Operating systems

What are the different types?

Linux is a Unix-like computer operating system assembled under the model of free and open source software

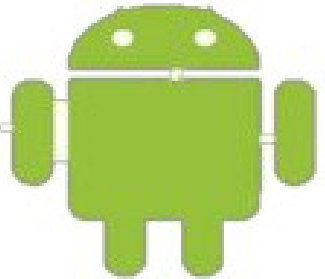
development and distribution.

Microsoft Windows is a series of graphical interface operating systems developed, marketed, and sold by Microsoft.

Mac OS is a series of graphical user interface-based operating systems developed by Apple Inc. for their Macintosh

iOS (previously **iPhone OS**) is a mobile operating system developed and distributed by Apple Inc. Originally unveiled in 2007 for the iPhone, it has been extended to support other Apple devices such as

the iPod Touch



Android is a Linux-based operating system designed primarily for touchscreen mobile devices such as smartphones and tablet computers. Initially developed by Android, Inc.

BSD/OS had a reputation for reliability in server roles; the renowned Unix programmer and author W. Richard Stevens used it for his own personal web server for this reason.

- A computer system can be divided roughly into four components: **the hardware, the operating system, the application programs, and the users**.
- The hardware—the central processing unit (CPU), the memory, and the input/output (I/O) devices—provides the basic computing resources for the system.
- The application programs—such as word processors, spreadsheets, compilers, and Web browsers—define the ways in which these resources are used to solve users' computing problems.
- The operating system controls the hardware and coordinates its use among the various application programs for the various users.
- Like a government, it performs no useful function by itself. It simply provides an environment within which other programs can do useful work

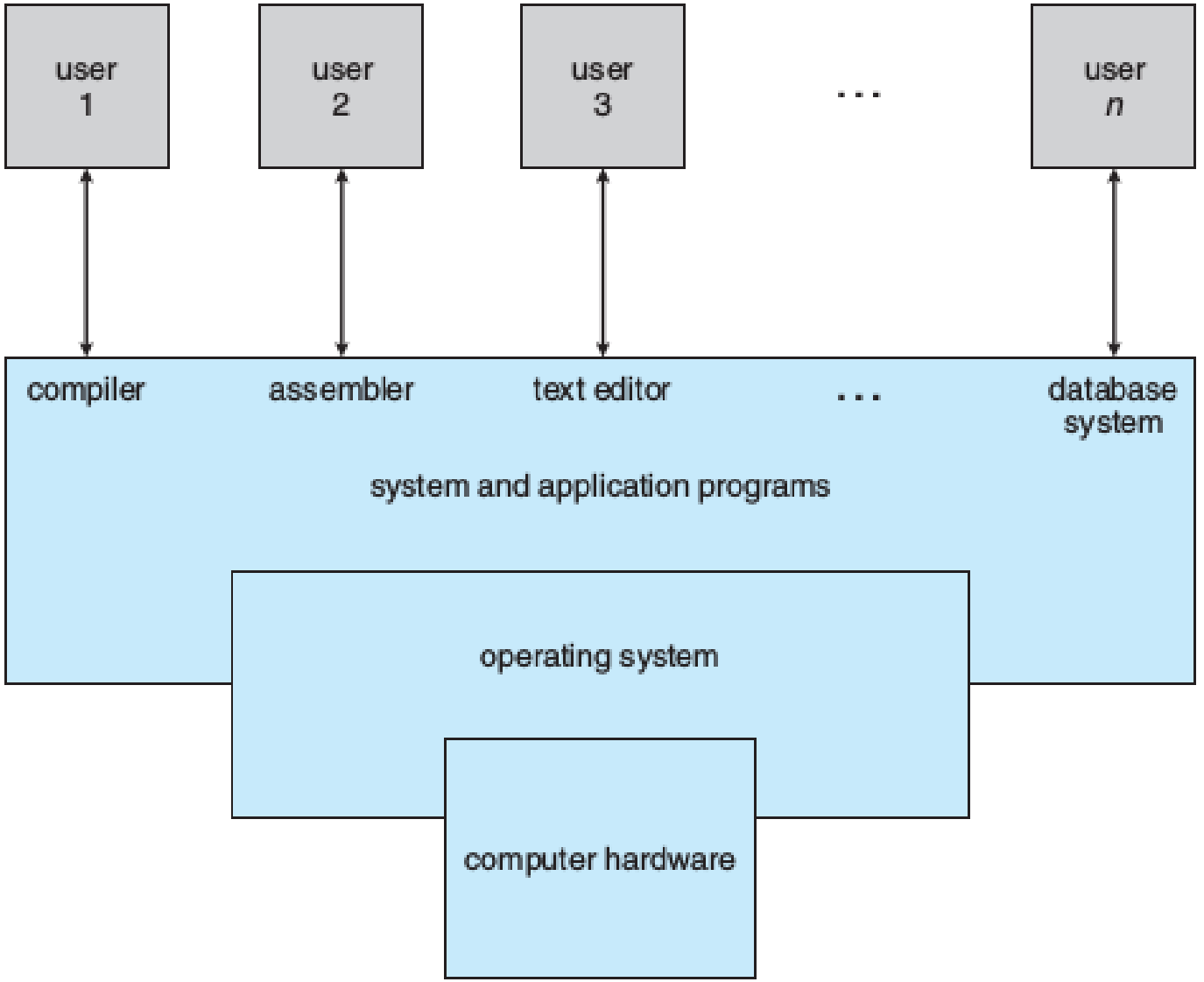


Figure 1.1 Abstract view of the components of a computer system.

User View

1. PC

- Most computer users sit in front of a PC , consisting of a monitor, keyboard, mouse, and system unit. Such a system is designed for one user to monopolize its resources.
- The goal is to maximize the work (or play) that the user is performing.
- In this case, the operating system is designed mostly for ease of use, with some attention paid to performance and none paid to resource utilization —how various hardware and software resources are shared.
- Performance is, of course, important to the user; but such systems are optimized for the single-user experience rather than the requirements of multiple users.

2. Mainframe

- In other cases, a user sits at a terminal connected to a mainframe or a minicomputer. Other users are accessing the same computer through other terminals.
- These users share resources and may exchange information.
- The operating system in such cases is designed to maximize resource utilization— to assure that all available CPU time, memory, and I/O are used efficiently and that no individual user takes more than her fair share.

System View

- We can view an operating system as a **resource allocator**. A computer system has many resources that may be required to solve a problem: CPU time, memory space, file-storage space, I/O devices, and so on.
- The operating system acts as the manager of these resources. Facing numerous and possibly conflicting requests for resources, the operating system must decide how to allocate them to specific programs and users so that it can operate the computer system efficiently and fairly.
- An operating system is a **control program**. A control program manages the execution of user programs to prevent errors and improper use of the computer. It is especially concerned with the operation and control of I/O devices.
- The operating system is the one program running at all times on the computer—usually called the **kernel**.

Computer System Architecture

1. Single Processor Systems
2. Multiprocessor Systems
3. Clustered Systems

1. Single Processor Systems

- On a single- processor system, there is one main CPU capable of executing a general-purpose instruction set, including instructions from user processes.
- Almost all single-processor systems have other special-purpose processors as well. They may come in the form of device-specific processors, such as disk, keyboard, and graphics controllers; or, on mainframes, they may come in the form of more general-purpose processors, such as I/O processors that move data rapidly among the components of the system.
- All of these special-purpose processors run a limited instruction set and do not run user processes.
- For example, a disk-controller microprocessor receives a sequence of requests from the main CPU and implements its own disk queue and scheduling algorithm. This arrangement relieves the main CPU of the overhead of disk scheduling.

2. Multiprocessor Systems

- Multiprocessor systems (also known as parallel systems or multicore systems) have two or more processors in close communication, sharing the computer bus and sometimes the clock, memory, and peripheral devices.
- Multiprocessor systems have three main advantages:

  a. **Increased throughput**. By increasing the number of processors, we expect to get more work done in less time. The speed-up ratio with N processors is not N , however; rather, it is less than N . When multiple processors cooperate on a task, a certain amount of overhead is incurred in keeping all the parts working correctly. This overhead, plus contention for shared resources, lowers the expected gain from additional processors.

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

3. Increased reliability. If functions can be distributed properly among several processors, then the failure of one processor will not halt the system, only slow it down. If we have ten processors and one fails, then each of the remaining nine processors can pick up a share of the work of the failed processor. Thus, the entire system runs only 10 percent slower, rather than failing altogether.

The multiple-processor systems in use today are of two types.

- **Asymmetric multiprocessing** Each processor is assigned a specific task. A boss processor controls the system; the other processors either look to the boss for instruction or have predefined tasks.
- This scheme defines a boss–worker relationship. The boss processor schedules and allocates work to the worker processors.
- **Symmetric multiprocessing (SMP)** Each processor performs all tasks within the operating system. SMP means that all processors are peers; no boss–worker relationship exists between processors.
- The benefit of this model is that many processes can run simultaneously— N processes can run if there are N CPU s— without causing performance to deteriorate significantly.
- However, we must carefully control I/O to ensure that the data reach the appropriate processor. Also, since the CPU s are separate, one may be sitting idle while another is overloaded, resulting in inefficiencies.

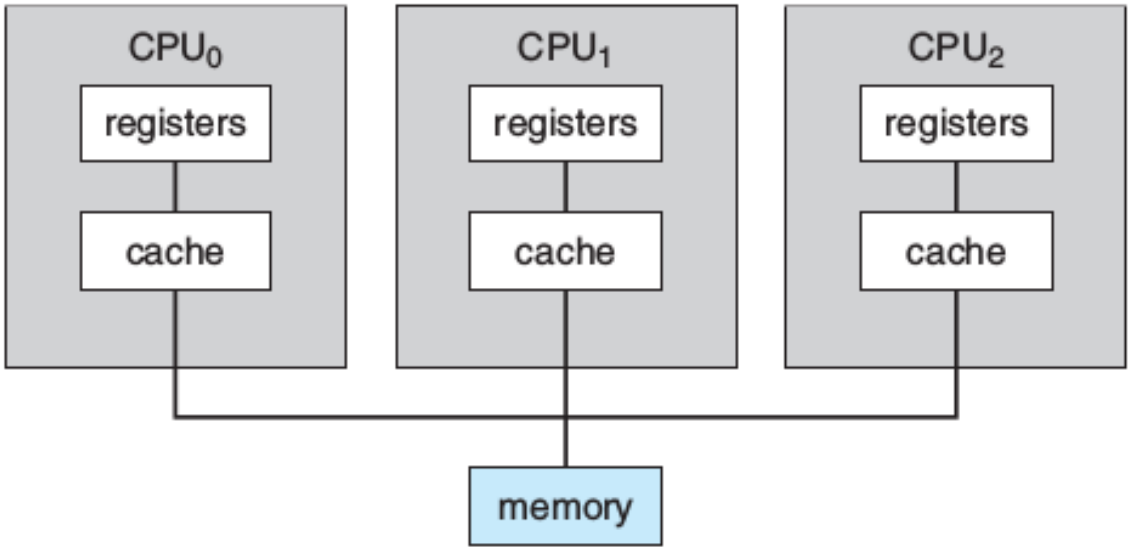


Figure 1.6 Symmetric multiprocessing architecture.

- **Multicore System** Multiple computing cores on a single chip. Such multiprocessor systems are termed multicore.
- They can be more efficient than multiple chips with single cores because on-chip communication is faster than between-chip communication. In addition, one chip with multiple cores uses significantly less power than multiple single-core chips.
- While multicore systems are multiprocessor systems, not all multiprocessor systems are multicore

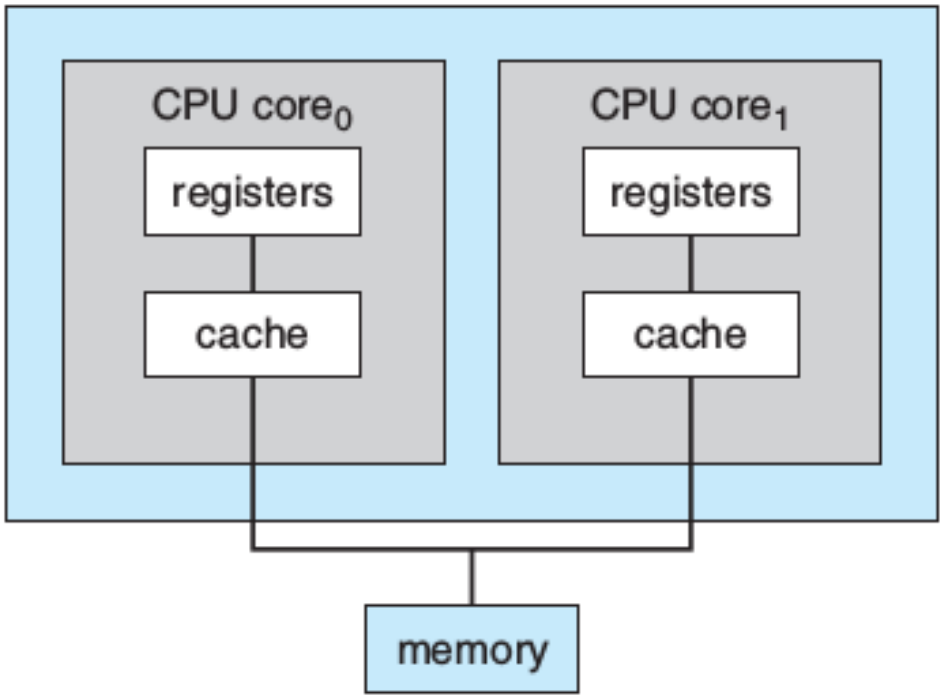


Figure 1.7 A dual-core design with two cores placed on the same chip.

3. Clustered Systems

- Another type of multiprocessor system is a clustered system, which gathers together multiple CPU s.
- They are composed of two or more individual systems—or nodes—joined together. Such systems are considered **loosely coupled**. Each node may be a single processor system or a multicore system.
- Clustering is usually used to provide high-availability service —that is, service will continue even if one or more systems in the cluster fail.
- In **asymmetric clustering**, one machine is in hot-standby mode while the other is running the applications. The hot-standby host machine does nothing but monitor the active server. If that server fails, the hot-standby host becomes the active server.
- In **symmetric clustering**, two or more hosts are running applications and are monitoring each other. This structure is obviously more efficient, as it uses all of the available hardware. However it does require that more than one application be available to run.

- Since a cluster consists of several computer systems connected via a network, clusters can also be used to provide high-performance computing environments.
- Such systems can supply significantly greater computational power than single-processor or even SMP systems because they can run an application concurrently on all computers in the cluster.
- The application must have been written specifically to take advantage of the cluster, however. This involves a technique known as **parallelization**, which divides a program into separate components that run in parallel on individual computers in the cluster.
- Typically, these applications are designed so that once each computing node in the cluster has solved its portion of the problem, the results from all the nodes are combined into a final solution.

Operating System Structure

1. Multiprogramming

- A single program cannot, in general, keep either the CPU or the I/O devices busy at all times. Single users frequently have multiple programs running. Multiprogramming increases CPU utilization by organizing jobs (code and data) so that the CPU always has one to execute.
- The operating system keeps several jobs in memory simultaneously. Since, in general, main memory is too small to accommodate all jobs, the jobs are kept initially on the disk in the **job pool**. This pool consists of all processes residing on disk awaiting allocation of main memory.

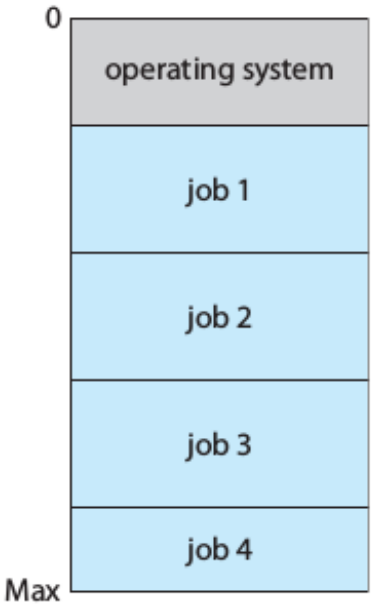


Figure 1.9 Memory layout for a multiprogramming system.

- The operating system picks and begins to execute one of the jobs in memory. Eventually, the job may have to wait for some task, such as an I/O operation, to complete.
- In a non-multiprogrammed system, the CPU would sit idle. In a multiprogrammed system, the operating system simply switches to, and executes, another job.
- When that job needs to wait, the CPU switches to another job, and so on. Eventually, the first job finishes waiting and gets the CPU back. As long as at least one job needs to execute, the CPU is never idle.
- Multiprogrammed systems provide an environment in which the various system resources (for example, CPU , memory, and peripheral devices) are utilized effectively, but they do not provide for user interaction with the computer system.

- **Time sharing (or multitasking)** is a logical extension of multiprogramming. In time-sharing systems, the CPU executes multiple jobs by switching among them, but the switches occur so frequently that the users can interact with each program while it is running.
- Time sharing requires an interactive computer system, which provides direct communication between the user and the system. The user gives instructions to the operating system or to a program directly, using a input device such as a keyboard, mouse, touch pad, or touch screen, and waits for immediate results on an output device. Accordingly, the response time should be short—typically less than one second.
- A time-shared operating system allows many users to share the computer simultaneously. Since each action or command in a time-shared system tends to be short, only a little CPU time is needed for each user. As the system switches rapidly from one user to the next, each user is given the impression that the entire computer system is dedicated to his use, even though it is being shared among many users.
- A time-shared operating system uses CPU scheduling and multiprogramming to provide each user with a small portion of a time-shared computer. Each user has at least one separate program in memory.

