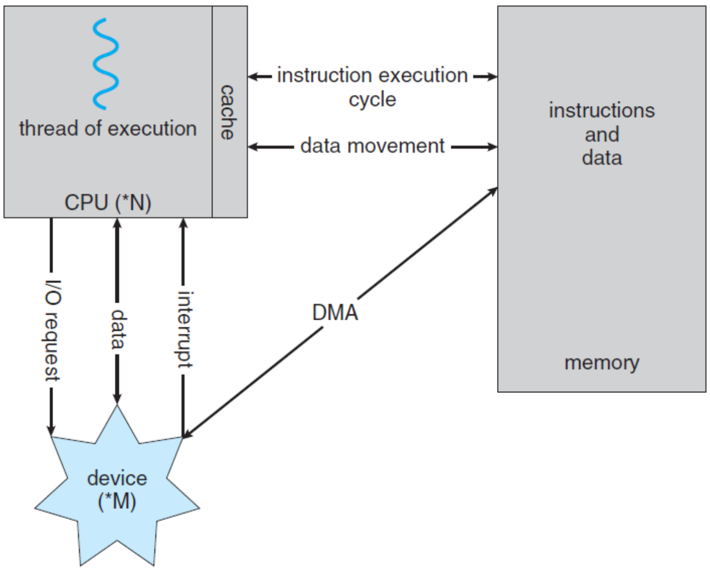


Figure 1. 7 How a modern computer system works.

Direct memory access (DMA)
1.3 Computer-System Architecture
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.
(My computer has a single processor containing one CPU with six processing cores, DELL G7)
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. That is, by increasing the number of processors, we expect to get more work done in less time. The speed-up ratio with N processors is not N, however; it is less than N. When multiple processors cooperate on a task, a certain amount of overhead is incurred in keeping all the parts working **correctly**. This overhead, plus **contention** for shared resources, lowers the expected gain from additional processors.
(multiprocessor systems: two or more central processing units (CPUs) -> more physical chips

than single one)
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.

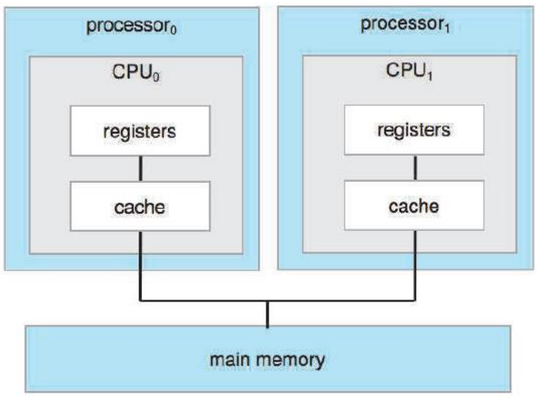


Figure 1. 8 Symmetric multiprocessing architecture.

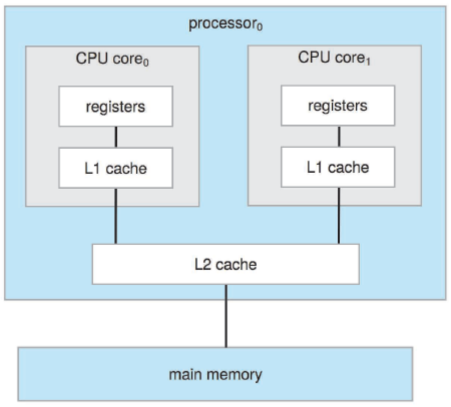


Figure 1. 9 A **dual-core** design with two cores on the **same** chip.

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.

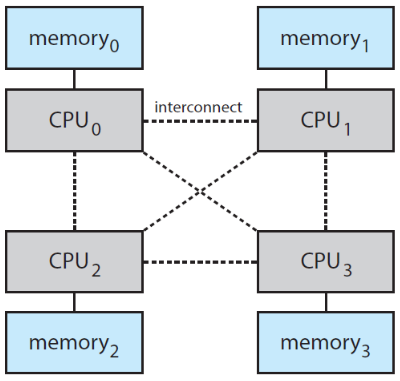


Figure 1. 10 NUMA multiprocessing architecture.

Finally, 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 blade-processor board boots independently and runs its own operating system. Some blade-server boards are multiprocessor as well, which blurs the lines between types of computers. In essence, these servers consist of multiple independent multiprocessor systems.

Clustered Systems

Another type of multiprocessor system is a clustered system, which gathers together multiple CPUs. Clustered systems differ from the multiprocessor systems described in Section 1.3.2 in 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. We should note that the definition of clustered is not concrete; many commercial and opensource packages wrestle to define what a clustered system is and why one form is better than another. The generally accepted definition is that clustered computers **share** storage and are closely **linked** via a local-area network LAN (as described in Chapter 19) 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**.

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. Some systems go beyond graceful degradation and are called fault tolerant, because they can suffer a failure of any single component and still continue operation. Fault tolerance requires a mechanism to allow the failure to be detected, diagnosed, and, if possible, corrected.

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.

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

Figure 1.11 depicts the general structure of a clustered system.

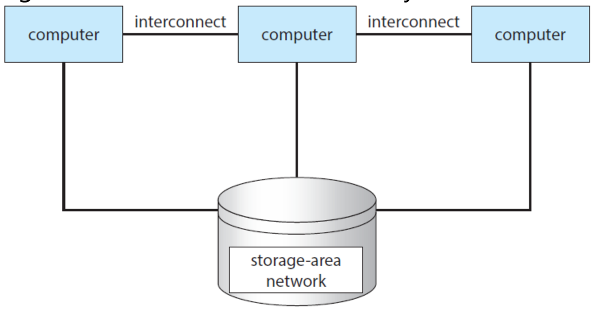


Figure 1. 11 General structure of a clustered system.

1.4 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, keep either the CPU or the I/O devices **busy** at all times. 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 task, such as an I/O operation, to complete. In a **non**-multiprogrammed system, the **CPU** would sit **idle**. In a multiprogrammed system, the operating system **simply switches** to, and executes, another 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.

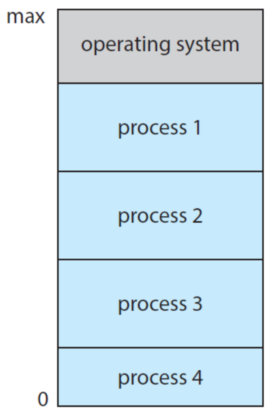


Figure 1. 12 Memory layout for a multiprogramming system.

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.
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**, which is discussed in Chapter 5.
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 (Chapter 10). 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

Since the operating system and its users share the hardware and software resources of the computer system, a properly designed operating system **must** ensure that an incorrect (or malicious) program cannot cause other programs—or the operating system itself—to execute incorrectly. In order to ensure the proper execution of the system, we **must** be able to **distinguish** between the execution of operating-system code and user-defined code. The **approach** taken by most computer systems is to provide hardware support that allow differentiation among various modes of execution.

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

However, when a user application requests a service from the operating system (via a system call), the system **must** transition from user to kernel mode to fulfill the request. This is shown in Figure 1.13. As we shall see, this architectural enhancement is useful for many other aspects of system operation as well.

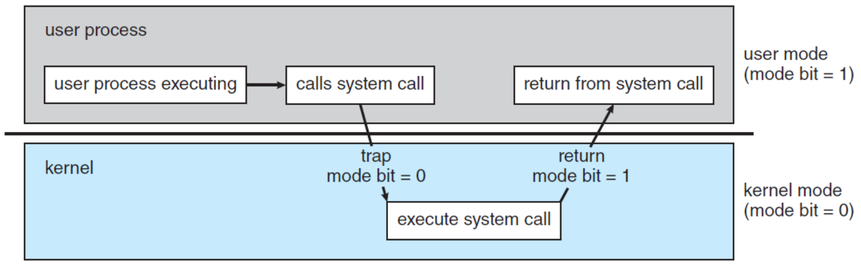


Figure 1. 13 Transition from user to kernel mode.

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.

System calls provide the **means** for a **user program** to **ask** the operating system to perform tasks reserved for the operating system on the user program's behalf.

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.

1.5 Resource Management

As we have seen, an operating system is a resource manager. The system's CPU, memory space, file-storage space, and I/O devices are among the resources that the operating system must manage.

Process Management

A program can do nothing unless its instructions are executed by a CPU. A program in execution, as mentioned, is a process.

We emphasize that a program by itself is not a process. A program is a ***passive*** entity, like the contents of a file stored on disk, whereas a process is an ***active*** entity. A single-threaded process has one program counter specifying the next instruction to execute. (Threads are covered in Chapter 4.) The execution of such a process must be **sequential**. The CPU executes one instruction of the process after another, until the process completes. Further, at any time, **one** instruction **at most** is executed on behalf of **the** process. Thus, although two processes may be associated with the same program, they are nevertheless considered two **separate** execution sequences. A **multithreaded** process has **multiple** program counters, each pointing to the next instruction to execute for a given thread.

Memory Management

For a program to be executed, it **must** be mapped to absolute addresses and **loaded** into memory.

File-System Management

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

Mass-Storage Management

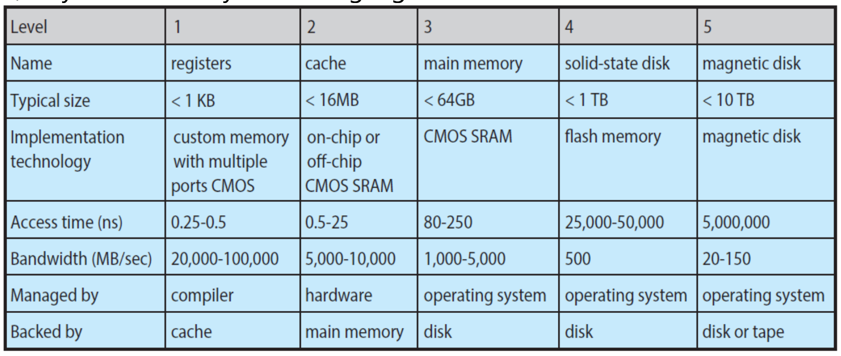
Cache Management

Caching is an important principle of computer systems. Here's how it works. Information is normally kept in some storage system (such as main memory). As it is used, it is copied into a faster storage system—the cache—on a temporary basis. When we need a particular piece of information, we first check whether it is in the cache. If it is, we use the information directly from the cache. If it is not, we use the information from the source, putting a **copy** in the cache under the **assumption** that we will need it again soon.

In addition, **internal** programmable registers provide a high-speed cache for main memory.

Other caches are implemented totally in hardware. For instance, most systems have an instruction cache to hold the instructions expected to be executed next.

Because caches have limited size, cache management is an important design problem. Careful selection of the cache size and of a replacement policy can result in greatly increased performance, as you can see by examining Figure 1.14.



Level	1	2	3	4	5
Name	registers	cache	main memory	solid-state disk	magnetic disk
Typical size	< 1 KB	< 16MB	< 64GB	< 1 TB	< 10 TB
Implementation technology	custom memory with multiple ports CMOS	on-chip or off-chip CMOS SRAM	CMOS SRAM	flash memory	magnetic disk
Access time (ns)	0.25-0.5	0.5-25	80-250	25,000-50,000	5,000,000
Bandwidth (MB/sec)	20,000-100,000	5,000-10,000	1,000-5,000	500	20-150
Managed by	compiler	hardware	operating system	operating system	operating system
Backed by	cache	main memory	disk	disk	disk or tape

Figure 1. 14 Characteristics of various types of storage.

In a hierarchical storage structure, the **same** data **may** appear in **different levels** of the storage system. For example, suppose that an integer A that is to be incremented by 1 is located in file B, and file B resides on hard disk.

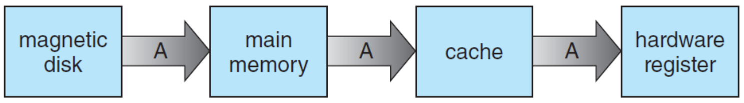


Figure 1. 15 Migration of integer A from disk to register.