- A program loaded into memory and executing is called a **process**. When a process executes, it typically executes for only a short time before it either finishes or needs to perform I/O .
- I/O may be interactive; that is, output goes to a display for the user, and input comes from a user keyboard, mouse, or other device. It may take a long time to complete.
- Rather than let the CPU sit idle as this interactive input takes place, the operating system will rapidly switch the CPU to the program of some other user.
- Time sharing and multiprogramming require that several jobs be kept simultaneously in memory. If several jobs are ready to be brought into memory, and if there is not enough room for all of them, then the system must choose among them. Making this decision involves job scheduling.
- When the operating system selects a job from the job pool, it loads that job into memory for execution. Having several programs in memory at the same time requires some form of memory management.
- In addition, if several jobs are ready to run at the same time, the system must choose which job will run first. Making this decision is CPU scheduling.

Operating System Operations

- Modern operating systems are interrupt driven. If there are no processes to execute, no I/O devices to service, and no users to whom to respond, an operating system will sit quietly, waiting for something to happen.
- Events are almost always signaled by the occurrence of an interrupt or a trap. A trap (or an exception) is a software-generated interrupt caused either by an error (for example, division by zero or invalid memory access) or by a specific request from a user program that an operating-system service be performed.
- For each type of interrupt, separate segments of code in the operating system determine what action should be taken. An **interrupt service routine** is provided to deal with the interrupt.
- Since the operating system and the users share the hardware and software resources of the computer system, we need to make sure that an error in a user program could cause problems only for the one program running.

- With sharing, many processes could be adversely affected by a bug in one program.
- For example, if a process gets stuck in an infinite loop, this loop could prevent the correct operation of many other processes. More subtle errors can occur in a multiprogramming system, where one erroneous program might modify another program, the data of another program, or even the operating system itself.
- Without protection against these sorts of errors, either the computer must execute only one process at a time or all output must be suspect. A properly designed operating system must ensure that an incorrect (or malicious) program cannot cause other programs to execute incorrectly.

1. Dual-Mode Operation

- In order to ensure the proper execution of the operating system, we must be able to distinguish between the execution of operating-system code and user-defined code.
- We need two separate modes of operation: **user mode** and **kernel mode** (also called supervisor mode, system mode, or privileged mode). A bit, called the **mode bit**, is added to the hardware of the computer to indicate the current mode: **kernel (0) or user (1)**.
- When the computer system is executing on behalf of a user application, the system is in user mode. However, when a user application requests a service from the operating system (via a system call), the system must transition from user to kernel mode to fulfill the request.

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

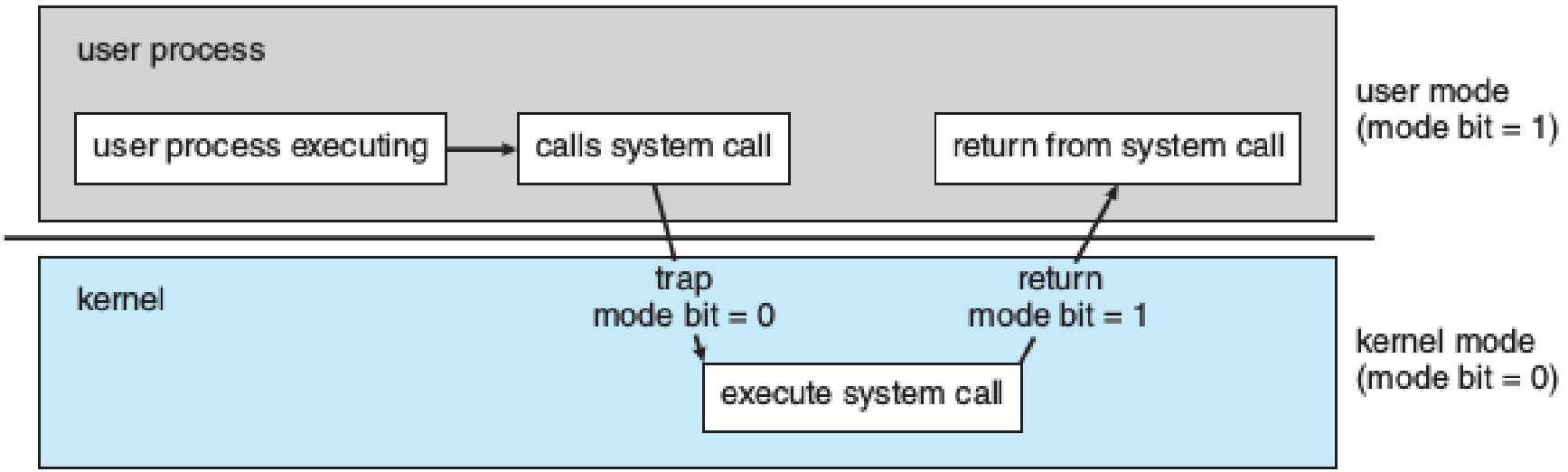


Figure 1.10 Transition from user to kernel mode.

- The dual mode of operation provides us with the means for protecting the operating system from errant users—and errant users from one another.
- Some of the machine instructions that may cause harm are **privileged instructions**. The hardware allows privileged instructions to be executed only in kernel mode.
- If an attempt is made to execute a privileged instruction in user mode, the hardware does not execute the instruction but rather treats it as illegal and traps it to the operating system.
- The instruction to switch to kernel mode is an example of a privileged instruction.

System Call

- Initial control resides in the operating system, where instructions are executed in kernel mode. When control is given to a user application, the mode is set to user mode. Eventually, control is switched back to the operating system via an interrupt, a trap, or a system call.
- **System calls** provide the means for a user program to ask the operating system to perform tasks reserved for the operating system on the user program's behalf.
- A system call usually takes the form of a trap to a specific location in the interrupt vector. When a system call is executed, it is typically treated by the hardware as a software interrupt. Control passes through the interrupt vector to a service routine in the operating system, and the mode bit is set to kernel mode. The system-call service routine is a part of the operating system.
- The kernel examines the interrupting instruction to determine what system call has occurred; a parameter indicates what type of service the user program is requesting. Additional information needed for the request may be passed in registers, on the stack, or in memory (with pointers to the memory locations passed in registers).
- The kernel verifies that the parameters are correct and legal, executes the request, and returns control to the instruction following the system call.

2. Timer

- 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.
- A timer can be set to interrupt the computer after a specified period. The period may be **fixed** or **variable**.
- A variable timer is generally implemented by a fixed-rate clock and a counter.
- The operating system sets the counter. Every time the clock ticks, the counter is decremented. When the counter reaches 0, an interrupt occurs. If the timer interrupts, control transfers automatically to the operating system, which may treat the interrupt as a fatal error or may give the program more time.
- Clearly, instructions that modify the content of the timer are privileged.
- We can use the timer to prevent a user program from running too long. A simple technique is to initialize a counter with the amount of time that a program is allowed to run. Every second, the timer interrupts, and the counter is decremented by 1. As long as the counter is positive, control is returned to the user program. When the counter becomes negative, the operating system terminates the program for exceeding the assigned time limit.

Process

- A program does nothing unless its instructions are executed by a CPU . A program in execution, is a process. A time-shared user program such as a compiler is a process.
- A word-processing program being run by an individual user on a PC is a process
- A process needs certain resources—including CPU time, memory, files, and I/O devices—to accomplish its task. These resources are either given to the process when it is created or allocated to it while it is running.
- In addition to the various physical and logical resources that a process obtains when it is created, various initialization data (input) may be passed along.
- For example, consider a process whose function is to display the status of a file on the screen of a terminal. The process will be given the name of the file as an input and will execute the appropriate instructions and system calls to obtain and display the desired information on the terminal. When the process terminates, the operating system will reclaim any reusable resources.

- A program is a passive entity, like the contents of a file stored on disk, whereas a process is an active entity. The CPU executes one instruction of the process after another, until the process completes.
- The operating system is responsible for the following activities in connection with process management:

  a. Scheduling processes and threads on the CPU s

  b. Creating and deleting both user and system processes

  c. Suspending and resuming processes

  d. Providing mechanisms for process synchronization

  e. Providing mechanisms for process communication

Memory Management

- Main memory is a large array of bytes, ranging in size from hundreds of thousands to billions. Each byte has its own address.
- Main memory is a repository of quickly accessible data shared by the CPU and I/O devices. The central processor reads instructions from main memory during the instruction-fetch cycle and both reads and writes data from main memory during the data-fetch cycle
- The main memory is generally the only large storage device that the CPU is able to address and access directly. For example, for the CPU to process data from disk, those data must first be transferred to main memory by CPU -generated I/O calls. In the same way, instructions must be in memory for the CPU to execute them.
- For a program to be executed, it must be mapped to absolute addresses and loaded into memory. As the program executes, it accesses program instructions and data from memory by generating these absolute addresses. Eventually, the program terminates, its memory space is declared available, and the next program can be loaded and executed.

- To improve both the utilization of the CPU and the speed of the computer’s response to its users, general-purpose computers must keep several programs in memory, creating a need for memory management.
- The operating system is responsible for the following activities in connection with memory management:

  a. Keeping track of which parts of memory are currently being used and who is using them

  b. Deciding which processes (or parts of processes) and data to move into and out of memory

  c. Allocating and deallocating memory space as needed

File System Management

- A file is a collection of related information defined by its creator. Commonly, files represent programs (both source and object forms) and data.
- Data files may be numeric, alphabetic, alphanumeric, or binary. Files may be free-form (for example, text files), or they may be formatted rigidly (for example, fixed fields)
- The operating system is responsible for the following activities in connection with file management:

  a. Creating and deleting files

  b. Creating and deleting directories to organize files

  c. Supporting primitives for manipulating files and directories

  d. Mapping files onto secondary storage

  e. Backing up files on stable (nonvolatile) storage media

Mass Storage Management

- Because main memory is too small to accommodate all data and programs, and because the data that it holds are lost when power is lost, the computer system must provide secondary storage to back up main memory.
- The operating system is responsible for the following activities in connection with disk management:

  a. Free-space management

  b. Storage allocation

  c. Disk scheduling

Caching

- 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.
- 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.

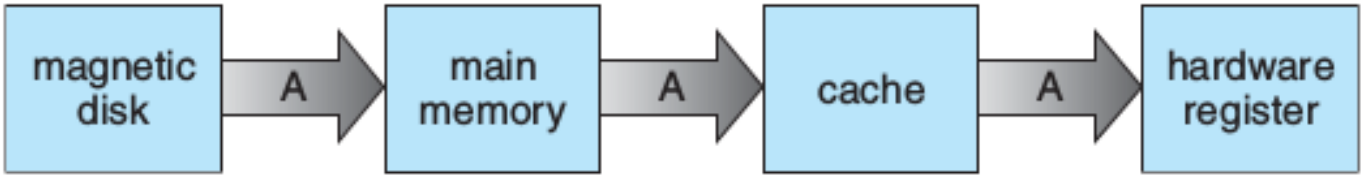


Figure 1.12 Migration of integer A from disk to register.

- In a hierarchical storage structure, the same data may appear in different levels of the storage system. For example, suppose that an integer A that is to be incremented by 1 is located in file B, and file B resides on magnetic disk.
- The increment operation proceeds by first issuing an I/O operation to copy the disk block on which A resides to main memory. This operation is followed by copying A to the cache and to an internal register. Thus, the copy of A appears in several places: on the magnetic disk, in main memory, in the cache, and in an internal register.
- Once the increment takes place in the internal register, the value of A differs in the various storage systems. The value of A becomes the same only after the new value of A is written from the internal register back to the magnetic disk.

- In a computing environment where only one process executes at a time, this arrangement poses no difficulties, since an access to integer A will always be to the copy at the highest level of the hierarchy.
- However, in a multitasking environment, where the CPU is switched back and forth among various processes, extreme care must be taken to ensure that, if several processes wish to access A, then each of these processes will obtain the most recently updated value of A.
- The situation becomes more complicated in a multiprocessor environment where, in addition to maintaining internal registers, each of the CPU s also contains a local cache.
- In such an environment, a copy of A may exist simultaneously in several caches. Since the various CPU s can all execute in parallel, we must make sure that an update to the value of A in one cache is immediately reflected in all other caches where A resides. This situation is called **cache coherency**.

Computing Environments

- Mobile Computing
- Distributed Systems
- Client Server Computing
- Peer to Peer Computing
- Virtualisation
- Cloud Computing
- Real Time Embedded Systems

Mobile Computing

- Mobile computing refers to computing on handheld smartphones and tablet computers. These devices share the distinguishing physical features of being portable and lightweight.
- Historically, compared with desktop and laptop computers, mobile systems gave up screen size, memory capacity, and overall functionality in return for handheld mobile access to services such as e-mail and web browsing.
- The memory capacity and processing speed of mobile devices, however, are more limited than those of PCs. Whereas a smartphone or tablet may have 64 GB in storage, it is not uncommon to find 1 TB in storage on a desktop computer.
- Similarly, because power consumption is such a concern, mobile devices often use processors that are smaller, are slower, and offer fewer processing cores than processors found on traditional desktop and laptop computers.
- Two operating systems currently dominate mobile computing: Apple iOS and Google Android.

Distributed Systems

- A distributed system is a collection of physically separate, possibly heterogeneous, computer systems that are networked to provide users with access to the various resources that the system maintains.
- Access to a shared resource increases computation speed, functionality, data availability, and reliability.
- A network, in the simplest terms, is a communication path between two or more systems. Distributed systems depend on networking for their functionality.
- 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.

Client Server Computing

- Many of today's systems act as server systems to satisfy requests generated by client systems. This form of specialized distributed system, is called a client–server system.
- Server systems can be broadly categorized as compute servers and file servers:

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

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

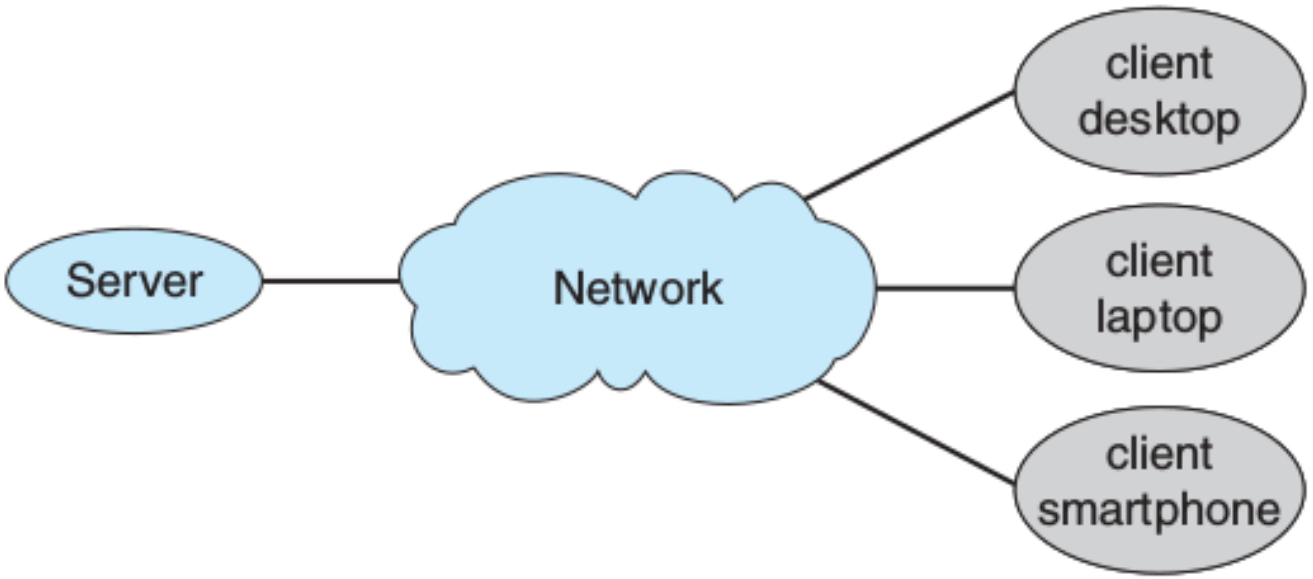


Figure 1.18 General structure of a client–server system.

Peer to Peer Computing

- In this model, clients and servers are not distinguished from one another. Instead, all nodes within the system are considered peers, and each may act as either a client or a server, depending on whether it is requesting or providing a service.
- To participate in a peer-to-peer system, a node must first join the network of peers. Once a node has joined the network, it can begin providing services to—and requesting services from—other nodes in the network.

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

  2. An alternative scheme uses no centralized lookup service. Instead, a peer acting as a client must discover what node provides a desired service by broadcasting a request for the service to all other nodes in the network. The node (or nodes) providing that service responds to the peer making the request.