Once the increment takes place in the internal register, the value of A **differs** in the various storage systems.

I/O System Management

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

1.6 Security and Protection

If a computer system has multiple users and allows the concurrent execution of multiple processes, then access to data must be regulated.

Protection, then, is any mechanism for controlling the access of processes or users to the resources defined by a computer system. This mechanism must provide means to specify the controls to be imposed and to enforce the controls.

It is the job of security to defend a system from external and internal attacks.

1.7 Virtualization

Virtualization is a technology that allows us to **abstract** the hardware of a **single** computer (the CPU, memory, disk drives, network interface cards, and so forth) into **several different** execution environments, thereby creating the **illusion** that each separate environment is running on its own **private** computer. These environments can be viewed as different individual operating systems (for example, Windows and UNIX) that may be running at the same time and may interact with each other.

Broadly speaking, virtualization software is one member of a class that also includes emulation. Emulation, which involves simulating computer hardware in software, is typically used when the source CPU type is different from the target CPU type.

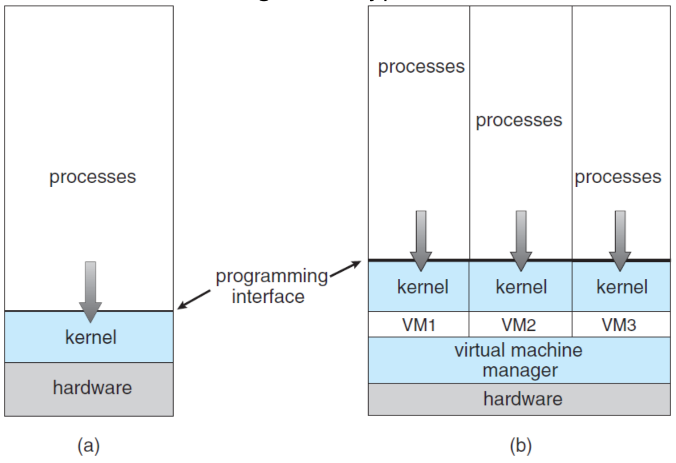


Figure 1. 16 A computer running (a) a single operating system and (b) three virtual machines.

Later, in response to problems with running multiple Microsoft Windows applications on the Intel x86 CPU, VMware created a new virtualization technology in the form of **an application** that ran on Windows. That application ran one or more guest copies of Windows or other native x86 operating systems, each running its own applications. (See Figure 1.16.) Windows was the host operating system, and the VMware application was the virtual machine manager (VMM). The VMM runs the guest operating systems, manages their resource use, and protects each guest from the others.

1.8 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.

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.

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.
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.
1.9 Kernel Data Structures

Lists, Stacks, and Queues

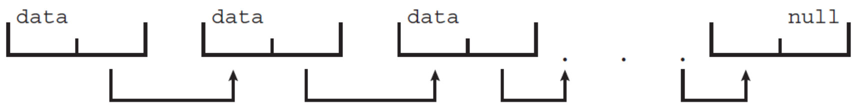


Figure 1. 17 Singly linked list.

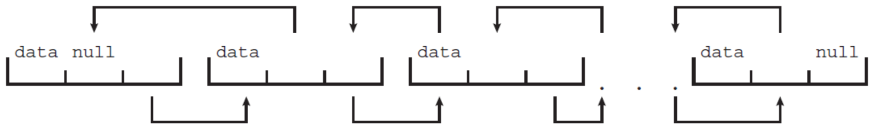


Figure 1. 18 Doubly linked list.

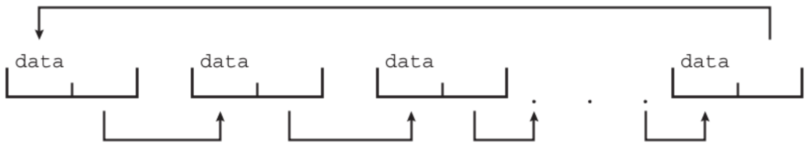


Figure 1. 19 Circularly linked list.

Trees

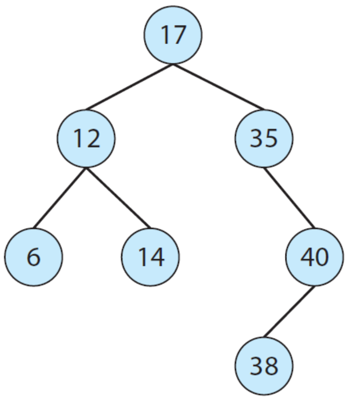


Figure 1. 20 Binary search tree.

Hash Functions and Maps

A hash function takes **data** as its input, performs a numeric operation on the data, and returns a **numeric value**. This numeric value can then be used as an **index** into a **table** (typically an array) to **quickly retrieve** the data.

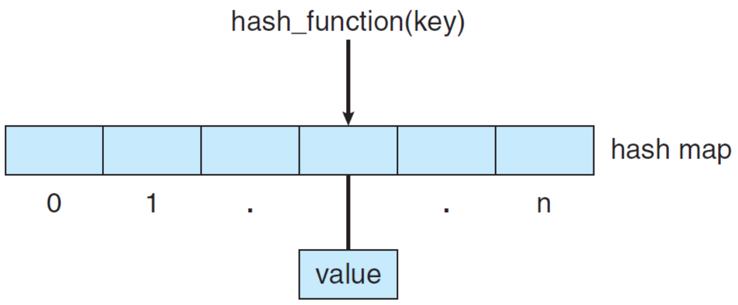


Figure 1. 21 Hash map.

Bitmaps
A bitmap is a string of n binary digits that can be used to represent the status of n items.
1.10 Computing Environments
Traditional Computing

Mobile Computing
Mobile computing refers to computing on handheld smartphones and tablet computers.

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.

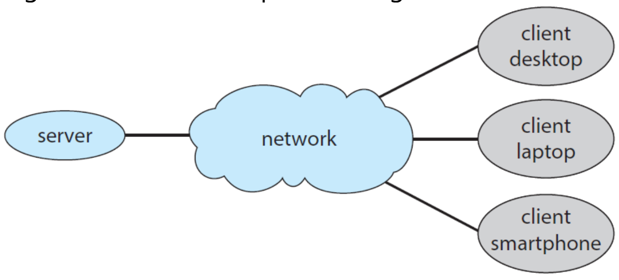


Figure 1. 22 General structure of a client–server system.

Peer-to-Peer Computing

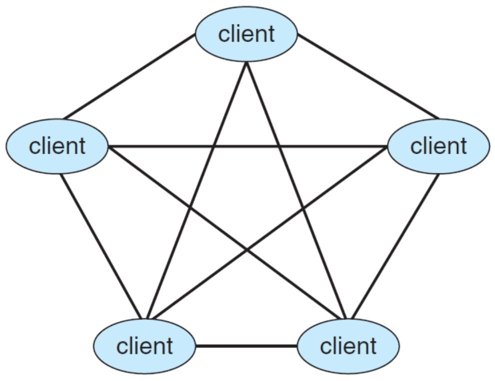


Figure 1. 23 Peer-to-peer system with no centralized service.

Cloud Computing
Cloud computing is a type of computing that delivers computing, storage, and even applications as a service across a network.

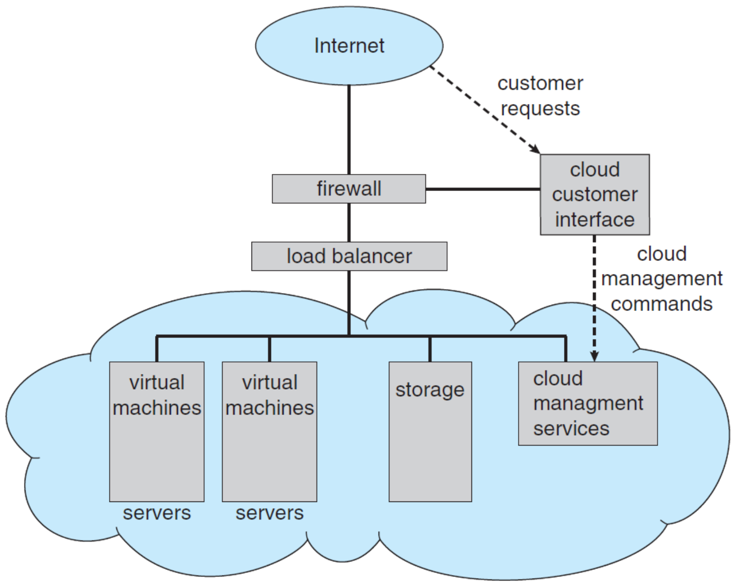


Figure 1. 24 Cloud computing.

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

Practice Exercises
1.1 What are the three **main purposes** of an operating system?
Answer:
The three main purposes are:
• To provide an environment for a computer user to execute programs on computer hardware in a convenient and efficient manner.
• To allocate the separate resources of the computer as needed to solve the problem given. The allocation process should be as **fair** and efficient as possible.
• As a control program it serves two major functions: (1) supervision of the execution of user programs to prevent errors and improper use of the computer, and (2) management of the operation and control of I/O devices.
1.3 What is the **main difficulty** that a programmer must overcome in writing an operating system for a real-time environment?
Answer:
The main difficulty is keeping the operating system within the fixed time constraints of a real-time system. If the system does not complete a task in a certain time frame, it **may** cause a breakdown of the entire system it is running. Therefore, when writing an operating system for a real-time system, the writer **must** be sure that his scheduling schemes don't allow response time to exceed the time constraint.
1.9 **Timers** could be used to compute the current time. Provide a short description of how this could be accomplished.
Answer:
A program could use the following approach to compute the current time using timer interrupts. The program could set a timer for some time in the future and go to sleep. When it is awakened by the interrupt, it could update its local state, which it is using to keep track of the number of interrupts it has received thus far. It could then repeat this process of continually setting timer

interrupts and updating its local state when the interrupts are actually raised.
1.10 Give two reasons why **caches** are useful. What problems do they solve? What problems do they cause? If a cache can be made as large as the device for which it is caching (for instance, a cache as large as a disk), why not make it that large and eliminate the device?
Answer:
Caches are useful when two or more components need to exchange data, **and** the components perform transfers at differing speeds. Caches solve the transfer problem by providing a buffer of intermediate speed between the components. If the fast device finds the data it needs in the cache, it need not wait for the slower device. The data in the cache must be kept **consistent** with the data in the components. If a component has a data value change, and the datum is also in the cache, the cache must also be updated. This is especially a problem on multiprocessor systems where more than one process may be accessing a datum. A component may be eliminated by an equal-sized cache, but only if: (a) the cache and the component have equivalent state-saving capacity (that is, if the component retains its data when electricity is removed, the cache must retain data as well), and (b) the cache is affordable, because faster storage tends to be more expensive.

CHAPTER 2: Operating-System Structures

An operating system provides the environment within which programs are executed.
We can view an operating system from several vantage points. **One** view focuses on the services that the system provides; **another**, on the interface that it makes available to users and programmers; a **third**, on its components and their interconnections.
2.1 Operating-System Services
An operating system provides an environment for the execution of programs.
Figure 2.1 (common classes) 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.

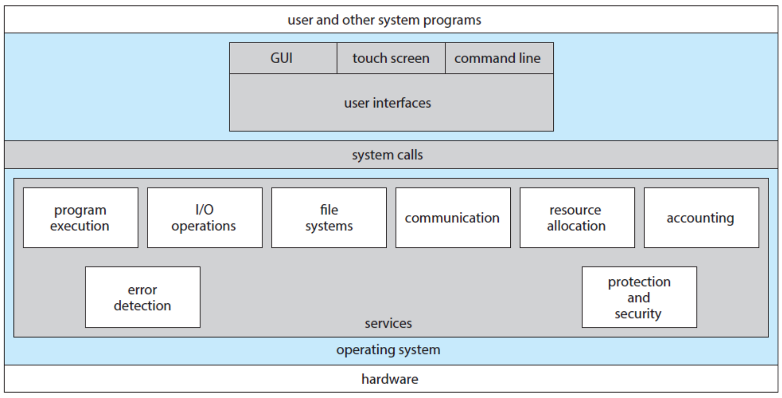


Figure 2. 1 A view of operating system services. (common classes)

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). Most commonly, a graphical user interface (GUI) is used. Mobile systems such as phones and tablets provide a touch-screen interface. Another option is a command-line interface (CLI), which uses text commands and a method for entering them.
• **Program execution**. The system must be able to load a program into memory and to run that program.
• **I/O operations**. 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**.
• **Communications**. There are many circumstances in which one process needs to exchange information with another process. 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.

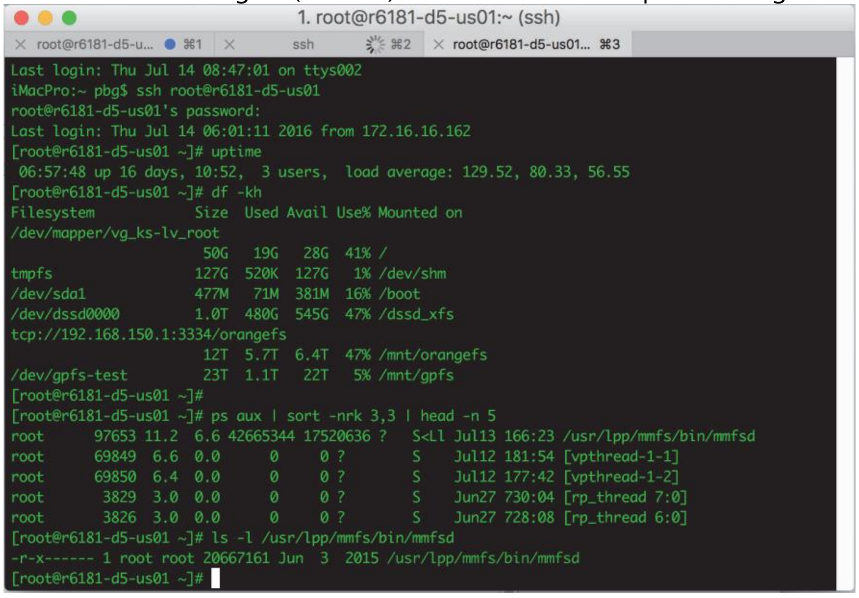
Another set of operating-system functions exists not for helping the user but rather for ensuring the efficient operation of the system itself.
• **Resource allocation**. When there are multiple processes running at the same time, resources must be allocated to each of them.
• **Logging**. We want to keep track of which programs use how much and what kinds of computer resources.
• **Protection and security**. Protection involves ensuring that all access to system resources is controlled. Security of the system from outsiders is also important.
2.2 User and Operating-System Interface
Here, we discuss three fundamental approaches. One provides a command-line interface, or command interpreter, that allows users to **directly** enter commands to be performed **by** the

operating system. The other two allow users to **interface** with the operating system via a graphical user interface, or GUI.

Command Interpreters

Most operating systems, including Linux, UNIX, and Windows, treat the command interpreter as a **special program** that is running when a process is initiated or when a user first logs on (on interactive systems). On systems with multiple command interpreters to choose from, the interpreters are known as shells.

Figure 2.2 shows the Bourne-Again (or bash) shell command interpreter being used on macOS.



```
Last login: Thu Jul 14 08:47:01 on ttys002
iMacPro:~ pbg$ ssh root@r6181-d5-us01
root@r6181-d5-us01's password:
Last login: Thu Jul 14 06:01:11 2016 from 172.16.16.162
[root@r6181-d5-us01 ~]# uptime
 06:57:48 up 16 days, 10:52,  3 users,  load average: 129.52, 80.33, 56.55
[root@r6181-d5-us01 ~]# df -kh
Filesystem            Size  Used Avail Use% Mounted on
/dev/mapper/vg_ks-lv_root
                       50G   19G   28G  41% /
tmpfs                 127G  520K  127G   1% /dev/shm
/dev/sda1             477M   71M  381M  16% /boot
/dev/dssd0000         1.0T  480G  545G  47% /dssd_xfs
tcp://192.168.150.1:3334/orangefs
                       12T  5.7T  6.4T  47% /mnt/orangefs
/dev/gpfs-test         23T  1.1T   22T   5% /mnt/gpfs
[root@r6181-d5-us01 ~]#
[root@r6181-d5-us01 ~]# ps aux | sort -nrk 3,3 | head -n 5
root     97653 11.2  6.6 42665344 17520636 ?   S<Ll Jul13 166:23 /usr/lpp/mmfs/bin/mmfsd
root     69849  6.6  0.0      0     0 ?        S    Jul12 181:54 [vpthread-1-1]
root     69850  6.4  0.0      0     0 ?        S    Jul12 177:42 [vpthread-1-2]
root      3829  3.0  0.0      0     0 ?        S    Jun27 730:04 [rp_thread 7:0]
root      3826  3.0  0.0      0     0 ?        S    Jun27 728:08 [rp_thread 6:0]
[root@r6181-d5-us01 ~]# ls -l /usr/lpp/mmfs/bin/mmfsd
-r-x------ 1 root root 20667161 Jun  3  2015 /usr/lpp/mmfs/bin/mmfsd
[root@r6181-d5-us01 ~]# 
```

Figure 2. 2 The bash shell command interpreter in macOS.

In one approach, the command interpreter **itself** contains the **code** to execute the command. In this case, the number of commands that can be given determines the size of the command interpreter, since each command requires its own implementing code.

An alternative approach—used by **UNIX**, among other operating systems—implements most commands **through** system programs. In this case, the command interpreter does not understand the command in any way; it merely uses the command to **identify a file** to be loaded into **memory** and **executed**.

Thus, the UNIX command to delete a file

```
rm file.txt
```

would search for a file called rm, load the file into memory, and execute it with the **parameter** file.txt. The logic associated with the rm command would be defined completely by the code in the file rm.

Graphical User Interface

A second strategy for interfacing with the operating system is through a user friendly graphical user interface, or GUI. Here, rather than entering commands directly via a command-line interface, users employ a mouse-based window and-menu system characterized by a desktop metaphor. The user moves the mouse to position its pointer on images, or icons, on the screen (the desktop) that represent programs, files, directories, and system functions. Depending on the mouse pointer's location, clicking a button on the mouse can invoke a program, select a file or directory—known as a folder—or pull down a menu that contains commands.

Touch-Screen Interface

Because a either a command-line interface or a mouse-and-keyboard system is impractical for most mobile systems, smartphones and handheld tablet computers **typically** use a touch-screen interface. Here, users interact by making gestures on the touch screen.

Figure 2.3 illustrates the touch screen of the Apple iPhone. Both the iPad and the iPhone use the **Springboard** touch-screen interface.

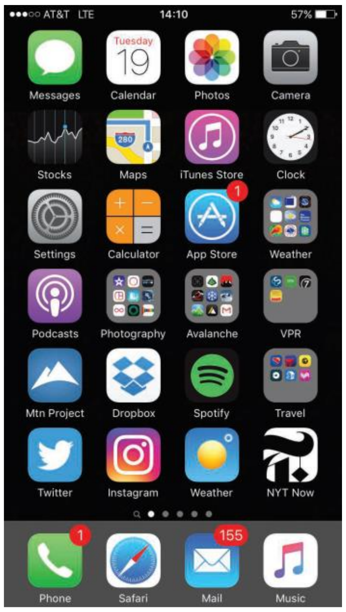


Figure 2. 3 The iPhone touch screen.

Choice of Interface

The choice of whether to use a command-line or GUI interface is mostly one of personal preference. **System administrators** who manage computers and **power users** who have deep knowledge of a system frequently use the command-line interface. For them, it is more efficient, giving them faster access to the activities they need to perform. **Indeed**, on some systems, only a **subset** of system functions is available via the GUI, leaving the less common tasks to those who are command-line knowledgeable. Further, command-line interfaces usually make repetitive tasks easier, in part because they have their own programmability. For example, if a **frequent task** requires **a set of command line steps**, those steps can be **recorded** into a file, and that file can be run just like a program. **The program** is **not** compiled into executable code **but** rather is interpreted by the command-line interface. These **shell scripts** are very common on systems that are command-line oriented, such as **UNIX** and **Linux**.

2.3 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.

For example, the UNIX cp command: (Page 62)

```
cp in.txt out.txt
```

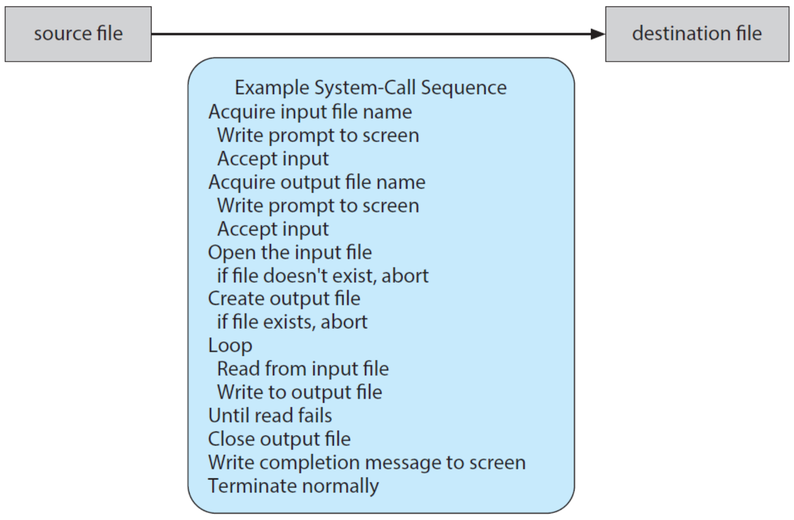


Figure 2. 4 Example of **how** system calls are used.

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.

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.

Behind the scenes, the functions that make up an API typically invoke the actual **system calls** on behalf of the application programmer. (like `man read`)

Why would an application programmer prefer programming according to an API rather than invoking actual system calls? There are several reasons for doing so. One benefit concerns program **portability**. Furthermore, actual system calls can often be **more** detailed and difficult to work with than the API available to an application programmer.

Another important factor in **handling system calls** is the **run-time environment** (**RTE**)—the **full suite** of software needed to execute applications written in **a given** programming language, including its **compilers** or **interpreters** as well as other software, such as **libraries** and **loaders**. The RTE provides a **system-call interface** that serves as the **link** to system calls made available **by the operating system**. The system-call interface **intercepts** function calls in the API and invokes the necessary system calls within the operating system. Typically, **a number** is associated with each system call, and the system-call interface maintains a **table** indexed according to these numbers. The system call interface then invokes the intended system call in the operating-system kernel and returns the status of the system call.

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 as a result of the execution of that system call.

The relationship among an API, the system-call interface, and the operating system is shown in Figure 2.5, which illustrates how the operating system handles a user application invoking the **open()** system call.

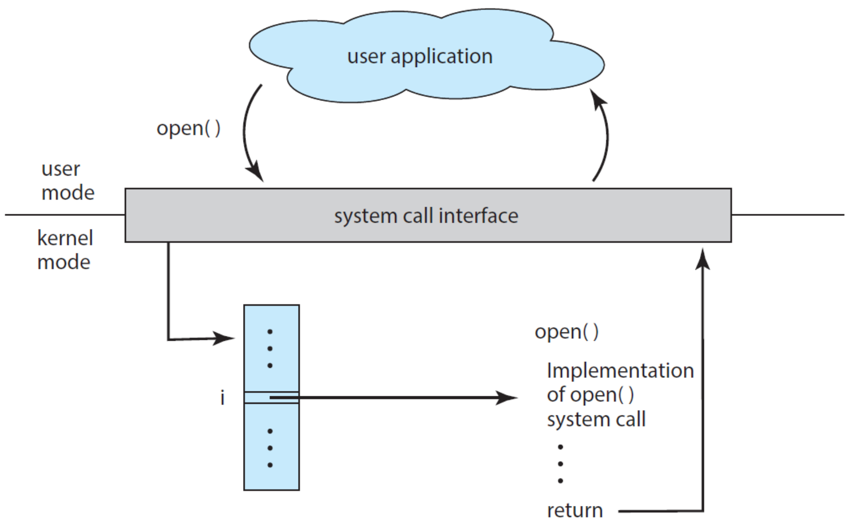


Figure 2. 5 The handling of a user application invoking the open() system call.