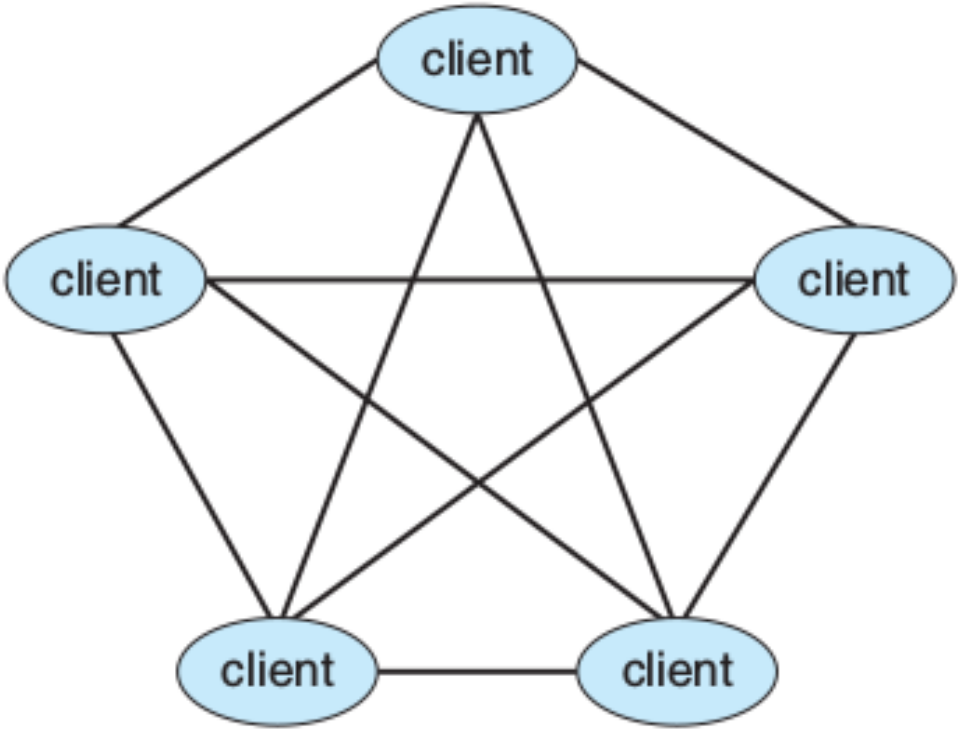


Figure 1.19 Peer-to-peer system with no centralized service.

Virtualization

- Virtualization is a technology that allows operating systems to run as applications within other operating systems.
- With virtualization, in contrast, an operating system that is natively compiled for a particular CPU architecture runs within another operating system also native to that CPU.
- Windows was the host operating system, and the VM ware 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.

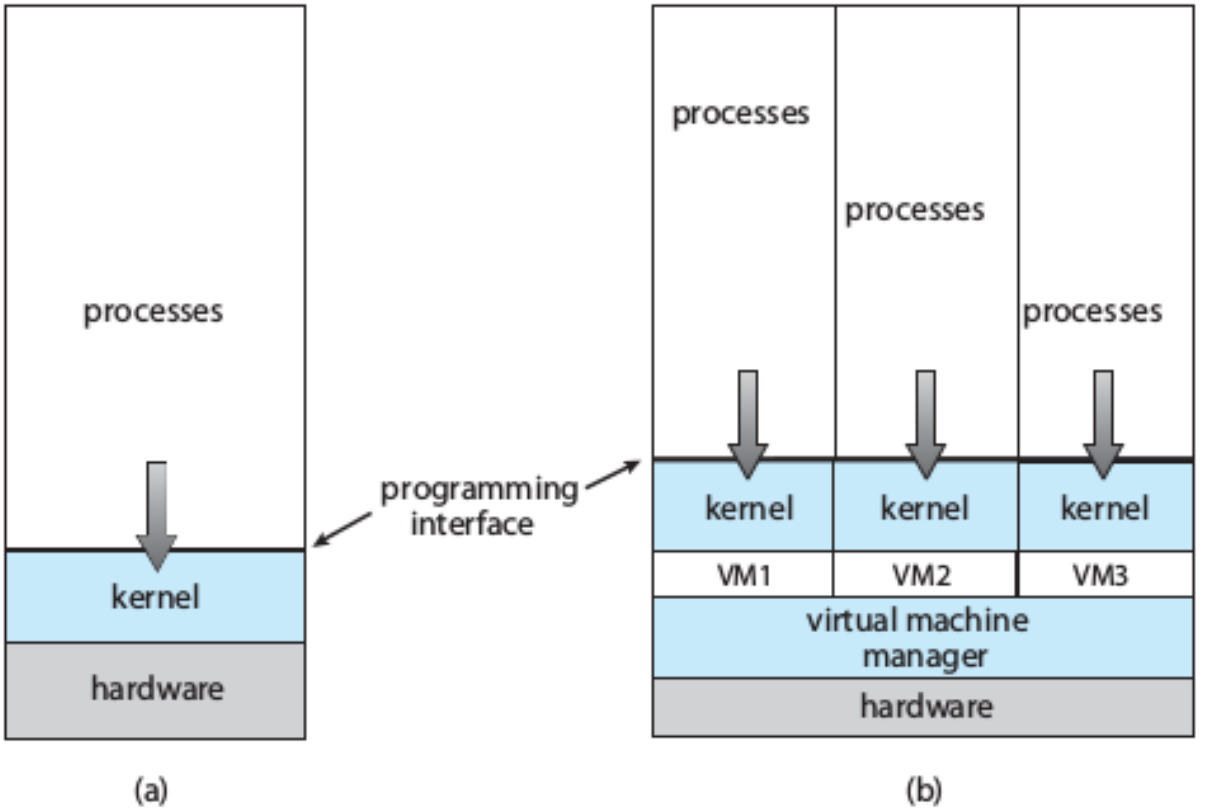


Figure 1.20 VMware.

Cloud Computing

- Cloud computing is a type of computing that delivers computing, storage, and even applications as a service across a network. In some ways, it's a logical extension of virtualization, because it uses virtualization as a base for its functionality.
- There are actually many types of cloud computing, including the following:

  1. **Public cloud** —a cloud available via the Internet to anyone willing to pay for the services

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

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

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

  5. **Infrastructure as a service (IaaS)**—servers or storage available over the Internet (for example, storage available for making backup copies of production data)

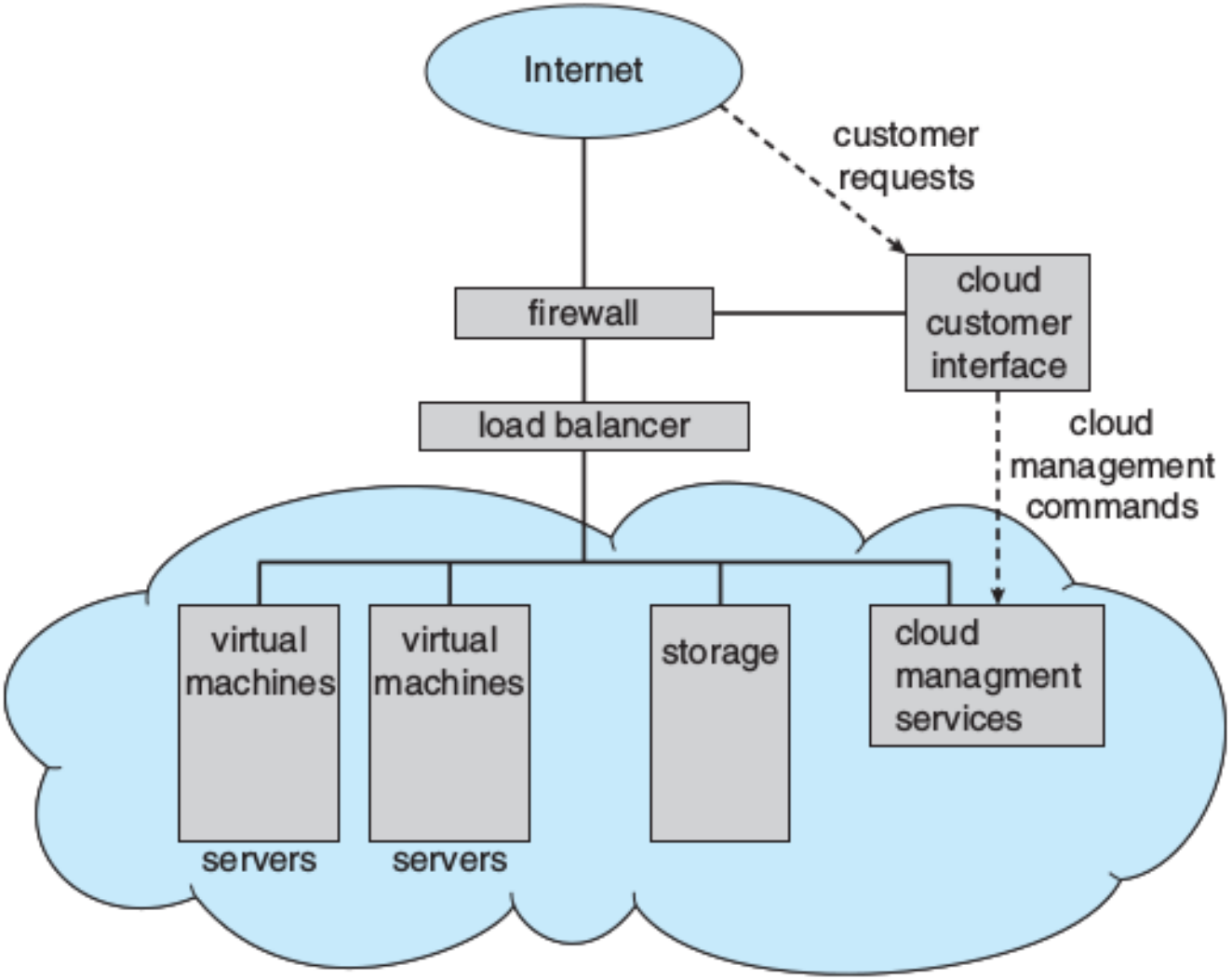


Figure 1.21 Cloud computing.

Real Time Embedded Systems

- They tend to have very specific tasks. The systems they run on are usually primitive, and so the operating systems provide limited features.
- Usually, they have little or no user interface, preferring to spend their time monitoring and managing hardware devices, such as automobile engines and robotic arms.
- Embedded systems almost always run real-time operating systems. A real-time system has well-defined, fixed time constraints. Processing must be done within the defined constraints, or the system will fail. For instance, it would not do for a robot arm to be instructed to halt after it had smashed into the car it was building.
- A real-time system functions correctly only if it returns the correct result within its time constraints. Contrast this system with a time-sharing system, where it is desirable (but not mandatory) to respond quickly, or a batch system, which may have no time constraints at all.

Operating-System Services

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

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

3. **I/O operations**. A running program may require I/O , which may involve a file or an I/O device. For efficiency and protection, users usually cannot control I/O devices directly. Therefore, the operating system must provide a means to do I/O.

4. **File-system manipulation.** Programs need to read and write files and directories. They also need to create and delete them by name, search for a given file, and list file information. Finally, some operating systems include permissions management to allow or deny access to files or directories based on file ownership.

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

6. **Error detection.** The operating system needs to be detecting and correcting errors constantly. Errors may occur in the CPU and memory hardware (such as a memory error or a power failure), in I/O devices (such as a parity error on disk, a connection failure on a network, or lack of paper in the printer), and in the user program (such as an arithmetic overflow, an attempt to access an illegal memory location, or a too-great use of CPU time). For each type of error, the operating system should take the appropriate action to ensure correct and consistent computing.

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

8. **Accounting.** We want to keep track of which users use how much and what kinds of computer resources. This record keeping may be used for accounting (so that users can be billed) or simply for accumulating usage statistics.

9. **Protection and security.** When several separate processes execute concurrently, it should not be possible for one process to interfere with the others or with the operating system itself. Protection involves ensuring that all access to system resources is controlled. Security of the system from outsiders is also important. Such security starts with requiring each user to authenticate himself or herself to the system, usually by means of a password, to gain access to system resources.

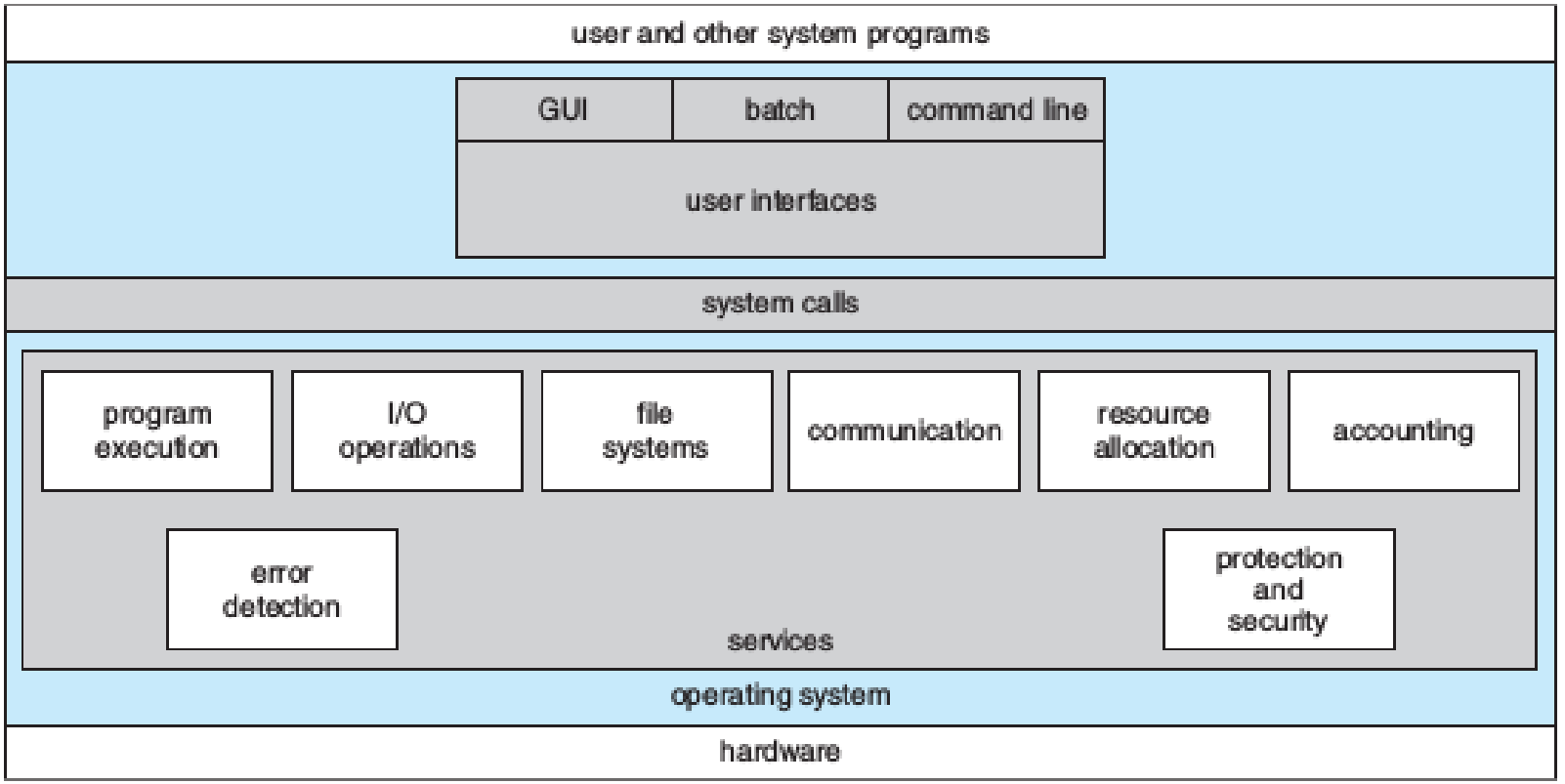


Figure 2.1 A view of operating system services.

System Calls

- System calls provide an interface to the services made available by an operating system.
- Even simple programs may make heavy use of the operating system. Frequently, systems execute thousands of system calls per second.