Three general methods are used to **pass parameters** to the **o**perating **s**ystem. 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 2.6). 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.

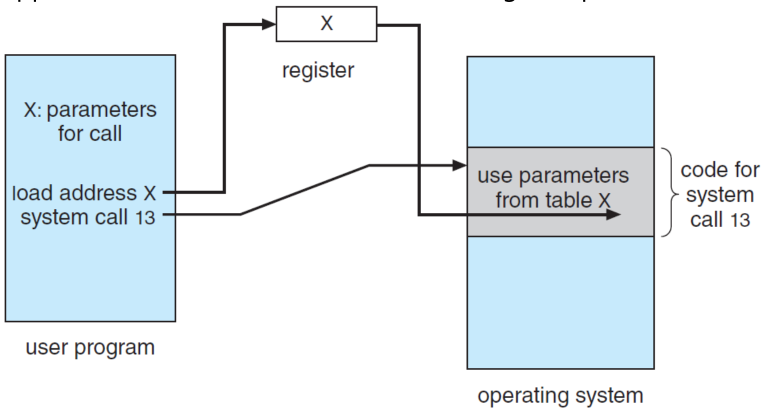
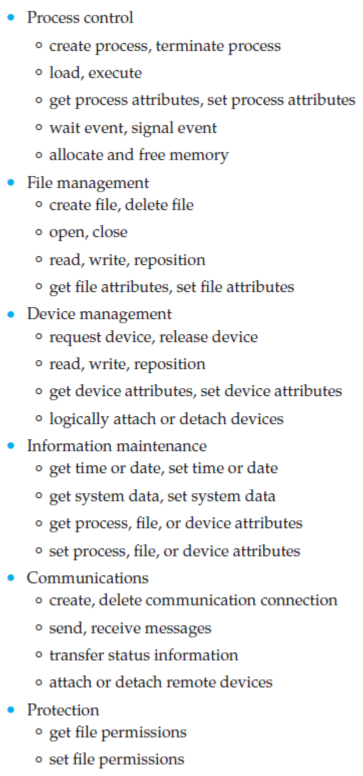


Figure 2. 6 Passing of parameters as a table.

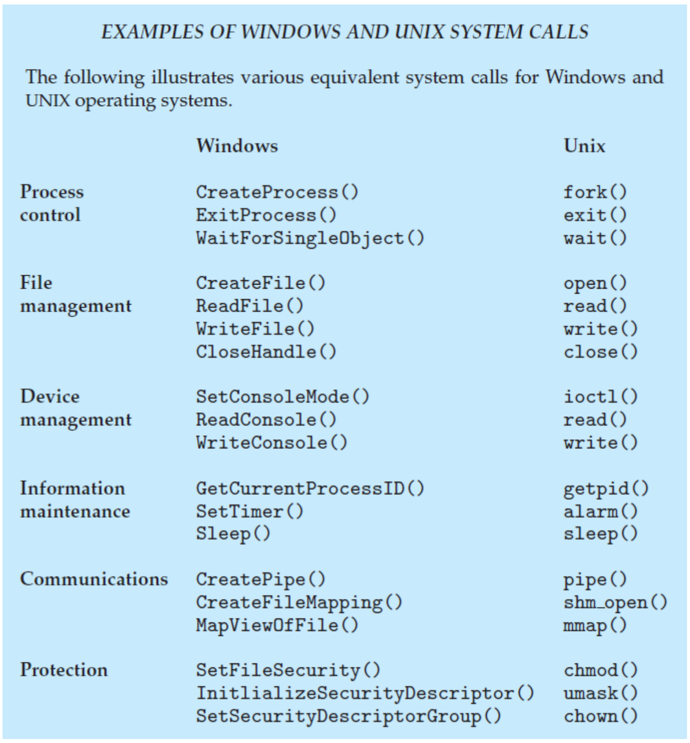
Types of System Calls

System calls can be grouped roughly into six major categories: process control, file management, device management, information maintenance, communications, and protection.
Figure 2.7 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**.



- 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

Figure 2. 7 Types of system calls.



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()`
	`ExitProcess()`
	`WaitForSingleObject()`
		`fork()`
		`exit()`
		`wait()`

File management	`CreateFile()`
	`ReadFile()`
	`WriteFile()`
	`CloseHandle()`
		`open()`
		`read()`
		`write()`
		`close()`

Device management	`SetConsoleMode()`
	`ReadConsole()`
	`WriteConsole()`
		`ioctl()`
		`read()`
		`write()`

Information maintenance	`GetCurrentProcessID()`
	`SetTimer()`
	`Sleep()`
		`getpid()`
		`alarm()`
		`sleep()`

Communications	`CreatePipe()`
	`CreateFileMapping()`
	`MapViewOfFile()`
		`pipe()`
		`shm_open()`
		`mmap()`

Protection	`SetFileSecurity()`
	`InitlializeSecurityDescriptor()`
	`SetSecurityDescriptorGroup()`
		`chmod()`
		`umask()`
		`chown()`

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.

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

FreeBSD (derived from Berkeley UNIX) is an example of a multitasking system. When a user logs on to the system, the shell of the user's choice is run, awaiting commands and running programs the user requests. However, since FreeBSD is a multitasking system, the command interpreter may continue running while another program is executed. To start a new process, the shell executes a **fork()** system call. Then, the selected program is loaded into memory via an **exec()** system call, and the program is executed. (Figure 2.8)
When the process is done, it executes an **exit()** system call to terminate, returning to the invoking process a status code of 0 or a nonzero error code.

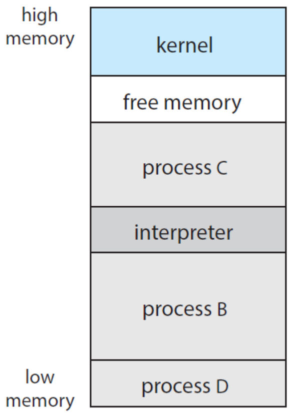


Figure 2. 8 FreeBSD running multiple programs.

File Management
We first need to be able to **create()** and **delete()** files.
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).
Device Management
A system with multiple users may require us to first **request()** a device, to ensure exclusive use of it. After we are finished with the device, we **release()** it. These functions are **similar** to the **open()** and **close()** system calls for files.
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.
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()**.
Another set of system calls is helpful in debugging a program. Many systems provide system calls to **dump()** memory.
Communication
There are two common models of interprocess communication: the message-passing model and the shared-memory model. (Page 72)
Protection

2.4 System Services
System services, also known as system utilities, provide a **convenient environment** for program development and execution. (Page 74)
2.5 Linkers and Loaders
Usually, a program resides on **disk** as a **binary executable file**—for example, a.out or prog.exe. To run on a CPU, the program **must** be brought into memory and **placed** in the context of a process. In this section, we describe the steps in this procedure, from compiling a program to placing it in memory, where it becomes eligible to run on an available CPU core. The steps are highlighted in Figure 2.9.

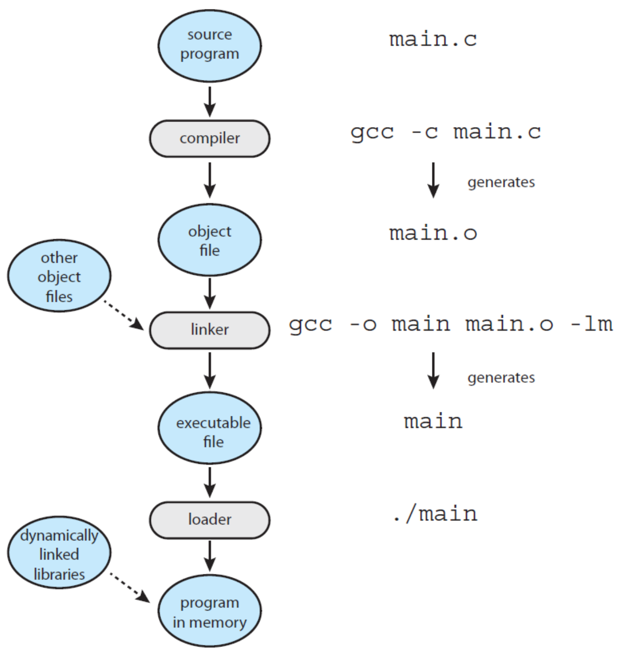


Figure 2. 9 The role of the linker and loader.

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 file. 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.
The process described thus far **assumes** that all libraries are linked into the executable file and loaded into memory. In reality, most systems allow a program to dynamically link libraries as the program is loaded. 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. Instead, the library is conditionally linked and is loaded if it is required during program run time.
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.

2.6 Why Applications Are Operating-System Specific

Fundamentally, applications compiled on one operating system are not executable on other operating systems.
APIs, as mentioned above, specify certain functions at the application level. At the architecture level, an application binary interface (ABI) is used to define how different components of binary code can interface for a given operating system on a given architecture.

2.7 Operating-System Design and Implementation

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

Design Goals

The first problem in designing a system is to define goals and specifications. At the highest level, the design of the system will be affected by the choice of hardware and the type of system: 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**.

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.

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.

2.8 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.

Monolithic Structure

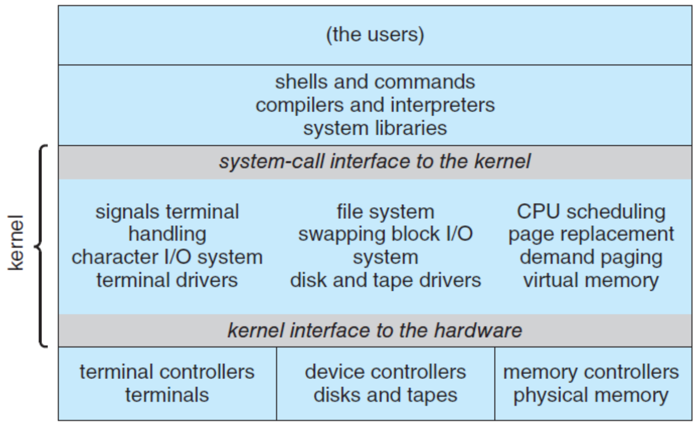


Figure 2. 10 Traditional UNIX system structure.

The **simplest** structure for organizing an operating system is no structure at all. That is, place all of 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.

The Linux operating system is based on UNIX and is structured similarly, as shown in Figure 2.11. Applications typically use the **glibc** standard C library when communicating with the system call interface to the kernel. The Linux kernel is **monolithic** in that it runs entirely in kernel mode in a single address space, but as we shall see in Section 2.8.4, it does have a modular design that allows the kernel to be modified during run time.

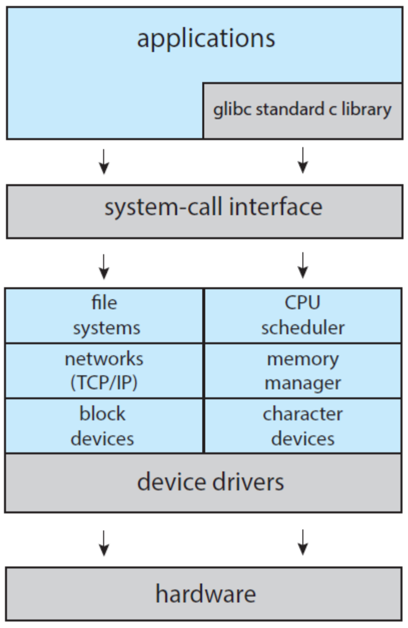


Figure 2. 11 Linux system structure.

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**.