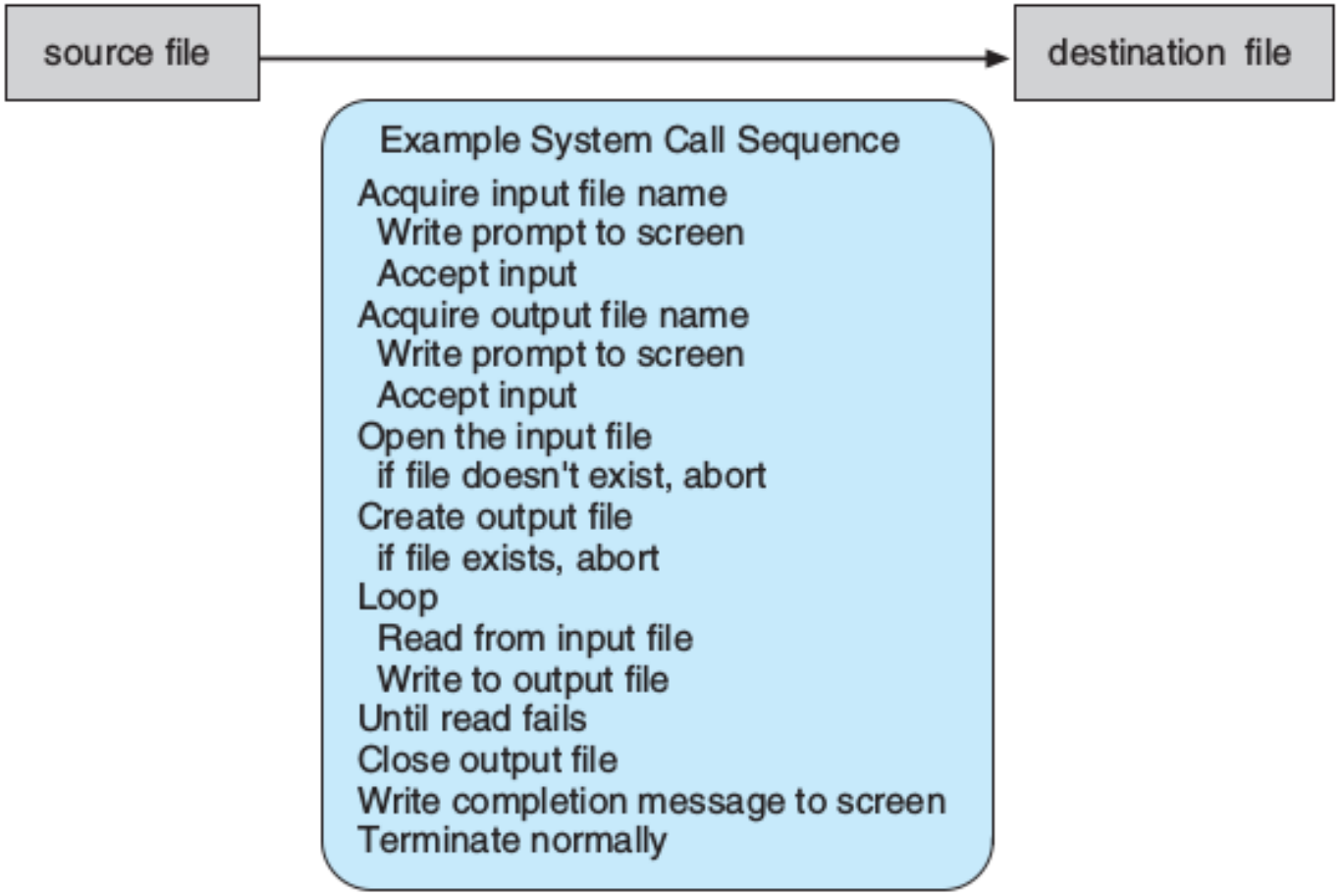


Figure 2.5 Example of how system calls are used.

- An **application programming interface (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.
- The run-time support system (a set of functions built into libraries included with a compiler) provides a **system-call interface** that serves as the link to system calls made available by the operating system.
- The system-call interface intercepts function calls in the API and invokes the necessary system calls within the operating system.
- Typically, a number is associated with each system call, and the system-call interface maintains a table indexed according to these numbers. The system call interface then invokes the intended system call in the operating-system kernel and returns the status of the system call and any return values.

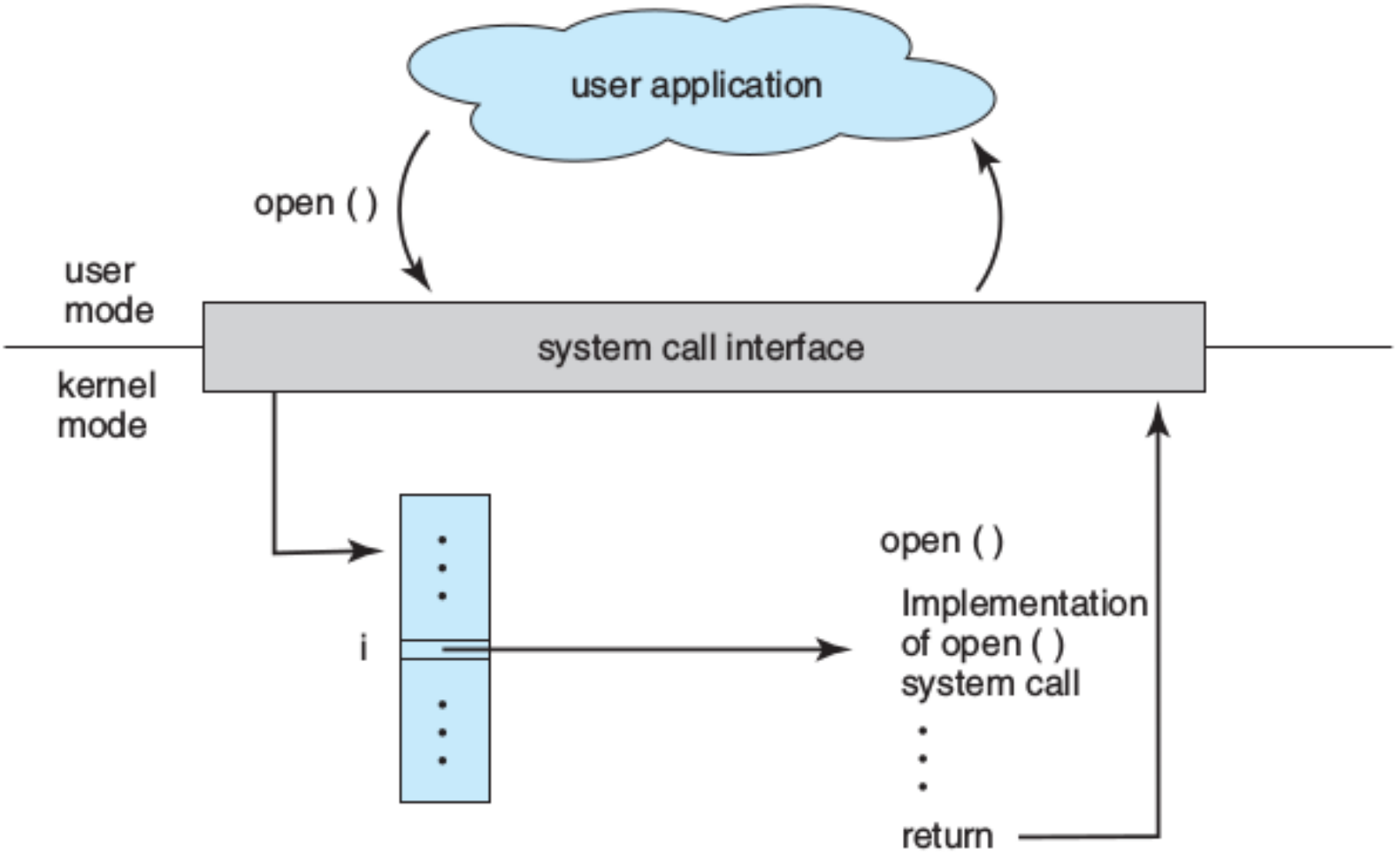


Figure 2.6 The handling of a user application invoking the `open()` system call.

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

  1. The simplest approach is to pass the parameters in registers. In some cases, however, there may be more parameters than registers.

  2. 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.

  3. Parameters also can be placed, or pushed, onto the stack by the program and popped off the stack by the operating system.

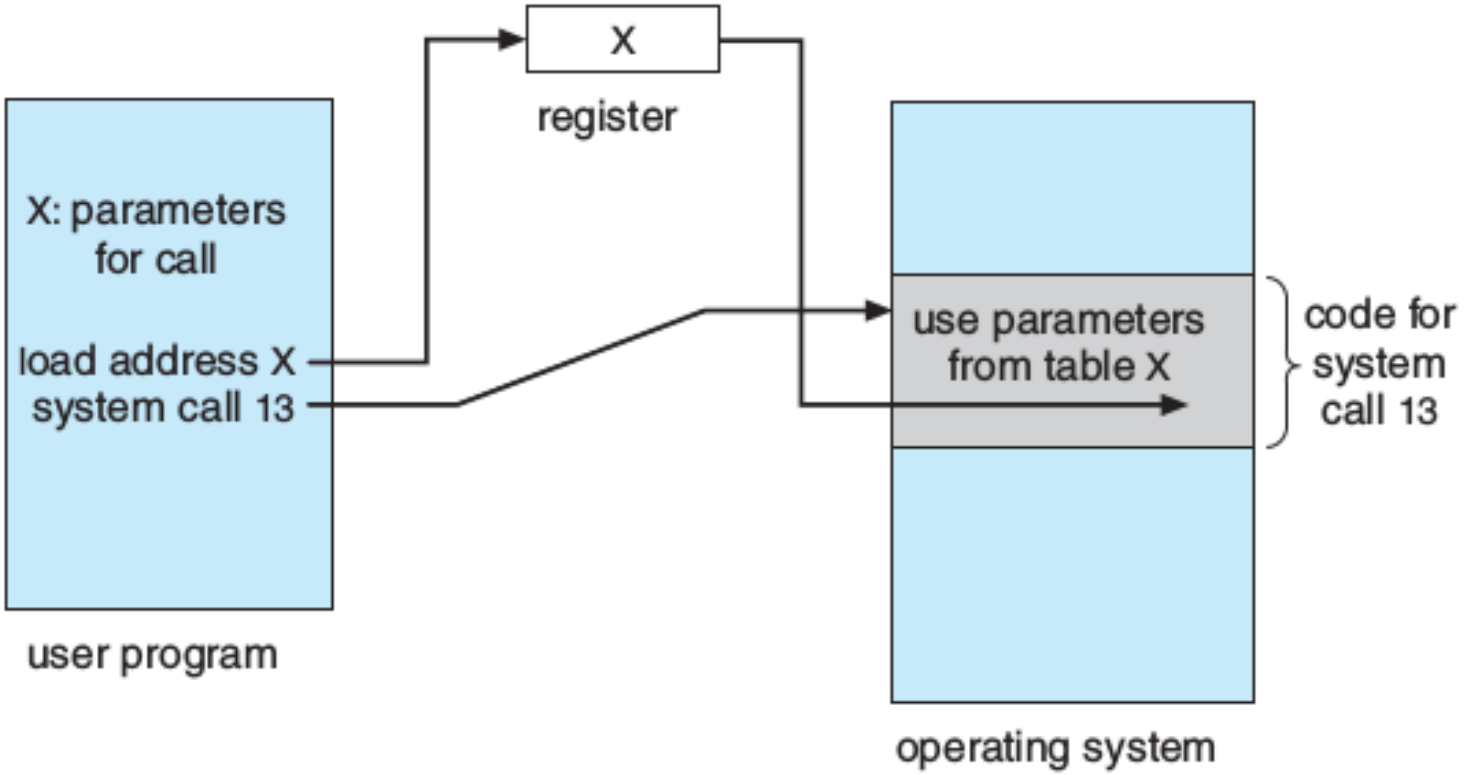


Figure 2.7 Passing of parameters as a table.

Types of System Calls

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

1. **Process Control**

- A running program needs to be able to halt its execution either normally (**end()**) or abnormally (**abort()**).
- A process or job executing one program may want to **load()** and **execute()** another program.
- If both programs continue concurrently, we have created a new job or process (**create_process()**) to be multiprogrammed.
- The ability to determine and reset the attributes of a job or process, including the job's priority, its maximum allowable execution time, and so on (**get_process_attributes() and set_process_attributes()**).
- Terminate a job or process that we created (**terminate_process()**) if we find that it is incorrect or is no longer needed.

- Having created new jobs or processes, we may need to wait for them to finish their execution. We may want to wait for a certain amount of time to pass (**wait_time()**). More probably, we will want to wait for a specific event to occur (**wait_event()**). The jobs or processes should then signal when that event has occurred (**signal_event()**).
- To ensure the integrity of the data being shared, operating systems often provide system calls allowing a process to lock shared data. Then, no other process can access the data until the lock is released. Typically, such system calls include **acquire_lock()** and **release_lock()** .

2. 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). Finally, we need to **close()** the file, indicating that we are no longer using it.
- File attributes include the file name, file type, protection codes, accounting information, and so on. At least two system calls, **get_file_attributes()** and **set file attributes()** , are required for this function.

3. Device Management

- The various resources controlled by the operating system can be thought of as devices. Some of these devices are physical devices (for example, disk drives), while others can be thought of as abstract or virtual devices (for example, files).
- A system with multiple users may require us to first **request()** a device, to ensure exclusive use of it. After we are finished with the device, we **release()** it.
- 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.

4. Information Maintenance

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

5. Communication

- There are two common models of interprocess communication: the message-passing model and the shared-memory model.
- In the **message-passing model**, the communicating processes exchange messages with one another to transfer information. Messages can be exchanged between the processes either directly or indirectly through a common mailbox. Before communication can take place, a connection must be opened.

- Each computer in a network has a host name by which it is commonly known. Similarly, each process has a process name, and this name is translated into an identifier by which the operating system can refer to the process. The **get_hostid()** and **get_processid()** system calls do this translation.
- The identifiers are then passed to the **open_connection()** and **close_connection()** system calls.
- The recipient process usually must give its permission for communication to take place with an **accept_connection()** call. Most processes that will be receiving connections are special-purpose **daemons**, which are system programs provided for that purpose. They execute a **wait_for_connection()** call and are awakened when a connection is made.
- The source of the communication, known as the client, and the receiving daemon, known as a server, then exchange messages by using **read_message()** and **write_message()** system calls. The **close_connection()** call terminates the communication.

- In the **shared-memory model**, processes use **shared_memory_create()** and **shared_memory_attach()** system calls to create and gain access to regions of memory owned by other processes.
- They can then exchange information by reading and writing data in the shared areas.

7. Protection

- Protection provides a mechanism for controlling access to the resources provided by a computer system.
- System calls providing protection include **set_permission()** and **get_permission()** , which manipulate the permission settings of resources such as files and disks.
- The **allow_user()** and **deny_user()** system calls specify whether particular users can—or cannot—be allowed access to certain resources.

System Programs

- System programs, also known as system utilities, provide a convenient environment for program development and execution.

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

  2. **Status information**. Some programs simply ask the system for the date, time, amount of available memory or disk space, number of users, or similar status information. Others are more complex, providing detailed performance, logging, and debugging information.

  3. **File modification**. Several text editors may be available to create and modify the content of files stored on disk or other storage devices.

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

5. Program loading and execution. Once a program is assembled or compiled, it must be loaded into memory to be executed. The system may provide absolute loaders, relocatable loaders, linkage editors, and overlay loaders.

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

Operating System Structure

1. Simple Structure

2. Layered Approach

3. Microkernels

4. Modules

1. Simple Structure

- Such systems started as small, simple, and limited systems and then grew beyond their original scope. MS-DOS is an example of such a system. It was written to provide the most functionality in the least space, so it was not carefully divided into modules.

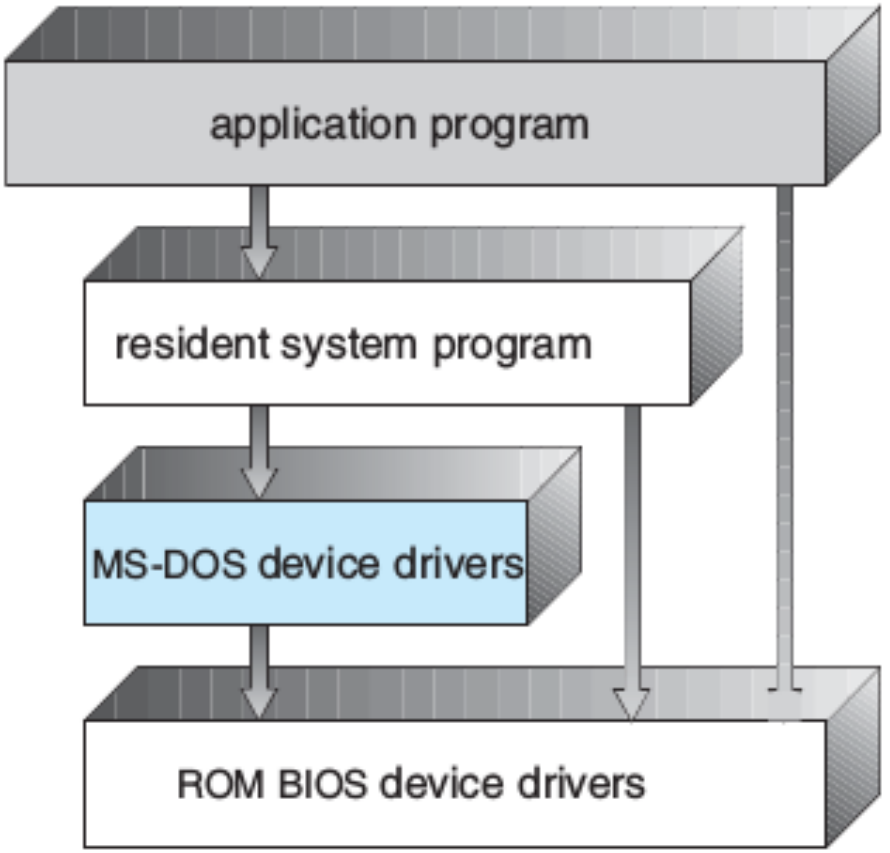


Figure 2.11 MS-DOS layer structure.

- In MS-DOS , the interfaces and levels of functionality are not well separated. For instance, application programs are able to access the basic I/O routines to write directly to the display and disk drives. Such freedom leaves MS-DOS vulnerable to errant (or malicious) programs, causing entire system crashes when user programs fail.
- Another example of limited structuring is the original UNIX operating system. Like MS-DOS , UNIX initially was limited by hardware functionality. It consists of two separable parts: the kernel and the system programs.
- The kernel provides the file system, CPU scheduling, memory management, and other operating-system functions through system calls. That is an enormous amount of functionality to be combined into one level. This monolithic structure was difficult to implement and maintain.

2. Layered Approach

- The operating system is broken into a number of layers (levels). The bottom layer (layer 0) is the hardware; the highest (layer N) is the user interface.
- A typical operating-system layer—say, layer M —consists of data structures and a set of routines that can be invoked by higher-level layers. Layer M, in turn, can invoke operations on lower-level layers.

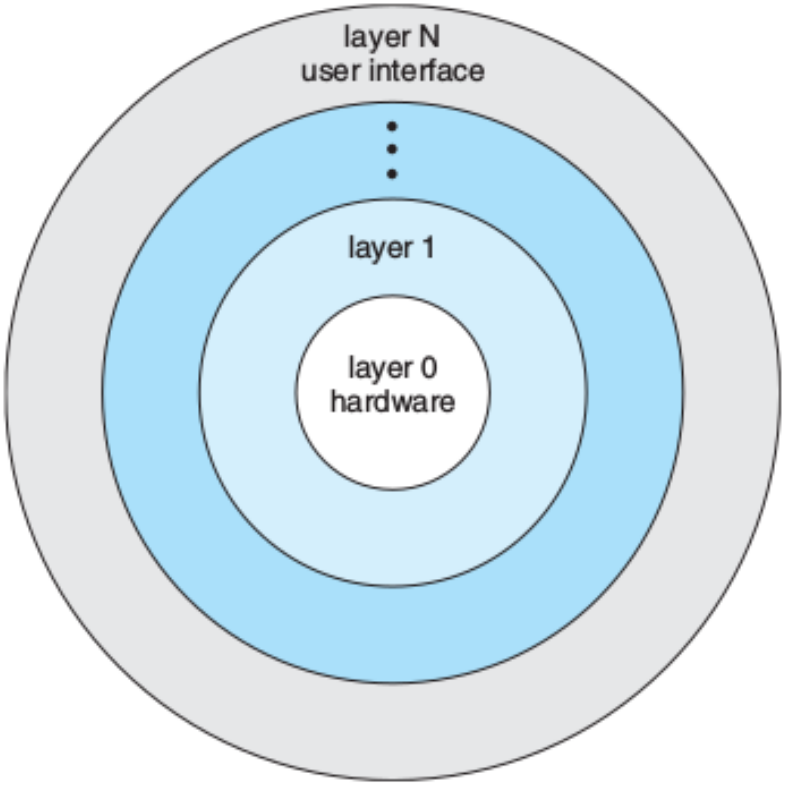


Figure 2.13 A layered operating system.

- The layers are selected so that each uses functions (operations) and services of only lower-level layers. This approach simplifies debugging and system verification.
- The first layer can be debugged without any concern for the rest of the system, because, by definition, it uses only the basic hardware (which is assumed correct) to implement its functions.
- Once the first layer is debugged, its correct functioning can be assumed while the second layer is debugged, and so on. If an error is found during the debugging of a particular layer, the error must be on that layer, because the layers below it are already debugged.
- Each layer is implemented only with operations provided by lower-level layers. A layer does not need to know how these operations are implemented; it needs to know only what these operations do. Hence, each layer hides the existence of certain data structures, operations, and hardware from higher-level layers.

- The major difficulty with the layered approach involves appropriately defining the various layers. Because a layer can use only lower-level layers, careful planning is necessary.
- Layered implementations tend to be less efficient than other types. For instance, when a user program executes an I/O operation, it executes a system call that is trapped to the I/O layer, which calls the memory-management layer, which in turn calls the CPU -scheduling layer, which is then passed to the hardware.
- At each layer, the parameters may be modified, data may need to be passed, and so on. Each layer adds overhead to the system call. The net result is a system call that takes longer than does one on a nonlayered system.

3. Microkernels

- This method structures the operating system by removing all nonessential components from the kernel and implementing them as system and user-level programs. The result is a smaller kernel.
- The main function of the microkernel is to provide communication between the client program and the various services that are also running in user space. Communication is provided through message passing.
- For example, if the client program wishes to access a file, it must interact with the file server. The client program and service never interact directly. Rather, they communicate indirectly by exchanging messages with the microkernel.
- One benefit of the microkernel approach is that it makes extending the operating system easier. All new services are added to user space and consequently do not require modification of the kernel.

- When the kernel does have to be modified, the changes tend to be fewer, because the microkernel is a smaller kernel.
- The resulting operating system is easier to port from one hardware design to another.
- The microkernel also provides more security and reliability, since most services are running as user—rather than kernel—processes. If a service fails, the rest of the operating system remains untouched.

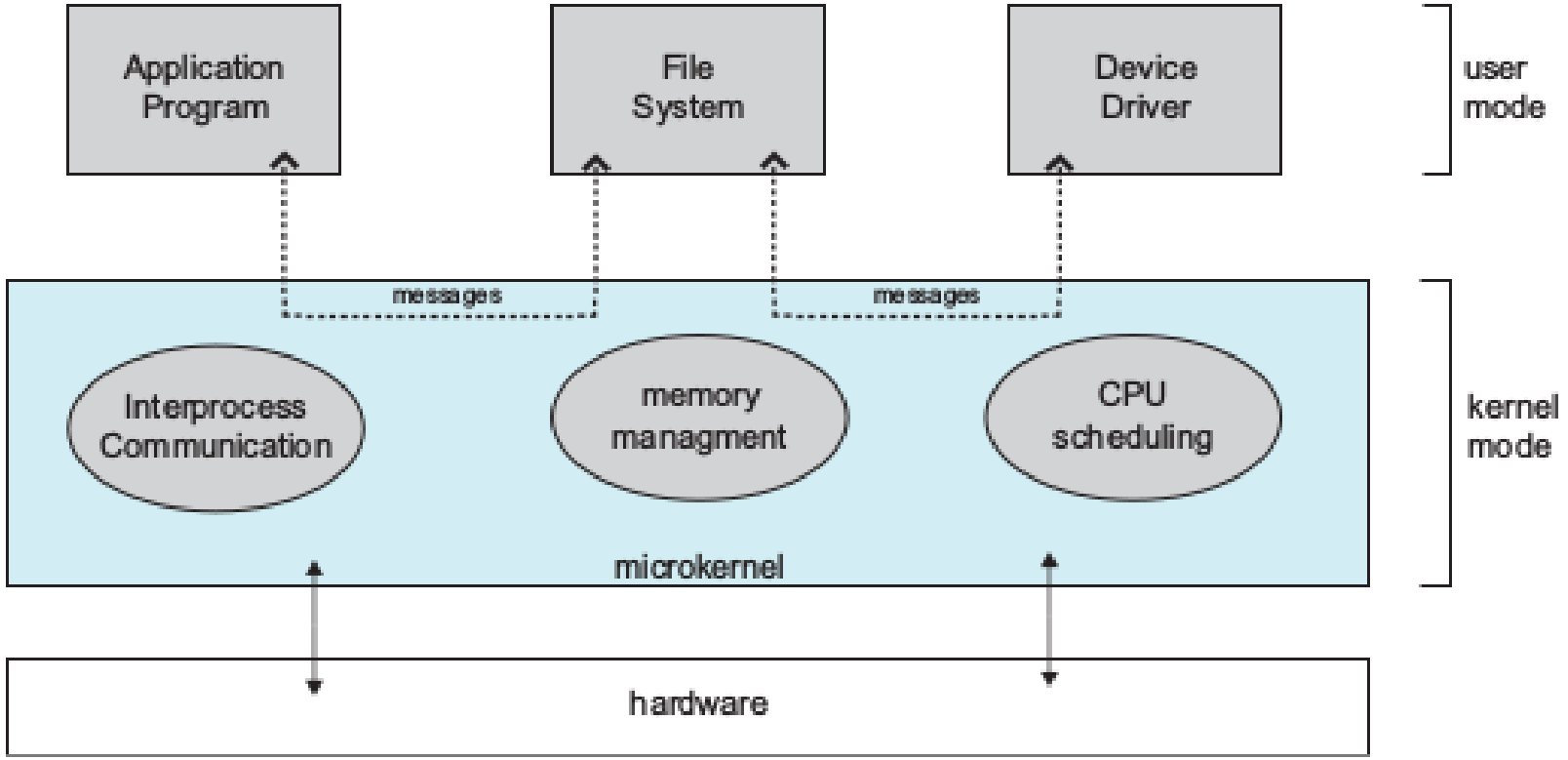


Figure 2.14 Architecture of a typical microkernel.

4. Modules

- The kernel has a set of core components and links in additional services via modules, either at boot time or during run time.
- This type of design is common in modern implementations of UNIX , such as Solaris, Linux, and Mac OS X , as well as Windows.
- Kernel provides core services while other services are implemented dynamically, as the kernel is running.
- Linking services dynamically is preferable to adding new features directly to the kernel, which would require recompiling the kernel every time a change was made.
- The overall result resembles a layered system in that each kernel section has defined, protected interfaces; but it is more flexible than a layered system, because any module can call any other module.
- The approach is also similar to the microkernel approach in that the primary module has only core functions and knowledge of how to load and communicate with other modules; but it is more efficient, because modules do not need to invoke message passing in order to communicate.

System Boot

- After an operating system is generated, it must be made available for use by the hardware. The procedure of starting a computer by loading the kernel is known as **booting** the system.
- On most computer systems, a small piece of code known as the **bootstrap program** or **bootstrap loader** locates the kernel, loads it into main memory, and starts its execution.
- When a CPU receives a reset event—for instance, when it is powered up or rebooted —the instruction register is loaded with a predefined memory location, and execution starts there. At that location is the initial bootstrap program.
- Now that the full bootstrap program has been loaded, it can traverse the file system to find the operating system kernel, load it into memory, and start its execution. It is only at this point that the system is said to be running.

The Process

- A process is a program in execution. A process is more than the program code, which is sometimes known as the **text section**.
- It also includes the current activity, as represented by the value of the **program counter** and the contents of the processor's registers.
- A process generally also includes the **process stack**, which contains temporary data (such as function parameters, return addresses, and local variables), and a **data section**, which contains global variables.
- A process may also include a **heap**, which is memory that is dynamically allocated during process run time.
- 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.

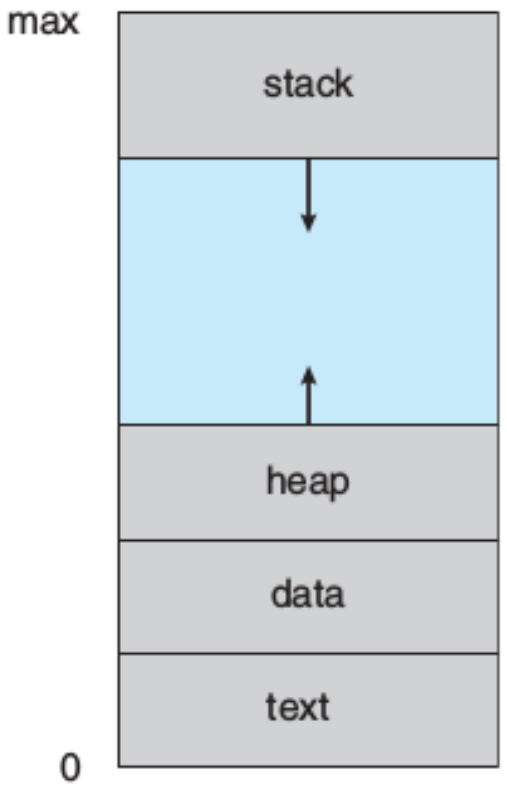


Figure 3.1 Process in memory.

- 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.

- A process itself can be an execution environment for other code. The Java programming environment provides a good example.
- An executable Java program is executed within the Java virtual machine (JVM). The JVM executes as a process that interprets the loaded Java code and takes actions.
- For example, to run the compiled Java program Program.class, we would enter

  java Program
- The command **java** runs the JVM as an ordinary process, which in turns executes the Java program Program in the virtual machine.

Process State

- As a process executes, it changes state. A process may be in one of the following states:

  1. **New**. The process is being created.

  2. **Running**. Instructions are being executed.

  3. **Waiting**. The process is waiting for some event to occur (such as an I/O completion or reception of a signal).

  4. **Ready**. The process is waiting to be assigned to a processor.

  5. **Terminated**. The process has finished execution.

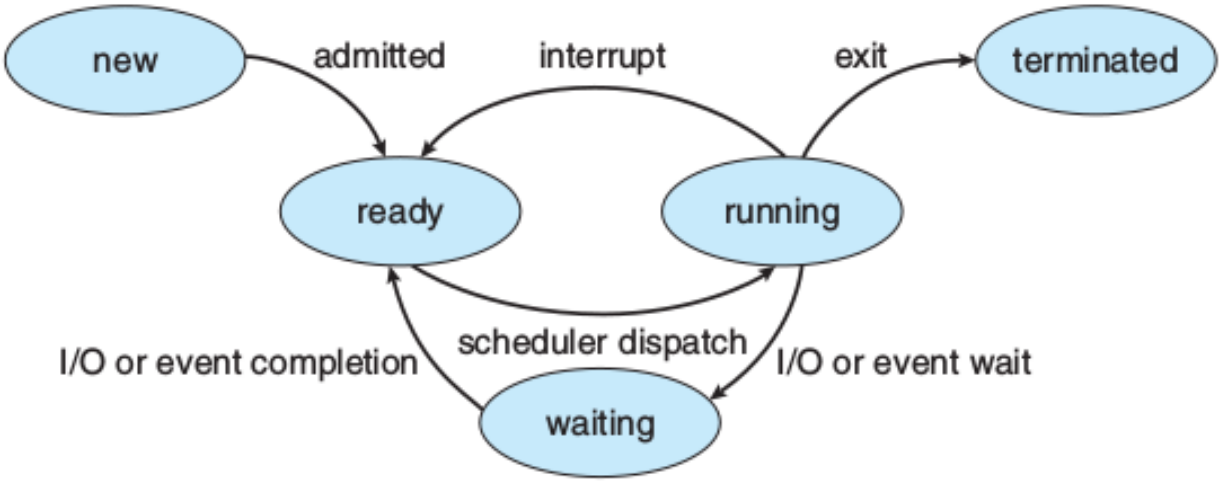


Figure 3.2 Diagram of process state.

Process Control Block

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

  1. **Process state**. The state may be new, ready, running, waiting, halted, and so on.

  2. **Program counter**. The counter indicates the address of the next instruction to be executed for this process.

  3. **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. Along with the program counter, this state information must be saved when an interrupt occurs, to allow the process to be continued correctly afterward.

  4. **CPU -scheduling information**. This information includes a process priority, pointers to scheduling queues, and any other scheduling parameters.

5. **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.

6. **Accounting information**. This information includes the amount of CPU and real time used, time limits, job or process numbers, and so on.

7. **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.

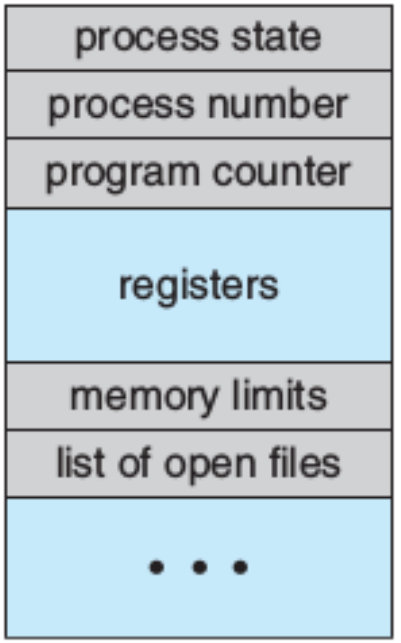


Figure 3.3 Process control block (PCB).