Layered Approach

A system can be made **modular** in many ways. One method is the layered approach, in which the operating system is broken into a number of layers (levels).

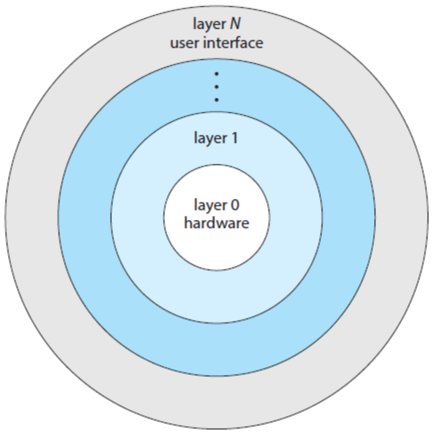


Figure 2. 12 A layered operating system.

The main **advantage** of the layered approach is simplicity of construction and debugging. Layered systems have been successfully used in computer networks (such as TCP/IP) and web applications. Nevertheless, relatively few operating systems use a pure layered approach. One reason involves the challenges of appropriately defining the functionality of each layer. In addition, the overall performance of such systems is **poor** due to the overhead of requiring a user program to traverse through multiple layers to obtain an operating-system service.

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 microkernel approach. This method structures the operating system by **removing** all nonessential components from the kernel and implementing them as user level programs that reside in **separate** address spaces. The result is a smaller kernel.

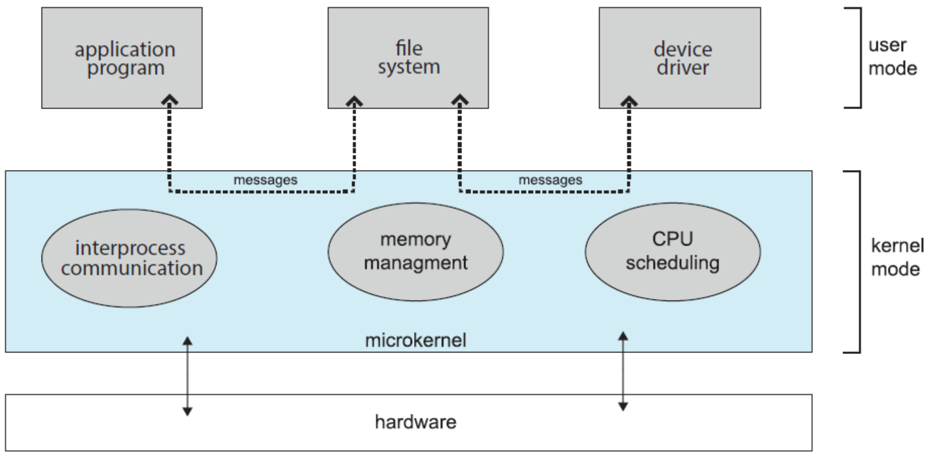


Figure 2. 13 Architecture of a typical microkernel.

One benefit of the microkernel approach is that it makes extending the operating system **easier**. Unfortunately, the performance of microkernels can **suffer** due to increased system-function overhead.

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.

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.

macOS and iOS

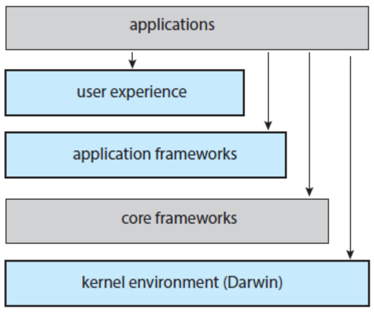


Figure 2. 14 Architecture of Apple's macOS and iOS operating systems.

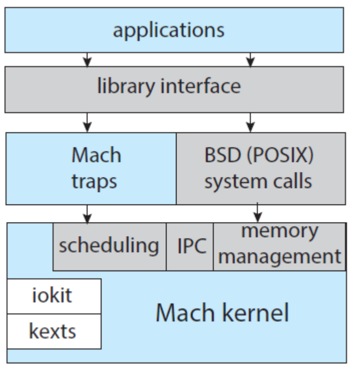


Figure 2. 15 The structure of Darwin.

Android

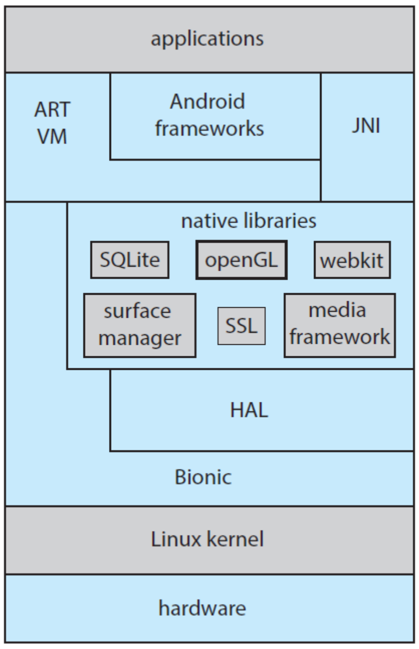


Figure 2. 16 Architecture of Google's Android.

2.9 Building and Booting an Operating System

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

Operating-System Generation

If you are generating (or building) an operating system from scratch, you must follow these steps:

1. Write the operating system source code (or obtain previously written source code).
2. Configure the operating system for the system on which it will run.
3. Compile the operating system.
4. Install the operating system.
5. Boot the computer and its new operating system.

System Boot

After an operating system is generated, it **must** be made available for use by the hardware. **But** how does the hardware know where the kernel is or how to load that kernel? The process of starting a computer by loading the kernel is known as **booting the system**. On most systems, the boot process proceeds as follows:

1. A small piece of code known as the bootstrap program or boot loader **locates** the kernel.
2. The kernel is loaded into memory and started.
3. The kernel **initializes** hardware.
4. The **root** file system is mounted.

2.10 Operating-System Debugging

Broadly, debugging is the activity of finding and fixing errors in a system, both in hardware and in software.

Failure Analysis

(Page 95)

Performance Monitoring and Tuning

(Page 96)

Counters

(Page 96)

Tracing
(Page 97)
BCC
(Page 98)

Practice Exercises
2.1 What is the **purpose** of **system calls**?
Answer:
System calls allow user-level processes to request services of the operating system.
2.3 What system calls have to be executed by a command interpreter or shell in order to start a new process on a UNIX system?
Answer:
In Unix systems, a *fork* system call followed by an *exec* system call need to be performed to start a new process. The *fork* call **clones** the currently executing process, while the *exec* call overlays a new process based on a different executable over the calling process.
2.7 Why do some systems store the operating system in firmware, while others store it on disk?
Answer:
For certain devices, such as handheld PDAs and cellular telephones, a disk with a file system may be **not be available** for the device. In this situation, the operating system must be stored in firmware.

CHAPTER 3: Processes

A process, which is **a program** in **execution**, is the **unit** of work in a modern computing system.
3.1 Process Concept
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:

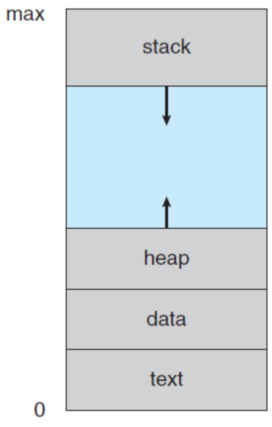


Figure 3. 1 Layout of a process in memory.

- 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)

Notice that the sizes of the text and data sections are **fixed**, as their sizes **do not change** during program run time. However, the stack and heap sections can shrink and grow dynamically during program execution. Each time a function is called, an activation record containing **function parameters**, **local variables**, and the **return address** is pushed onto the stack; when control is returned from the function, the activation record is popped from the stack. Similarly, the heap will grow as memory is dynamically allocated, and will shrink when memory is returned to the system. Although the stack and heap sections grow ***toward*** one another, the operating system **must ensure** they do **not *overlap*** one another.
We **emphasize** that a program by itself is not a process. A program is a ***passive*** entity, such as **a file** containing a list of instructions stored on disk (often called an executable file). In contrast, a process is an ***active*** entity, with a program counter specifying the next instruction to execute and a set of associated resources. A program **becomes** a process **when** an executable file is loaded into memory. **Two common** techniques for loading executable files are double-clicking an icon representing the executable file and entering the name of the executable file on the command line (as in prog.exe or a.out).
Although two processes may be **associated** with the same program, they are **nevertheless** considered **two separate** execution sequences. For instance, several users may be running different copies of the mail program, or the same user may invoke many copies of the web browser program. Each of these is a separate process; and although the text sections are **equivalent**, the data, heap, and stack sections **vary**. It is **also** common to have a process that spawns many processes as it runs.
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.
These names are **arbitrary**, and they **vary** across operating systems. The states that they represent are found on all systems, however. Certain operating systems also more finely delineate process states. It is **important** to realize that **only one process** can be running on **any** processor core at **any instant**. Many processes may be ***ready*** and ***waiting***, however. The state diagram corresponding to these states is presented in Figure 3.2.

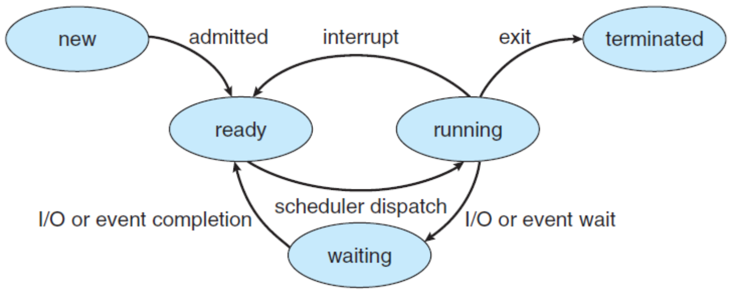


Figure 3. 2 Diagram of process state.

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:

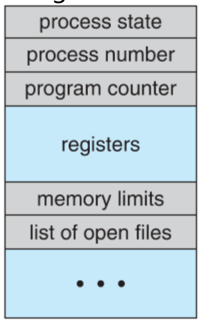


Figure 3. 3 Process control block (PCB).

• **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 (Chapter 9).
• **Accounting information**. This information includes the amount of CPU and real time used, time limits, account numbers, job or process numbers, and so on.
• **I/O status information**. This information includes the list of I/O devices allocated to the

process, a list of open files, and so on.
In brief, the PCB simply serves as the **repository** for all the data needed to start, or restart, a process, along with some accounting data.

Threads
The process model discussed so far **has implied** that a process is a program that performs a single thread of execution. For example, when a process is running a word-processor program, a single thread of instructions is being executed. This **single** thread of control allows the process to perform only **one task** at a time. Thus, the user **cannot** simultaneously type in characters and run the spell checker. Most modern operating systems **have extended** the process concept to allow a process to have multiple threads of execution and thus to perform more than one task at a time. This feature is especially beneficial on **multicore** systems, where multiple threads can run **in parallel**. A multithreaded word processor could, for example, assign one thread to manage user input while another thread runs the spell checker. On systems that support threads, the **PCB** is expanded to include information for each thread. Other **changes** throughout the system are **also** needed to support threads.

3.2 Process Scheduling
The **objective of multiprogramming** is to have some process running at **all times** so as to **maximize CPU utilization**. The **objective of time sharing** is to switch **a CPU** core among processes so frequently that users **can interact** with each program while it is running. (multitask?) To meet these objectives, the **process scheduler** selects an available process (possibly from a set of several available processes) for program execution **on a core**. Each CPU core can run **one process at a time**.
For a system with a **single** CPU core, there will **never be more than** one process running at a time, whereas a multicore system can run multiple processes at one time. If there are more processes than cores, excess processes will have **to wait** until a core is free and can be rescheduled. The **number** of processes **currently** in **memory** is known as the **degree of multiprogramming**.
Balancing the objectives of multiprogramming and time sharing also requires taking the general behavior of a process into account. **In general**, most processes can be described as **either** I/O bound or CPU bound. An **I/O-bound process** is one that spends more of its time doing I/O than it spends doing computations. A **CPU-bound process**, in contrast, generates I/O requests infrequently, using more of its time doing computations. (Also, compute intensive vs memory intensive)

Scheduling Queues
As processes enter the system, they are put into a ready queue, where they are ready and waiting to execute on a CPU's core This queue is generally **stored** as a linked list; a ready-queue header contains pointers to the **first PCB** in the list, and **each** PCB includes a pointer field that points to the next PCB in the ready queue.
The system **also includes other queues**. When a process is allocated a CPU core, it executes for a while and **eventually terminates**, is **interrupted**, or **waits** for the occurrence of a particular event, such as the completion of an I/O request. Suppose the process makes an I/O request to a device such as a disk. Since devices run significantly slower than processors, the process will have to wait for the I/O to become available. Processes that are waiting for a certain event to occur — such as completion of I/O —are placed in a **wait queue** (Figure 3.4).