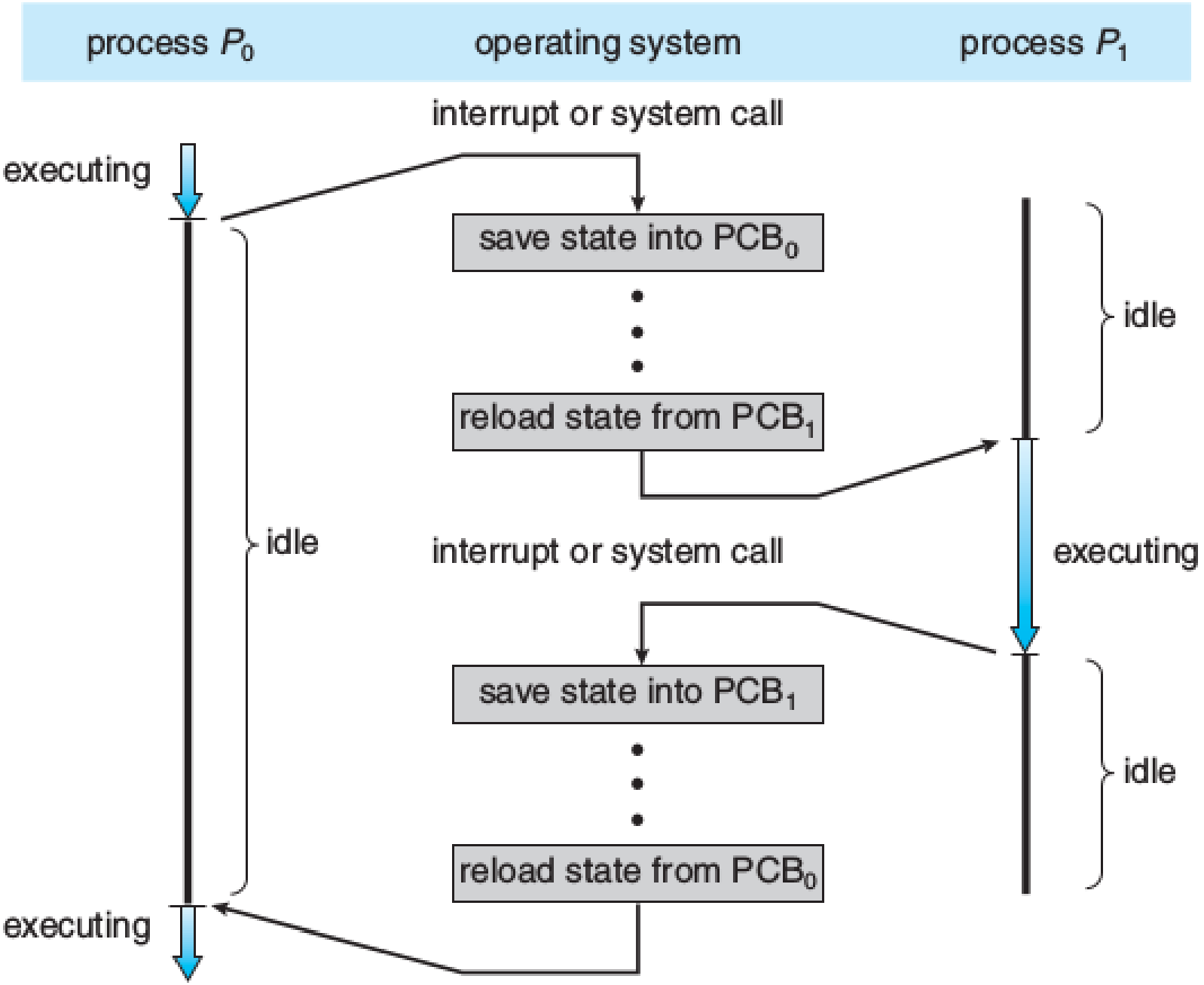


Figure 3.4 Diagram showing CPU switch from process to process.

Threads

- A process is a program that performs a single thread of execution.
- 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. On a system that supports threads, the PCB is expanded to include information for each thread.
- Simultaneously type in characters and run the spell checker within the same process, for example

Process Scheduling

- The objective of multiprogramming is to have some process running at all times, to maximize CPU utilization.
- The objective of time sharing is to switch the CPU among processes so frequently that users can interact with each program while it is running.
- To meet these objectives, the process scheduler selects an available process (possibly from a set of several available processes) for program execution on the CPU.
- For a single-processor system, there will never be more than one running process. If there are more processes, the rest will have to wait until the CPU is free and can be rescheduled.

Scheduling Queues

- As processes enter the system, they are put into a **job queue**, which consists of all processes in the system.
- The processes that are residing in main memory and are ready and waiting to execute are kept on a list called the **ready queue**. A ready-queue header contains pointers to the first and final PCBs in the list. Each PCB includes a pointer field that points to the next PCB in the ready queue.
- When a process is allocated the CPU , it executes for a while and eventually quits, is interrupted, or waits for the occurrence of a particular event, such as the completion of an I/O request.
- Suppose the process makes an I/O request to a shared device, such as a disk. Since there are many processes in the system, the disk may be busy with the I/O request of some other process. The process therefore may have to wait for the disk. The list of processes waiting for a particular I/O device is called a **device queue**.

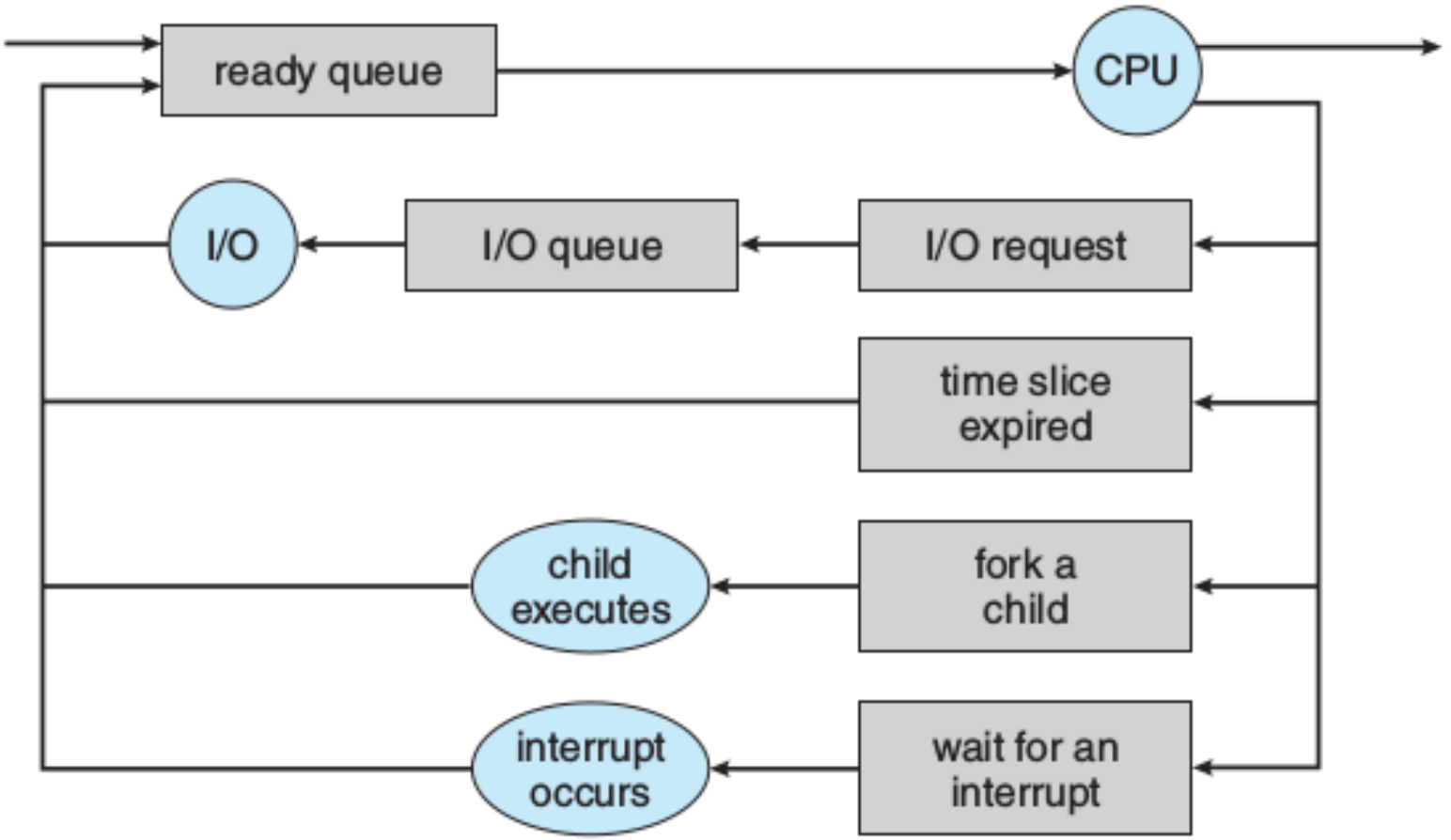


Figure 3.6 Queueing-diagram representation of process scheduling.

- Each rectangular box represents a queue. Two types of queues are present: the ready queue and a set of device queues. The circles represent the resources that serve the queues, and the arrows indicate the flow of processes in the system.

- A new process is initially put in the ready queue. It waits there until it is selected for execution, or dispatched. Once the process is allocated the CPU and is executing, one of several events could occur:

  1. The process could issue an I/O request and then be placed in an I/O queue.

  2. The process could create a new child process and wait for the child's termination.

  3. The process could be removed forcibly from the CPU , as a result of an interrupt, and be put back in the ready queue.

- In the first two cases, the process eventually switches from the waiting state to the ready state and is then put back in the ready queue.
- A process continues this cycle until it terminates, at which time it is removed from all queues and has its PCB and resources deallocated.

Schedulers

- A process migrates among the various scheduling queues throughout its lifetime. The operating system must select, for scheduling purposes, processes from these queues. The selection process is carried out by the appropriate scheduler.
- The **long-term scheduler**, or **job scheduler**, selects processes from job pool and loads them into memory for execution.
- The **short-term scheduler**, or **CPU scheduler**, selects from among the processes that are ready to execute and allocates the CPU to one of them.
- The short-term scheduler must select a new process for the CPU frequently. A process may execute for only a few milliseconds before waiting for an I/O request.
- Because of the short time between executions, the short-term scheduler must be fast.

- The long-term scheduler executes much less frequently; minutes may separate the creation of one new process and the next.
- The long-term scheduler controls the **degree of multiprogramming** (the number of processes in memory). If the degree of multiprogramming is stable, then the average rate of process creation must be equal to the average departure rate of processes leaving the system.
- Thus, the long-term scheduler may need to be invoked only when a process leaves the system.
- Because of the longer interval between executions, the long-term scheduler can afford to take more time to decide which process should be selected for execution.

- 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.
- It is important that the long-term scheduler select a good process mix of I/O -bound and CPU -bound processes.
- If all processes are I/O bound, the ready queue will almost always be empty, and the short-term scheduler will have little to do.
- If all processes are CPU bound, the I/O waiting queue will almost always be empty, devices will go unused, and again the system will be unbalanced.
- The system with the best performance will thus have a combination of CPU -bound and I/O -bound processes.

- A **medium-term scheduler** is removes a process from memory and thus reduces the degree of multiprogramming.
- Later, the process can be reintroduced into memory, and its execution can be continued where it left off. This scheme is called **swapping**.
- The process is swapped out, and is later swapped in, by the medium-term scheduler.
- Swapping may be necessary to improve the process mix or because a change in memory requirements has overcommitted available memory, requiring memory to be freed up.

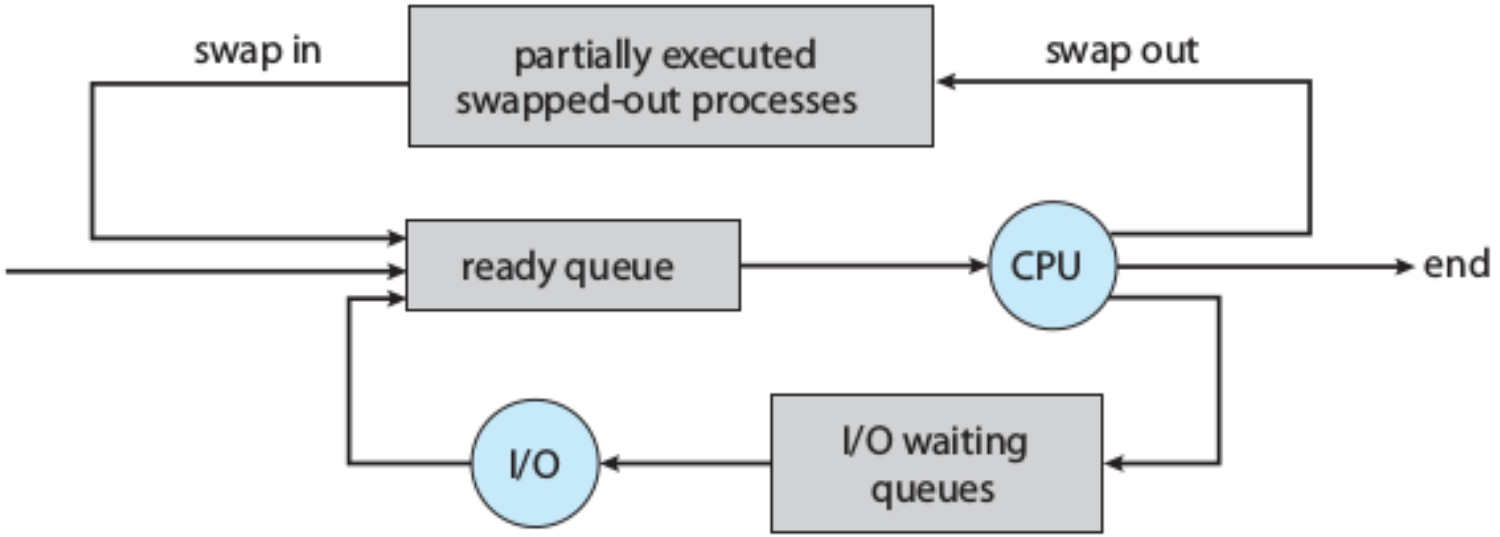


Figure 3.7 Addition of medium-term scheduling to the queueing diagram.

Context Switch

- When an interrupt occurs, the system needs to save the current context of the process running on the CPU 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 and memory-management information.
- Switching the CPU to another process requires performing a **state save** of the current process and a **state restore** of a different process. This task is known as a **context switch**.
- When a context switch occurs, the kernel saves the context of the old process in its PCB and loads the saved context of the new process scheduled to run.
- Context-switch time is pure overhead, because the system does no useful work while switching.

Process Creation

- During the course of execution, a process may create several new processes. The creating process is called a **parent** process, and the new processes are called the **children** of that process.
- 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.
- 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.
- 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, image.jpg —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 image.jpg. Using that file name, it will open the file and write the contents out. It may also get the name of the output device.

- 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.

- In UNIX, a new process is created by the **fork()** system call. The new process consists of a copy of the address space of the original process.
- This mechanism allows the parent process to communicate easily with its child process.
- Both processes (the parent and the child) continue execution at the instruction after the fork() , with one difference: the return code for the fork() is zero for the new (child) process, whereas the (nonzero) process identifier of the child is returned to the parent.
- After a fork() system call, one of the two processes typically uses the **exec()** system call to replace the process's memory space with a new program.
- The exec() system call loads a binary file into memory (destroying the memory image of the program containing the exec() system call) and starts its execution.
- The parent can then create more children; or, if it has nothing else to do while the child runs, it can issue a **wait()** system call to move itself off the ready queue until the termination of the child.

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

int main()
{
pid_t pid;

    /* 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 3.9 Creating a separate process using the UNIX `fork()` system call.

- In the example, the child process overlays its address space with the UNIX command /bin/ls (used to get a directory listing) using the execlp() system call (execlp() is a version of the exec() system call).
- The parent waits for the child process to complete with the wait() system call.
- When the child process completes (by either implicitly or explicitly invoking exit()), the parent process resumes from the call to wait() , where it completes using the exit() system call.

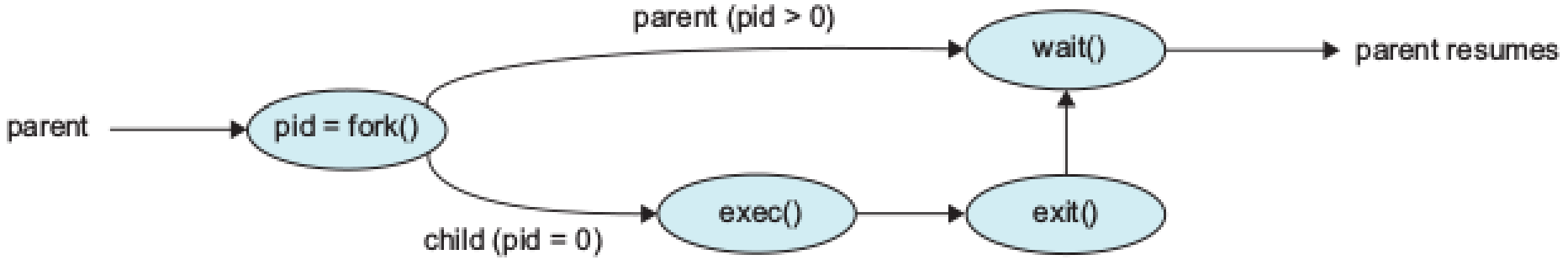


Figure 3.10 Process creation using the `fork()` system call.

Process Termination

- A process terminates when it finishes executing its final statement and asks the operating system to delete it by using the exit() system call.
- At that point, the process may return a status value (typically an integer) to its parent process (via the wait() system call). All the resources of the process—including physical and virtual memory, open files, and I/O buffers—are deallocated by the operating system.
- A process can cause the termination of another process via an appropriate system call. Usually, such a system call can be invoked only by the parent of the process that is to be terminated. Otherwise, users could arbitrarily kill each other's jobs.
- 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:

  1. The child has exceeded its usage of some of the resources that it has been allocated.

  2. The task assigned to the child is no longer required.

  3. The parent is exiting, and the operating system does not allow a child to continue if its parent terminates.

- 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.
- The wait() system call is passed a parameter that allows the parent to obtain the exit status of the child. This system call also returns the process identifier of the terminated child so that the parent can tell which of its children has terminated:

```
pid t pid;

int status;

pid = wait(&status);
```

- 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.

- If a parent did not invoke wait() and instead terminated, leaving its child processes as **orphans**. Linux and UNIX address this scenario by assigning the init process as the new parent to orphan processes.
- 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.

Shell Programming

- Cat command

  cat [OPTION] [FILE]

  1. cat //listen to standard input

     This is a new line

     This is a new line

  2. cat > readme.txt //writes contents

     This is a readme file.

     This is a new line.

  3. cat readme.txt //prints contents

     This is a readme file.

     This is a new line.

  4. cat >> readme.txt //appends contents

     This is an appended line.

Awk command

- awk options 'selection _criteria {action }' input-file > output-file

  Options:

- -f program-file : Reads the AWK program source from the file program-file, instead of from the first command line argument.
- -F fs : Use fs for the input field separator

  1. awk '{print}' employee.txt //print every line of file

  2. awk '/manager/ {print}' employee.txt //print lines which contain pattern

  3. Splitting a Line Into Fields : For each record i.e line, the awk command splits the record delimited by whitespace character by default and stores it in the $n variables. If the line has 4 words, it will be stored in $1, $2, $3 and $4 respectively. Also, $0 represents the whole line.

  awk '{print $1,$4}' employee.txt

Built-In Variables In Awk

- Awk's built-in variables include the field variables—$1, $2, $3, and so on ($0 is the entire line) — that break a line of text into individual words or pieces called fields.

  1. NR: NR command keeps a current count of the number of input records. Remember that records are usually lines. Awk command performs the pattern/action statements once for each record in a file.

  2. NF: NF command keeps a count of the number of fields within the current input record.

  3. FS: FS command contains the field separator character which is used to divide fields on the input line. The default is "white space", meaning space and tab characters. FS can be reassigned to another character (typically in BEGIN) to change the field separator.

- awk '{print NR,$0}' employee.txt

  prints line no. and the line
- awk '{print $1,$NF}' employee.txt

  pritns first and last word of each line
- awk 'NR==3, NR==6 {print NR,$0}' employee.txt

  prints line 3 to 6
- awk '{print NR "- " $1 }' employee.txt

  prints line no., then ‘–‘ and then first word

Threads

- A thread is a basic unit of CPU utilization; it comprises a thread ID , a program counter, a register set, and a stack.
- It shares with other threads belonging to the same process its code section, data section, and other operating-system resources, such as open files and signals.
- A traditional (or heavyweight) process has a single thread of control. If a process has multiple threads of control, it can perform more than one task at a time.
- A web browser might have one thread display images or text while another thread retrieves data from the network, for example.

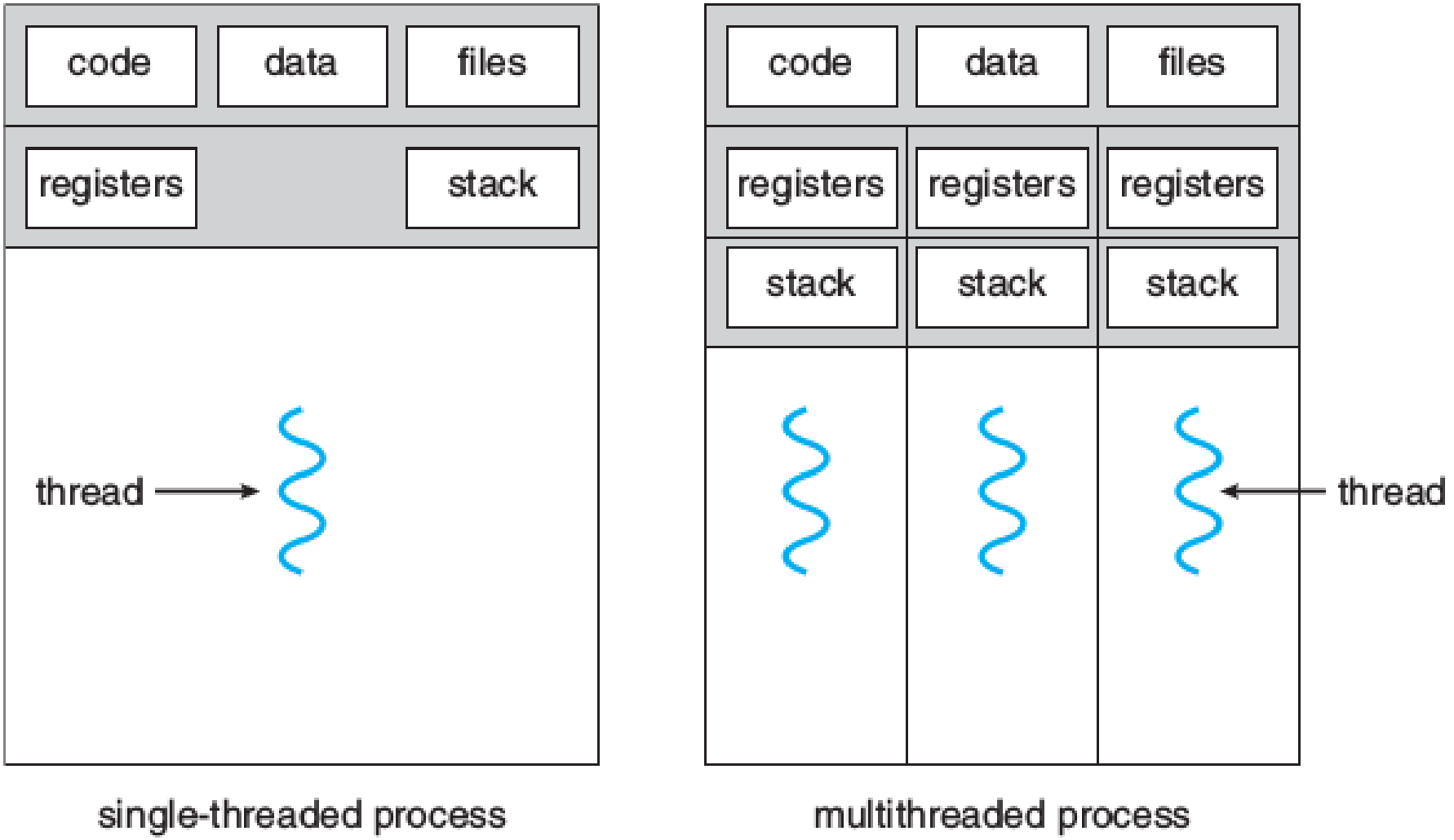


Figure 4.1 Single-threaded and multithreaded processes.

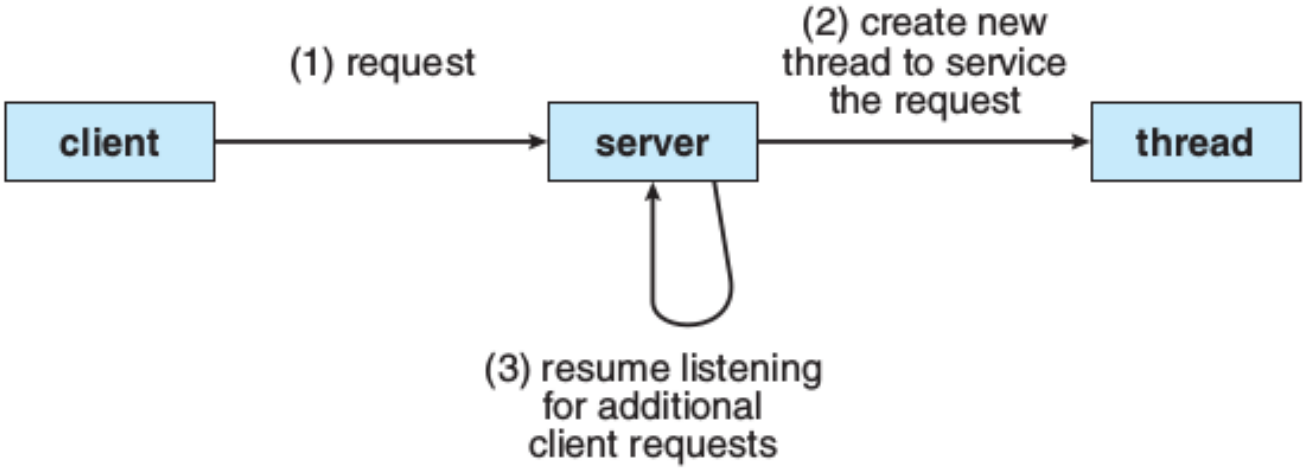


Figure 4.2 Multithreaded server architecture.

- A busy web server may have several (perhaps thousands of) clients concurrently accessing it.
- 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. Process creation is time consuming and resource intensive.
- If the web-server process is multithreaded, the server will create a separate thread that listens for client requests. When a request is made, rather than creating another process, the server creates a new thread to service the request and resume listening for additional requests.

Benefits

1. **Responsiveness**. Multithreading an interactive application may allow a program to continue running even if part of it is blocked or is performing a lengthy operation, thereby increasing responsiveness to the user. This quality is especially useful in designing user interfaces.

2. Resource sharing. Processes can only share resources through techniques such as shared memory and message passing. However, threads share the memory and the resources of the process to which they belong by default. The benefit of sharing code and data is that it allows an application to have several different threads of activity within the same address space.

3. **Economy**. Allocating memory and resources for process creation is costly. Because threads share the resources of the process to which they belong, it is more economical to create and context-switch threads.

4. **Scalability**. In a multiprocessor architecture, threads may run in parallel on different processing cores. A single-threaded process can run on only one processor, regardless how many are available.

Multicore Programming

- Multiple computing cores are placed on a single chip. Each core appears as a separate processor to the operating system.
- 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, because the processing core is capable of executing only one thread at a time.
- On a system with multiple cores, however, concurrency means that the threads can run in parallel, because the system can assign a separate thread to each core.
- A system is **parallel** if it can perform more than one task simultaneously. In contrast, a **concurrent** system supports more than one task by allowing all the tasks to make progress.

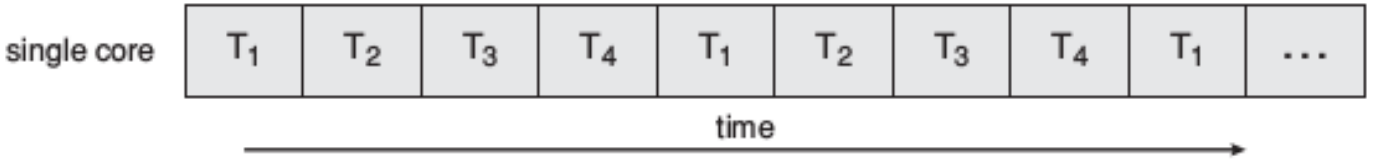


Figure 4.3 Concurrent execution on a single-core system.

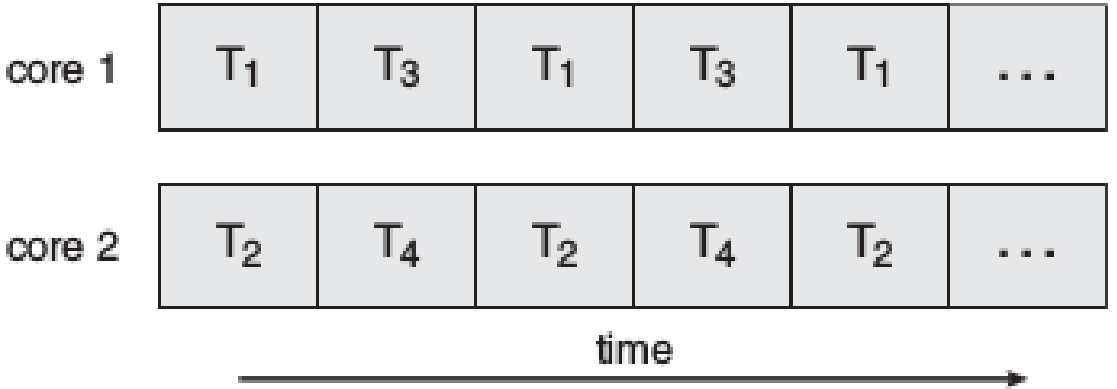


Figure 4.4 Parallel execution on a multicore system.

- Modern Intel CPUs frequently support two threads per core, while the Oracle T4 CPU supports eight threads per core.
- This support means that multiple threads can be loaded into the core for fast switching.

Programming challenges for multicore systems

1. **Identifying tasks**. This involves examining applications to find area that can be divided into separate, concurrent tasks. Ideally, tasks are independent of one another and thus can run in parallel on individual cores.

2. **Balance**. While identifying tasks that can run in parallel, programmers must also ensure that the tasks perform equal work of equal value. A certain task may not contribute as much value to the overall process as other tasks. Using a separate execution core to run that task may not be worth the cost.

3. **Data splitting**. Just as applications are divided into separate tasks, the data accessed and manipulated by the tasks must be divided to run on separate cores.

4. **Data dependency**. When one task depends on data from another, programmers must ensure that the execution of the tasks is synchronized to accommodate the data dependency.

5. **Testing and debugging**. When a program is running in parallel on multiple cores, many different execution paths are possible. Testing and debugging such concurrent programs is inherently more difficult than testing and debugging single-threaded applications.

Types of parallelism

- **Data parallelism** focuses on distributing subsets of the same data across multiple computing cores and performing the same operation on each core. Consider, for example, summing the contents of an array of size N. On a single-core system, one thread would simply sum the elements [0] . . . [N − 1]. On a dual-core system, however, thread A, running on core 0, could sum the elements [0] . . . [N/2 − 1] while thread B, running on core 1, could sum the elements [N/2] . . . [N − 1]. The two threads would be running in parallel on separate computing cores.
- **Task parallelism** involves distributing not data but tasks (threads) across multiple computing cores. Each thread is performing a unique operation. Different threads may be operating on the same data, or they may be operating on different data. An example of task parallelism might involve two threads, each performing a unique statistical operation on the array of elements. The threads again are operating in parallel on separate computing cores, but each is performing a unique operation.

Multithreading models

- Support for threads may be provided either at the user level, for **user threads**, or by the kernel, for **kernel threads**.
- User threads are supported above the kernel and are managed without kernel support, whereas kernel threads are supported and managed directly by the operating system.
- A relationship must exist between user threads and kernel threads. There are three common ways of establishing such a relationship: the **many-to-one** model, the **one-to-one** model, and the **many-to-many** model.

1. **Many to one model**

- The many-to-one model maps many user-level threads to one kernel thread.
- Thread management is done by the thread library in user space, so it is efficient
- However, the entire process will block if a thread makes a blocking system call.
- Also, because only one thread can access the kernel at a time, multiple threads are unable to run in parallel on multicore systems.

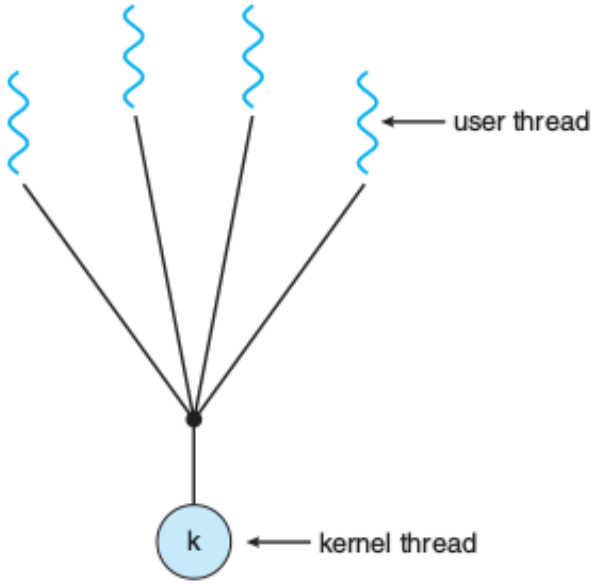


Figure 4.5 Many-to-one model.

2. **One to One model**

- The one-to-one model maps each user thread to a kernel thread.
- It provides more concurrency than the many-to-one model by allowing another thread to run when a thread makes a blocking system call.
- It also allows multiple threads to run in parallel on multiprocessors.
- The only drawback to this model is that creating a user thread requires creating the corresponding kernel thread. The overhead of creating kernel threads can burden the performance of an application

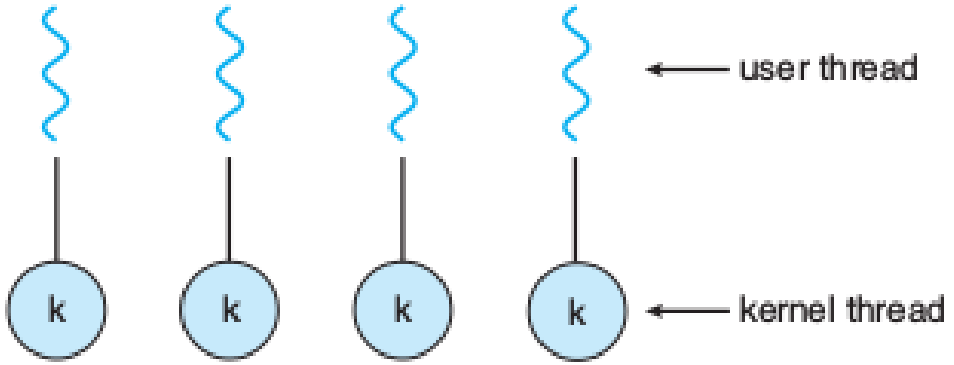


Figure 4.6 One-to-one model.

3. Many to Many model

- The many-to-many model multiplexes many user-level threads to a smaller or equal number of kernel threads.
- Developers can create as many user threads as necessary, and the corresponding kernel threads can run in parallel on a multiprocessor.
- Also, when a thread performs a blocking system call, the kernel can schedule another thread for execution.