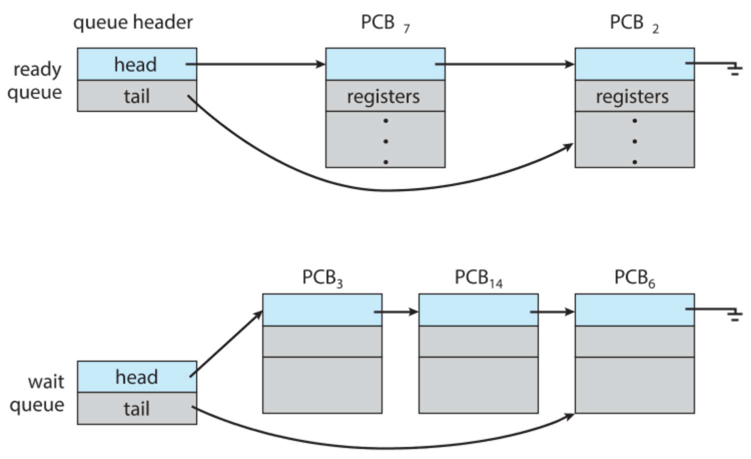


Figure 3. 4 The ready queue and wait queues.

A common representation of **process scheduling** is a **queueing diagram**, such as that in Figure 3.5. **Two types** of queues are present: the ready queue and a **set** of wait queues. The circles represent the resources that serve the queues, and the arrows indicate the flow of processes in the system.

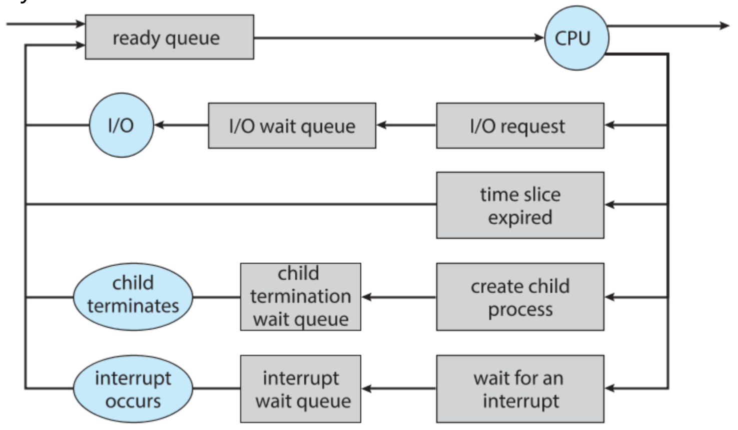


Figure 3. 5 Queueing-diagram representation of process scheduling.

A new process is **initially** put in the ready queue. It waits there until it is selected for execution, or **dispatched**. Once the process is allocated a CPU core and is executing, one of several events could occur:
• The process could issue an I/O request and then be placed in an I/O wait queue.
• The process could create a new child process and then be placed in a wait queue while it awaits the child's termination.
• The process could be removed forcibly from the core, as a result of an interrupt or having its time slice expire, and be put back in the ready queue.
In the first two cases, the process eventually switches from the waiting state to the ready state and is then put back in the ready queue. A process continues this cycle **until it terminates**, at which time it is removed from all queues and has its PCB and resources deallocated.

CPU Scheduling
A process **migrates among** the ready queue and various wait queues throughout its lifetime. **The role** of the **CPU scheduler** is to **select** from among the processes that are in the **ready queue** and **allocate** a CPU core to one of them. The CPU scheduler must select a new process for the CPU frequently. An I/O-bound process may execute for only a few milliseconds before waiting for an I/O request. Although a CPU-bound process will require a CPU core for longer durations, the scheduler is unlikely to grant the core to a process for an extended period. Instead, it is likely designed to forcibly remove the CPU from a process and schedule another process to

run. Therefore, the CPU scheduler executes at least once **every 100 milliseconds, although** typically much more frequently.
Some operating systems have **an intermediate form of scheduling**, known as **swapping**, whose key idea is that sometimes it can be advantageous to remove a process from memory (and from active contention for the CPU) and thus **reduce** the degree of multiprogramming. Later, the process can be reintroduced into memory, and its execution can be continued where it left off. This scheme is known as ***swapping*** because a process can be "swapped out" from memory to disk, where its current status is saved, and later "swapped in" from disk back to memory, where its status is restored. Swapping is typically **only necessary** when memory has been overcommitted and must be freed up. Swapping is discussed in Chapter 9.

Context Switch
As mentioned in Section 1.2.1, **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 restore** 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**.

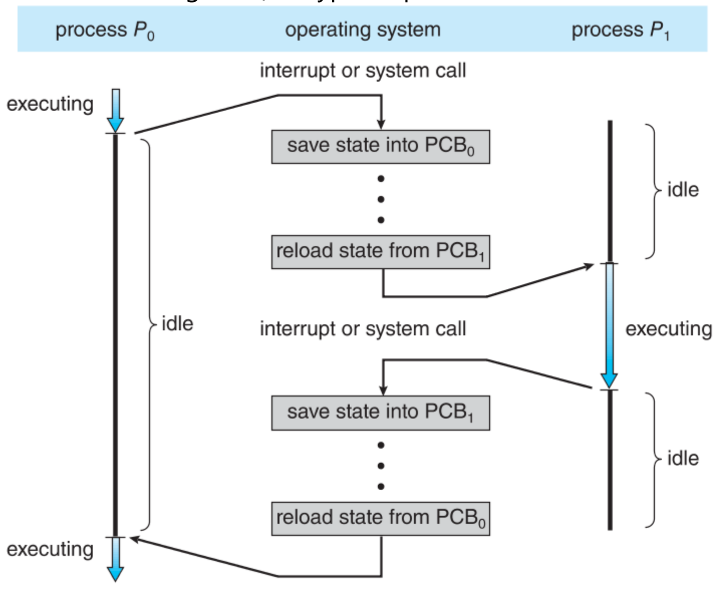


Figure 3. 6 Diagram showing context switch from process to process.

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 memory-management method of the operating system.

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

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***).

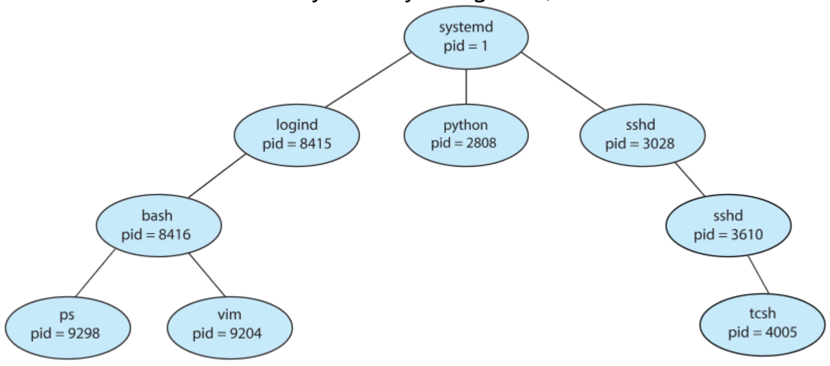


Figure 3. 7 A tree of processes on a typical Linux system.

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

```
ps -el
```

will list complete information for all processes currently active in the system. A process tree similar to the one shown in Figure 3.7 **can be constructed** by recursively tracing parent processes all the way to the systemd process. (In addition, Linux systems provide the pstree command, which displays a tree of all processes in the system.)

In general, when a process creates a child process, that child process will need certain resources (CPU time, memory, files, I/O devices) to accomplish its task. A child process may be able to

obtain its resources **directly** from the operating system, or it may be constrained to a **subset** of the resources of the parent process. The parent may have to partition its resources among its children, or it may be able to share some resources (such as memory or files) among several of its children. Restricting a child process to a subset of the parent's resources prevents any process from overloading the system by creating too many child processes.
In addition to supplying various physical and logical resources, the **parent** process **may pass** along **initialization data (input)** to the **child** process. For example, consider a process whose function is to display the contents of a file—say, hw1.c—on the screen of a terminal. When the process is created, it will get, as an input from its parent process, the name of the file hw1.c. Using that file name, it will open the file and write the contents out. It **may also** get the name of the output device. **Alternatively**, some operating systems pass resources to child processes. 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 of its children have terminated.

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

1. The child process is a **duplicate** of the parent process (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. (Page 118)

```
#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.

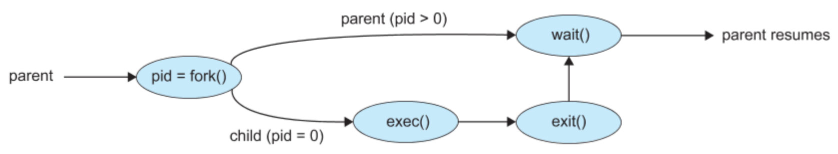
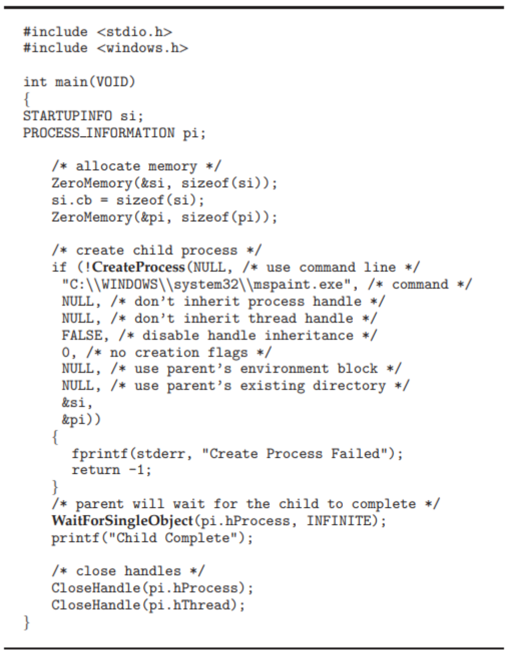


Figure 3. 9 Process creation using the fork() system call.

As an alternative example, we next consider process creation in Windows. (Page 119)



```c
#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.

To illustrate process execution and termination, consider that, in Linux and UNIX systems… (Page 121)

When a process terminates**, its resources are deallocated** by the operating system. **However, its entry** in the process table **must remain there until** the parent calls wait(), because the process table contains the process's exit status. A process that has terminated, but whose parent has not yet called wait(), is known as a **zombie process**. All processes transition to this state when they terminate, **but generally** they exist as zombies only briefly. Once the parent calls

wait(), the process identifier of the zombie process and its entry in the process table are **released**.
Now consider what would happen if a parent **did not invoke** wait() and instead terminated, thereby leaving its child processes as **orphans**. Traditional UNIX systems addressed this scenario by assigning the init process as the new parent to orphan processes. (Recall from Section 3.3.1 that init serves as the root of the process hierarchy in UNIX systems.) The init process **periodically** invokes wait(), thereby allowing the exit status of any orphaned process to be collected and releasing the orphan's process identifier and process-table entry.
Although most Linux systems have replaced init with systemd, the latter process can still serve the same role, although Linux also allows processes other than systemd to inherit orphan processes and manage their termination.

Android Process Hierarchy

Because of resource constraints such as limited memory, mobile operating systems **may have to** terminate existing processes to reclaim limited system resources. Rather than terminating an arbitrary process, Android has identified an importance hierarchy of processes, and when the system must terminate a process to make resources available for a new, or more important, process, it terminates processes in order of increasing importance. From most to least important, the hierarchy of process classifications is as follows:

- **Foreground process** – The current process visible on the screen, representing the application the user is currently interacting with.
- **Visible process** - A process that is not directly visible on the foreground but that is performing an activity that the foreground process is referring to (that is, a process performing an activity whose status is displayed on the foreground process).
- **Service process** - A process that is similar to a background process but is performing an activity that is apparent to the user (such as streaming music).
- **Background process** - A process that may be performing an activity but is not apparent to the user.
- **Empty process** - A process that holds no active components associated with any application.

If system resources must be reclaimed, Android will first terminate empty processes, followed by background processes, and so forth. Processes are assigned an importance ranking, and Android attempts to assign a process as high a ranking as possible. For example, if a process is providing a service and is also visible, it will be assigned the more-important visible classification.

3.4 Interprocess 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, as we discussed in Chapter 2.

MULTIPROCESS ARCHITECTURE-CHROME BROWSER (Page 124)

Cooperating processes require an interprocess 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 interprocess 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.

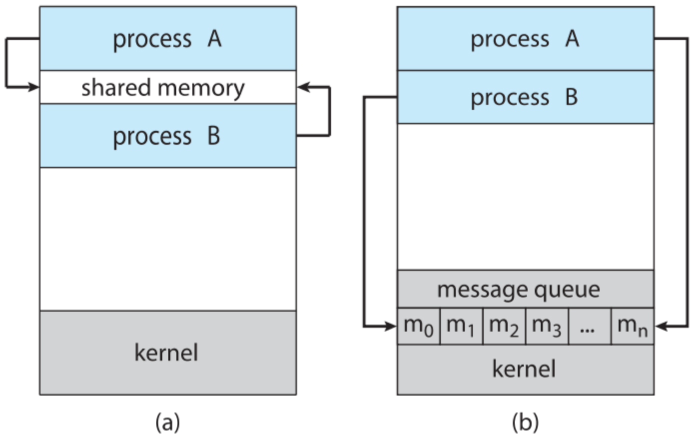


Figure 3. 11 Communications models. (a) Shared memory. (b) Message passing.

Both of the models just mentioned are common in operating systems, and many systems implement both. Message passing is useful for exchanging smaller amounts of data, because no conflicts need be avoided. Message passing is **also easier** to implement in a distributed system than shared memory. (Although there are systems that provide distributed shared memory, we do not consider them in this text.) Shared memory can be faster than message passing, since message-passing systems are **typically** implemented using system calls and thus require the more time-consuming task of kernel intervention. In shared-memory systems, system calls are required only to establish shared memory regions. Once shared memory is established, all accesses are treated as routine memory accesses, and no assistance from the kernel is required.

3.5 IPC in Shared-Memory Systems

Interprocess communication using shared memory requires communicating processes to establish a region of shared memory. Typically, a shared-memory region resides in the address space of the process creating the shared-memory segment. Other processes that wish to communicate using this shared-memory segment must attach it to their address space. 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. (Page 126)

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.

Let's look more closely at how the bounded buffer illustrates interprocess communication using shared memory. The following variables reside in a region of memory shared by the producer and consumer processes:

```
#define BUFFER_SIZE 10

typedef struct {
    . . .
} item;

item buffer[BUFFER_SIZE];
int in = 0;
int out = 0;
```

The shared buffer is implemented as a **circular array** with two logical pointers: *in* and *out*. The variable *in* points to the next free position in the buffer; *out* points to the first full position in the buffer. The buffer is empty when *in* == *out*; the buffer is full when ((*in* +1) % BUFFER_SIZE)== *out*.

The code for the producer process is shown in Figure 3.12, and the code for the consumer process is shown in Figure 3.13. The producer process has a local variable *next_produced* in which the new item to be produced is stored. The consumer process has a local variable *next_consumed* in which the item to be consumed is stored.

This scheme allows at most BUFFER_SIZE-1 items in the buffer at the same time. (write *out+1* instead of *out*?)

```
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.

One issue this illustration does not address concerns the situation in which both the producer process and the consumer process attempt to access the shared buffer concurrently.

3.6 IPC in Message-Passing Systems

In Section 3.5, we showed 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 tradeoff 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
We look at issues related to each of these features next.

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 process and the receiver process must name the other to communicate. A variant of this scheme employs ***asymmetry*** in addressing. Here, only the sender names the recipient; the recipient is not required to name the sender. In this scheme, the *send*() and *receive*() primitives are defined as follows:
• *send*(*P*, message)—Send a message to process *P*.
• *receive*(*id*, message)—Receive a message from any process. The variable *id* is set to the name of the process with which communication has taken place.

The disadvantage in both of these schemes (symmetric and asymmetric) is the **limited modularity** of the resulting process definitions. Changing the identifier of a process may necessitate examining all other process definitions. 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 a number of 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, a number of different links may exist, with each link corresponding to one mailbox.

Now **suppose** that processes P_1, P_2, and P_3 all share mailbox *A*. Process P_1 sends a message to *A*, while both P_2 and P_3 execute a *receive*() from *A*. Which process will receive the message sent by P_1? 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 P_2 or P_3, 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.

A mailbox **may be owned either** by a process or by the operating system. If the mailbox is owned by a process (that is, the mailbox is part of the address space of the process), then we distinguish between the owner (which can only **receive** messages through this mailbox) and the user (which can only **send** messages to the mailbox). Since each mailbox has a **unique** owner, there can be no confusion about which process should receive a message sent to this mailbox. When a process that owns a mailbox terminates, the mailbox **disappears**. Any process that subsequently sends a message to this mailbox must be notified that the mailbox no longer exists.
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 particular 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.

The process that creates a new mailbox is that mailbox's owner **by default**. Initially, the owner is the only process that can receive messages through this mailbox. However, the ownership and receiving privilege may be passed to other processes through appropriate system calls. Of course, this provision could result in multiple receivers for each 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 behavior in relation to various operating-system algorithms.)
• **Blocking send**. The sending process is blocked until the message is received by the receiving process or by the mailbox.
• **Nonblocking send**. The sending process sends the message and resumes operation.
• **Blocking receive**. The receiver blocks until a message is available.
• **Nonblocking receive**. The receiver retrieves either a valid message or a null.

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.

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**.

3.7 Examples of IPC Systems
POSIX Shared Memory

Mach Message Passing

Windows
The message-passing facility in Windows is called the advanced local procedure call (ALPC) facility. (Page 138)

Pipes
A **pipe** acts as a conduit allowing two processes to communicate. Pipes were one of the first IPC mechanisms in early UNIX systems. They typically provide one of the simpler ways for processes to communicate with one another, although they also have some limitations. In implementing a pipe, four issues must be considered:
• Does the pipe allow **bidirectional** communication, or is communication **unidirectional**?
• If two-way communication is allowed, is it **half duplex** (data can travel only one way at a time) or **full duplex** (data can travel in both directions at the same time)?
• Must a **relationship** (such as parent–child) exist between the communicating processes?
• Can the pipes communicate over a network, or must the communicating processes reside on

the **same** machine?
(Page 139)
Ordinary Pipes (Page 140)

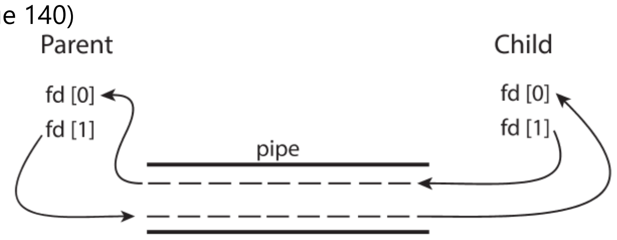
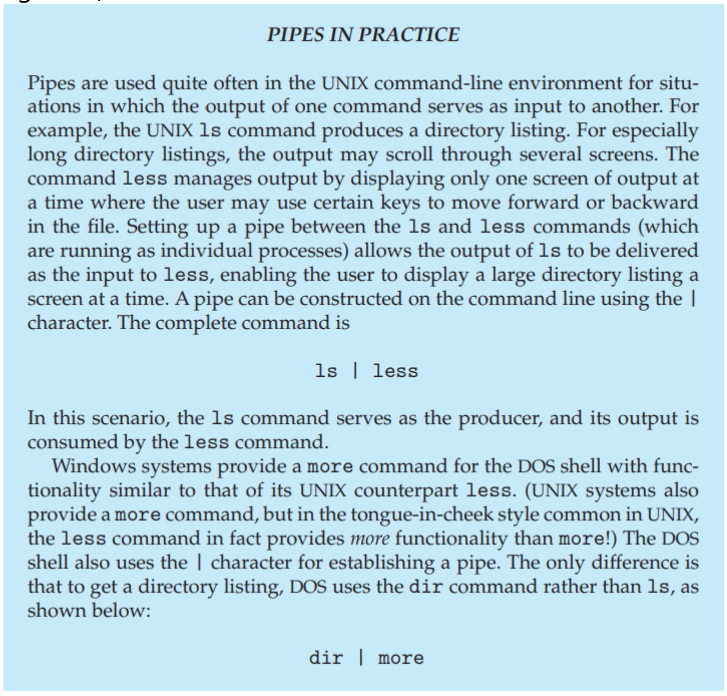


Figure 3. 14 File descriptors for an ordinary pipe.

Named Pipes (Page 143)



PIPES IN PRACTICE

Pipes are used quite often in the UNIX command-line environment for situations in which the output of one command serves as input to another. For example, the UNIX `ls` command produces a directory listing. For especially long directory listings, the output may scroll through several screens. The command `less` manages output by displaying only one screen of output at a time where the user may use certain keys to move forward or backward in the file. Setting up a pipe between the `ls` and `less` commands (which are running as individual processes) allows the output of `ls` to be delivered as the input to `less`, enabling the user to display a large directory listing a screen at a time. A pipe can be constructed on the command line using the | character. The complete command is

```
ls | less
```

In this scenario, the `ls` command serves as the producer, and its output is consumed by the `less` command.

Windows systems provide a `more` command for the DOS shell with functionality similar to that of its UNIX counterpart `less`. (UNIX systems also provide a `more` command, but in the tongue-in-cheek style common in UNIX, the `less` command in fact provides *more* functionality than `more`!) The DOS shell also uses the | character for establishing a pipe. The only difference is that to get a directory listing, DOS uses the `dir` command rather than `ls`, as shown below:

```
dir | more
```

3.8 Communication in Client–Server Systems
In Section 3.4, we described how processes can communicate using shared memory and message passing. These techniques can be used for communication in client-server systems (Section1.10.3) **as well**. In this section, we explore two other strategies for communication in client–server systems: sockets and remote procedure calls (RPCs). As we shall see in our coverage of RPCs, not only are they useful for client-server computing, but Android also uses remote procedures as a form of IPC between processes running on the same system.

Sockets
A socket is defined as an endpoint for communication. A pair of processes communicating over a network employs **a pair of** sockets-one for each process. A socket is identified by an **IP address** concatenated with **a port number**. In general, sockets use a client-server architecture. The server waits for incoming client requests by listening to a specified port. Once a request is received, the server accepts a connection from the client socket to complete the connection. Servers implementing specific services (such as SSH, FTP, and HTTP) listen to ***well-known*** ports (an SSH server listens to port 22; an FTP server listens to port 21; and a web, or HTTP, server listens to port 80). All ports below 1024 are considered well known and are used to implement standard services.
When a client process initiates a request for a connection, it is assigned a port by its host computer. This port has some arbitrary number greater than 1024. For example, if a client on

host X with IP address 146.86.5.20 wishes to establish a connection with a web server (which is listening on port 80) at address 161.25.19.8, host X may be assigned port 1625. The connection will consist of a pair of sockets: (146.86.5.20:1625) on host X and (161.25.19.8:80) on the web server. This situation is illustrated in Figure 3.26. The packets traveling between the hosts are delivered to the appropriate process based on the destination port number.

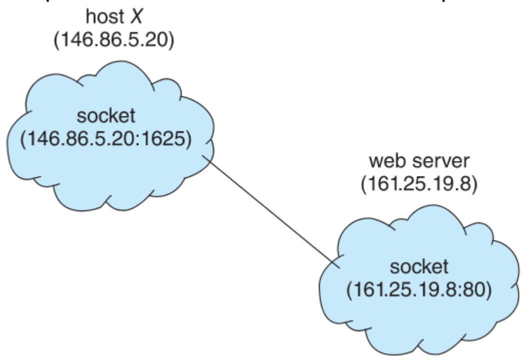


Figure 3. 15 Communication using sockets.

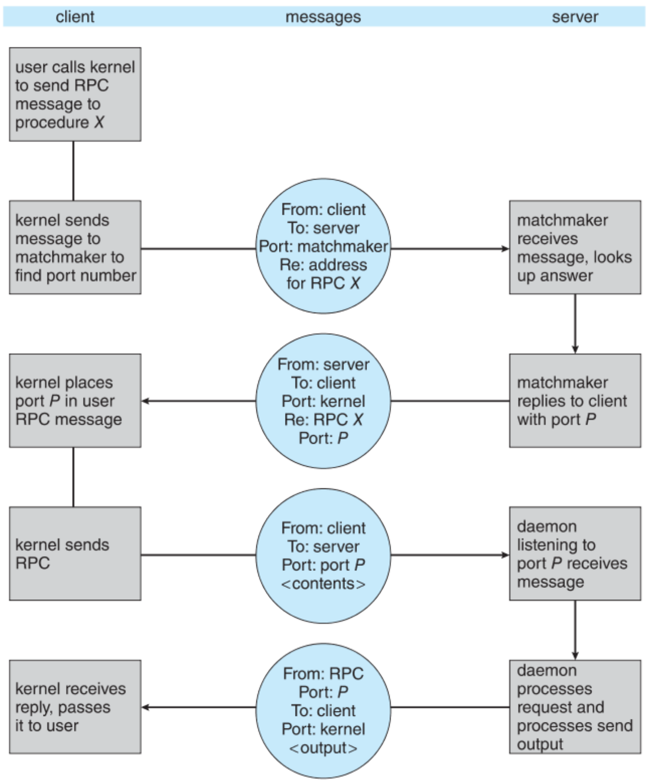
All connections must be unique. Therefore, if **another** process also on host X wished to establish another connection with the same web server, it would be assigned a port number greater than 1024 and not equal to 1625. This ensures that all connections consist of a unique pair of sockets.

Although most program examples in this text use C, we will illustrate sockets using Java, as it provides a much easier interface to sockets and has a rich library for networking utilities. (Page 147)

Communication using sockets-although common and efficient-is considered a **low-level** form of communication between distributed processes. One reason is that sockets allow only an **unstructured** stream of bytes to be exchanged between the communicating threads. It is the responsibility of the client or server application to impose a structure on the data. In the next subsection, we look a higher-level method of communication: remote procedure calls (RPCs).

Remote Procedure Calls

One of the most common forms of remote service is the RPC paradigm, which was designed as a way to abstract the procedure-call mechanism for use between systems with network connections. It is similar in many respects to the IPC mechanism described in Section 3.4, and it is usually built on top of such a system. Here, however, because we are dealing with an environment in which the processes are executing on separate systems, we must use a message-based communication scheme to provide remote service. (Page 149)



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

Figure 3. 16 Execution of a remote procedure call (RPC).

Practice Exercises

3.6 Consider the "**exactly once**" semantic with respect to the RPC mechanism. Does the algorithm for implementing this semantic execute correctly even if the ACK message sent back to the client is lost due to a network problem? Describe the sequence of messages, and discuss whether "exactly once" is still preserved.

Answer:

The "exactly once" semantics ensure that a remote procedure will be executed exactly once and only once. The general algorithm for ensuring this combines an acknowledgment (ACK) scheme combined with timestamps (or some other incremental counter that allows the server to distinguish between duplicate messages). The **general strategy** is for the client to send the RPC to the server along with a timestamp. The client will also start a **timeout** clock. The client will then wait for one of two occurrences: (1) it will receive an ACK from the server indicating that the remote procedure was performed, or (2) it will time out. If the client times out, it assumes the server was unable to perform the remote procedure, so the client invokes the RPC a **second time**, sending a later timestamp. The client may not receive the ACK for one of two reasons: (1) the original RPC was never received by the server, or (2) the RPC was correctly received-and performed-by the server but the ACK was lost. In situation (1), the use of ACKs allows the server ultimately to receive and perform the RPC. In situation (2), the server will receive a duplicate RPC, and it will use the timestamp to identify it as a duplicate so as not to perform the RPC a second time. It is **important** to note that the server must send a **second** ACK back to the client to inform the client the RPC has been performed.

3.7 Assume that a distributed system is susceptible to server failure. What mechanisms would be required to guarantee the "exactly once" semantic for execution of RPCs?

Answer:

The server should keep track in **stable storage** (such as a disk log) of information regarding what RPC operations were received, whether they were successfully performed, and the results associated with the operations. When a server crash takes place and an RPC message is received, the server can check whether the RPC has been previously performed and therefore guarantee “exactly once” semantics for the execution of RPCs.

CHAPTER 4: Threads & Concurrency

The process model introduced in Chapter 3 assumed that a process was an executing program with a single thread of control. Virtually all modern operating systems, however, provide features enabling a process to contain multiple threads of control. Identifying opportunities for parallelism through the use of threads is becoming increasingly important for modern **multicore** systems that provide **multiple** CPUs.

4.1 Overview

A thread is a **basic unit** of CPU utilization; it comprises a thread ID, a program counter (PC), a register set, and a stack. It shares with other threads belonging to the **same process** its code section, data section, and other operating-system resources, such as open files and signals. A traditional process has a single thread of control. If a process has multiple threads of control, it can perform more than one task at a time. Figure 4.1 illustrates the difference between a traditional single-threaded process and a multithreaded process.

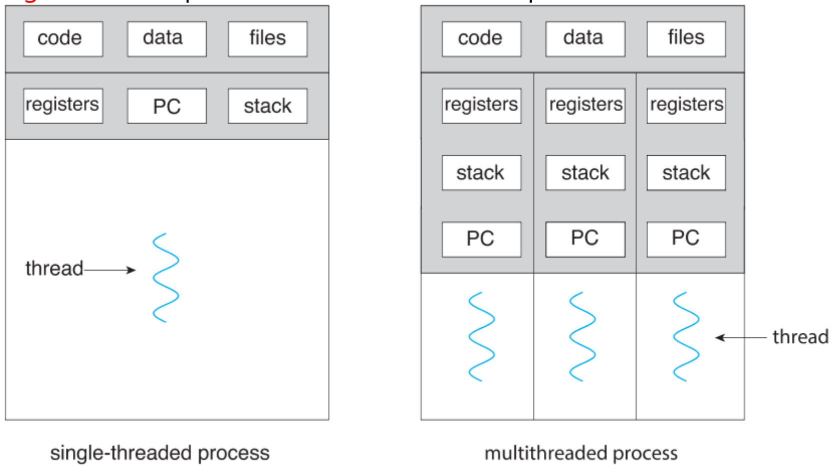


Figure 4. 1 Single-threaded and multithreaded processes.

Motivation

Most software applications that run on modern computers and mobile devices are multithreaded. An application typically is implemented as a separate process with several threads of control. Below we highlight a few examples of multithreaded applications:

• An application that creates photo thumbnails from a collection of images may use a separate thread to generate a thumbnail from each separate image.

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

• A word processor may have a thread for displaying graphics, another thread for responding to keystrokes from the user, and a third thread for performing spelling and grammar checking in the background.

In certain situations, a single application may be required to perform several similar tasks. For example, a web server accepts client requests for web pages, images, sound, and so forth. A busy web server may have several (perhaps thousands of) clients concurrently accessing it. If the web server ran as a traditional single-threaded process, it would be able to service only one client at a time, and a client might have to wait a very long time for its request to be serviced.
One solution is to have the server run as a single process that accepts requests. When the server receives a request, it creates a separate process to service that request. In fact, this process-creation method was in common use before threads became popular. Process creation is **time consuming and resource intensive**, however. If the new process will perform the same tasks as the existing process, why incur all that overhead? It is generally **more efficient** to use one process that contains multiple threads. If the web-server process is multithreaded, the server

will create a separate thread that listens for client requests. When a request is made, rather than creating another process, the server creates a new thread to service the request and resumes listening for additional requests. This is illustrated in Figure 4.2.

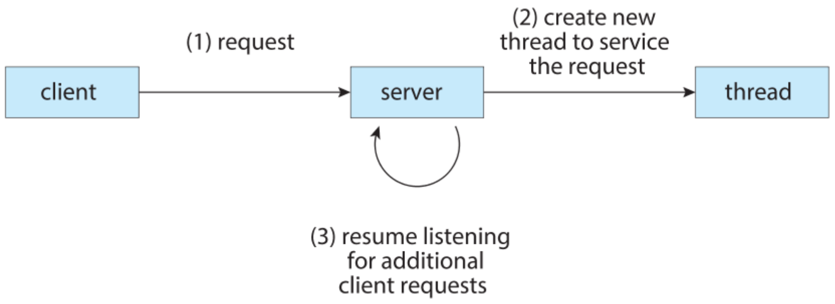

(1) request
(2) create new thread to service the request
client
server
thread
(3) resume listening for additional client requests

Figure 4. 2 Multithreaded server architecture.

Benefits
The benefits of multithreaded programming can be broken down into four major categories:
1. **Responsiveness**.
2. **Resource sharing**.
3. **Economy**.
4. **Scalability**.

4.2 Multicore Programming
Earlier in the history of computer design, in response to the need for more computing performance, single-CPU systems evolved into multi-CPU systems. A later, yet similar, trend in system design is to place multiple computing cores on a single processing chip where each core appears as a separate CPU to the operating system (Section 1.3.2). We refer to such systems as **multicore**, and multithreaded programming provides a mechanism for more efficient use of these multiple computing cores and improved concurrency. Consider an application with four threads. On a system with a single computing core, concurrency merely means that the execution of the threads will be **interleaved** over time (Figure 4.3), because the processing core is capable of executing **only one** thread at a time. On a system with multiple cores, however, **concurrency** means that some threads can run in parallel, because the system can assign a separate thread to each core (Figure 4.4).

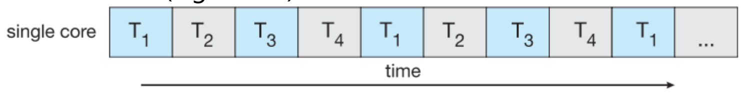


Figure 4. 3 Concurrent execution on a single-core system.

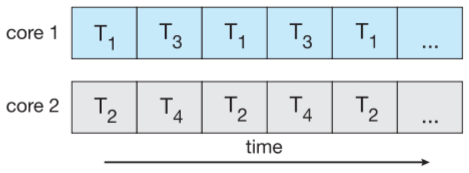

core 1
T_1 T_3 T_1 T_3 T_1 ...
core 2
T_2 T_4 T_2 T_4 T_2 ...
time

Figure 4. 4 Parallel execution on a multicore system.

Notice the distinction between concurrency and parallelism in this discussion. A concurrent system supports more than one task by allowing all the tasks to make progress. In contrast, a parallel system can perform more than one task simultaneously. Thus, it is possible to have concurrency without parallelism. Before the advent of multiprocessor and multicore architectures, most computer systems had only a single processor, and CPU schedulers were designed to provide the illusion of parallelism by rapidly switching between processes, thereby allowing each process to make progress. Such processes were running concurrently, but not in parallel.