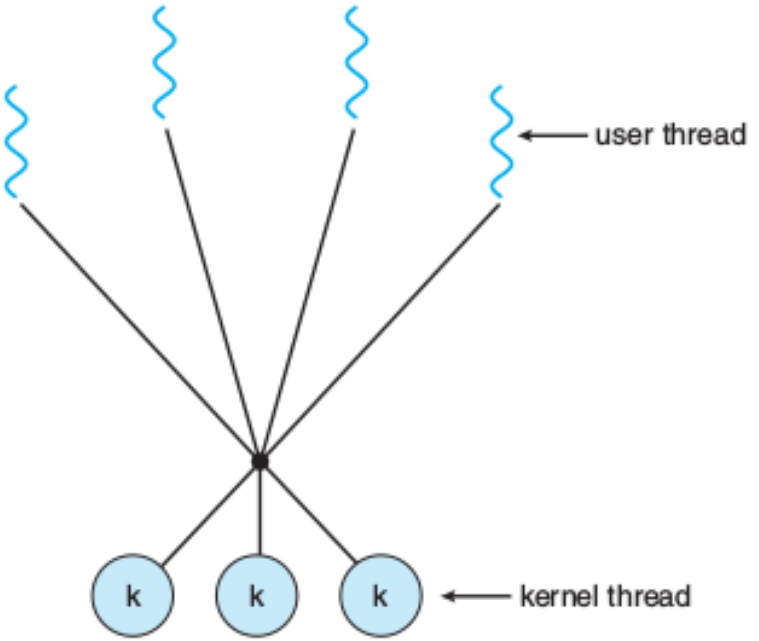


Figure 4.7 Many-to-many model.

- One variation on the many-to-many model still multiplexes many user-level threads to a smaller or equal number of kernel threads but also allows a user-level thread to be bound to a kernel thread. This variation is sometimes referred to as the **two-level model.**

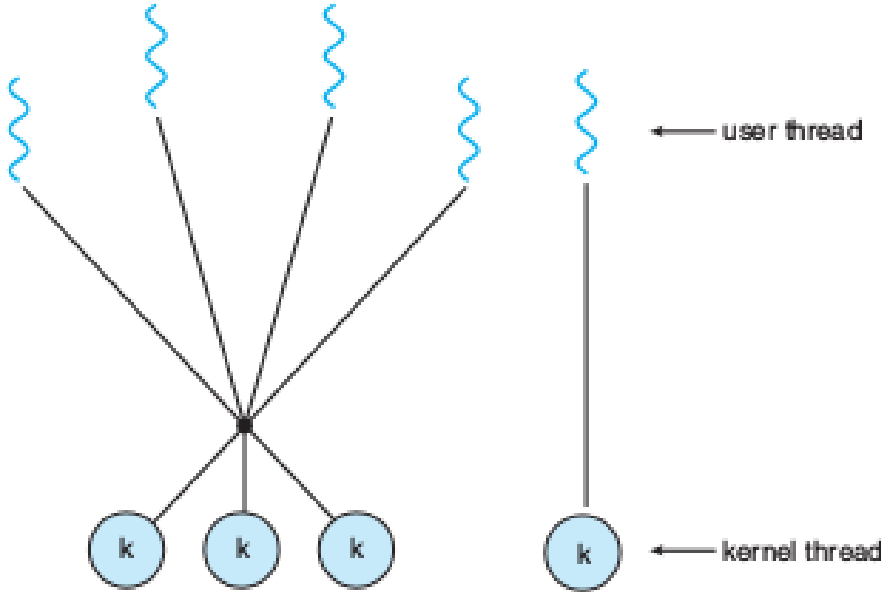


Figure 4.8 Two-level model.

Thread Libraries

- A thread library provides the programmer with an API for creating and managing threads. There are two primary ways of implementing a thread library.
- The first approach is to provide a library entirely in user space with no kernel support. All code and data structures for the library exist in user space. This means that invoking a function in the library results in a local function call in user space and not a system call.
- The second approach is to implement a kernel-level library supported directly by the operating system. In this case, code and data structures for the library exist in kernel space. Invoking a function in the API for the library typically results in a system call to the kernel.

- Three main thread libraries are in use today: POSIX (Portable OS Interface based on unix) Pthreads, Windows, and Java.
- The Windows thread library is a kernel-level library available on Windows systems. The Java thread API allows threads to be created and managed directly in Java programs.
- For POSIX and Windows threading, any data declared globally—that is, declared outside of any function—are shared among all threads belonging to the same process.
- Because Java has no notion of global data, access to shared data must be explicitly arranged between threads.
- Data declared local to a function are typically stored on the stack. Since each thread has its own stack, each thread has its own copy of local data.

- Two general strategies for creating multiple threads: **asynchronous threading** and **synchronous threading**.
- With **asynchronous threading**, once the parent creates a child thread, the parent resumes its execution, so that the parent and child execute concurrently.
- Each thread runs independently of every other thread, and the parent thread need not know when its child terminates.
- Because the threads are independent, there is typically little data sharing between threads.

- **Synchronous threading** occurs when the parent thread creates one or more children and then must wait for all of its children to terminate before it resumes —the so-called **fork-join** strategy.
- Here, the threads created by the parent perform work concurrently, but the parent cannot continue until this work has been completed.
- Once each thread has finished its work, it terminates and joins with its parent. Only after all of the children have joined can the parent resume execution.
- Typically, synchronous threading involves significant data sharing among threads.

Pthreads

- Pthreads refers to the POSIX standard (IEEE 1003.1c) defining an API for thread creation and synchronization.
- The following C program demonstrates the basic Pthreads API for constructing a multithreaded program that calculates the summation of a non-negative integer in a separate thread.
- In a Pthreads program, separate threads begin execution in a specified function. In the example, this is the runner() function.
- When this program begins, a single thread of control begins in main() . After some initialization, main() creates a second thread that begins control in the runner() function. Both threads share the global data sum .

```c
#include <pthread.h>
#include <stdio.h>

int sum; /* this data is shared by the thread(s) */
void *runner(void *param); /* threads call this function */

int main(int argc, char *argv[])
{
   pthread_t tid; /* the thread identifier */
   pthread_attr_t attr; /* set of thread attributes */

   if (argc != 2) {
      fprintf(stderr,"usage: a.out <integer value>\n");
      return -1;
   }
   if (atoi(argv[1]) < 0) {
      fprintf(stderr,"%d must be >= 0\n",atoi(argv[1]));
      return -1;
   }

   /* get the default attributes */
   pthread_attr_init(&attr);
   /* create the thread */
   pthread_create(&tid,&attr,runner,argv[1]);
   /* wait for the thread to exit */
   pthread_join(tid,NULL);

   printf("sum = %d\n",sum);
}

/* The thread will begin control in this function */
void *runner(void *param)
{
   int i, upper = atoi(param);
   sum = 0;

   for (i = 1; i <= upper; i++)
      sum += i;

   pthread_exit(0);
}
```

- Each thread has a set of attributes, including stack size and scheduling information. The pthread_attr_t attr declaration represents the attributes for the thread. We set the attributes in the function call pthread_attr_init(&attr).
- pthread_join() function takes two parameters –

  1. **thread-id** Is the thread to wait for.

  2. **status** Is the location where the exit status of the joined thread is stored. This can be set to NULL if the exit status is not required.

- A simple method for waiting on several threads using the pthread join() function is to enclose the operation within a simple for loop.

```
#define NUM_THREADS 10

/* an array of threads to be joined upon */
pthread_t workers[NUM_THREADS];

for (int i = 0; i < NUM_THREADS; i++)
   pthread_join(workers[i], NULL);
```

Figure 4.10 Pthread code for joining ten threads.

Process Synchronization

- One process may only partially complete execution before another process is scheduled.
- In fact, a process may be interrupted at any point in its instruction stream, and the processing core may be assigned to execute instructions of another process.
- **Parallel execution**, in which two instruction streams (representing different processes) execute simultaneously on separate processing cores.

Producer-Consumer problem

- Producer produces items in the buffer and consumer consumes them from the buffer
- Both are executing simultaneously
- They share the buffer

- One possibility is to add an integer variable counter , initialized to 0.
- counter is incremented every time we add a new item to the buffer and is decremented every time we remove one item from the buffer.
- **Producer**

```
while (true) {
    /* produce an item in next produced */
    while (counter == BUFFER SIZE ); /* do nothing */
    buffer[in] = next produced;
    in = (in + 1) % BUFFER SIZE ;
    counter++;
    }
```

Consumer

```
while (true) {
    while (counter == 0); /* do nothing */
    next consumed = buffer[out];
    out = (out + 1) % BUFFER SIZE ;
    counter--;
    /* consume the item in next consumed */
}
```

- Suppose that the value of the variable counter is currently 5 and that the producer and consumer processes concurrently execute the statements “ counter++ ” and “ counter-- ”.

- The statement “ counter++ ” may be implemented in machine language as follows:

  register1 = counter

  register1 = register1 + 1

  counter = register1

  where register1 is one of the local CPU registers. Similarly, the statement “ counter-- ” is implemented as follows:

  register2 = counter

  register2 = register2 − 1

  counter = register2

  where again register2 is one of the local CPU registers.

- The concurrent execution of “ counter++ ” and “ counter-- ” is equivalent to a sequential execution. One possible interleaving is the following:

- T 0 : producer execute register1 = counter {register1 = 5}
- T 1 : producer execute register1 = register1 + 1 {register1 = 6}
- T 2 : consumer execute register2 = counter {register2 = 5}
- T 3 : consumer execute register2 = register2 − 1 {register2 = 4}
- T 4 : producer execute counter = register1 {counter = 6}
- T 5 : consumer execute counter = register2 {counter = 4}
- We have arrived at the incorrect state “ counter == 4”, indicating that four buffers are full, when, in fact, five buffers are full.
- If we reversed the order of the statements at T 4 and T 5 , we would arrive at the incorrect state “ counter == 6”.
- This reaches incorrect state because we allowed both processes to manipulate the variable counter concurrently.
- When several processes access and manipulate the same data concurrently and the outcome of the execution depends on the particular order in which the access takes place, is called a **race condition**.
- To guard against the race condition above, we need to ensure that only one process at a time can be manipulating the variable counter .

The Critical-Section Problem

- Consider a system consisting of n processes {P0 , P1 , ..., Pn−1 }. Each process has a segment of code, called a **critical section**, in which the process may be changing common variables, updating a table, writing a file, and so on.
- When one process is executing in its critical section, no other process is allowed to execute in its critical section. That is, no two processes are executing in their critical sections at the same time.
- Each process must request permission to enter its critical section.
- The section of code implementing this request is the **entry section**. The critical section may be followed by an **exit section**. The remaining code is the **remainder section**.

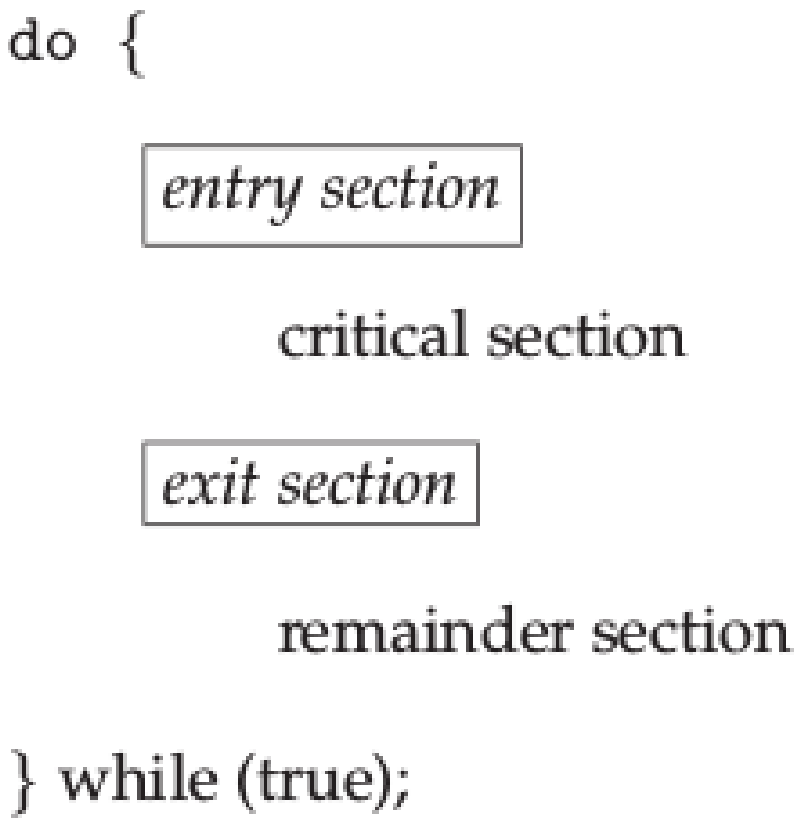


Figure 5.1 General structure of a typical process P_i.

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

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

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

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

- Two general approaches are used to handle critical sections in operating systems: **preemptive kernels and nonpreemptive kernels**.
- A preemptive kernel allows a process to be preempted while it is running in kernel mode.
- A nonpreemptive kernel does not allow a process running in kernel mode to be preempted; a kernel-mode process will run until it exits kernel mode, blocks, or voluntarily yields control of the CPU.

Peterson’s Solution

- Peterson’s solution is restricted to two processes that alternate execution between their critical sections and remainder sections. The processes are numbered P0 and P1 .
- Peterson’s solution requires the two processes to share two data items:

  int turn;

  boolean flag[2];
- The variable turn indicates whose turn it is to enter its critical section. That is, if turn == i , then process Pi is allowed to execute in its critical section.
- The flag array is used to indicate if a process is ready to enter its critical section. For example, if flag[i] is true , this value indicates that Pi is ready to enter its critical section.

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

        critical section

    flag[i] = false;

        remainder section

} while (true);
```

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

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

- We need to show that:

  1. Mutual exclusion is preserved.

- Each Pi enters its critical section only if either flag[j] == false or turn == i. flag[0] == flag[1] == true is possible at the same time but the value of turn can be either 0 or 1.
- Hence, one of the processes —say, Pj —must have successfully executed the while statement, whereas Pi had to execute at least one additional statement (“ turn == j ”).
- However, at that time, flag[j] == true and turn == j , and this condition will persist as long as Pj is in its critical section;

  2. The progress requirement is satisfied.

  3. The bounded-waiting requirement is met.

- A process Pi can be prevented from entering the critical section only if it is stuck in the while loop with the condition flag[j] == true and turn == j.
- If Pj is not ready to enter the critical section, then flag[j] == false , and Pi can enter its critical section.
- If Pj has set flag[j] to true and is also executing in its while statement, then either turn == i or turn == j . If turn == i , then Pi will enter the critical section. If turn == j , then Pj will enter the critical section.
- Let Pj enters. However, once Pj exits its critical section, it will reset flag[j] to false , allowing Pi to enter its critical section.
- If Pj resets flag[j] to true , it must also set turn to i . Thus, since Pi does not change the value of the variable turn while executing the while statement, Pi will enter the critical section (**progress**) after at most one entry by Pj (**bounded waiting**).

Semaphores

- A semaphore S is an integer variable that, apart from initialization, is accessed only through two standard atomic operations: wait() and signal() .
- wait(S) {

  while (S <= 0); // busy wait

  S--;

  }
- signal(S) {

  S++;

  }
- All modifications to the integer value of the semaphore in the wait() and signal() operations must be executed indivisibly. That is, when one process modifies the semaphore value, no other process can simultaneously modify that same semaphore value.
- In addition, in the case of wait(S) , the testing of the integer value of S ($S \leq 0$), as well as its possible modification (S--), must be executed without interruption.

- The value of a **counting semaphore** can range over an unrestricted domain.
- The value of a **binary semaphore** can range only between 0 and 1.
- Counting semaphores can be used to control access to a given resource consisting of a finite number of instances.
- The semaphore is initialized to the number of resources available. Each process that wishes to use a resource performs a wait() operation on the semaphore (thereby decrementing the count).
- When a process releases a resource, it performs a signal() operation (incrementing the count).
- When the count for the semaphore goes to 0, all resources are being used. After that, processes that wish to use a resource will block until the count becomes greater than 0.

- Consider two concurrently running processes: P1 with a statement S1 and P2 with a statement S2 .
- Suppose we require that S2 be executed only after S1 has completed. We can implement this scheme readily by letting P1 and P2 share a common semaphore synch , initialized to 0.
- In process P1 , we insert the statements

  S1 ;

  signal(synch);
- In process P 2 , we insert the statements

  wait(synch);

  S2 ;
- Because synch is initialized to 0, P 2 will execute S 2 only after P 1 has invoked signal(synch) , which is after statement S 1 has been executed.

CPU Scheduling

- With multiprogramming, several processes are kept in memory at one time.
- When one process has to wait, the operating system takes the CPU away from that process and gives the CPU to another process. This pattern continues.
- Every time one process has to wait, another process can take over use of the CPU .
- Process execution consists of a cycle of CPU execution and I/O wait. Processes alternate between these two states.
- Process execution begins with a **CPU burst**. That is followed by an **I/O burst**, which is followed by another CPU burst, then another I/O burst, and so on.
- Eventually, the final CPU burst ends with a system request to terminate execution

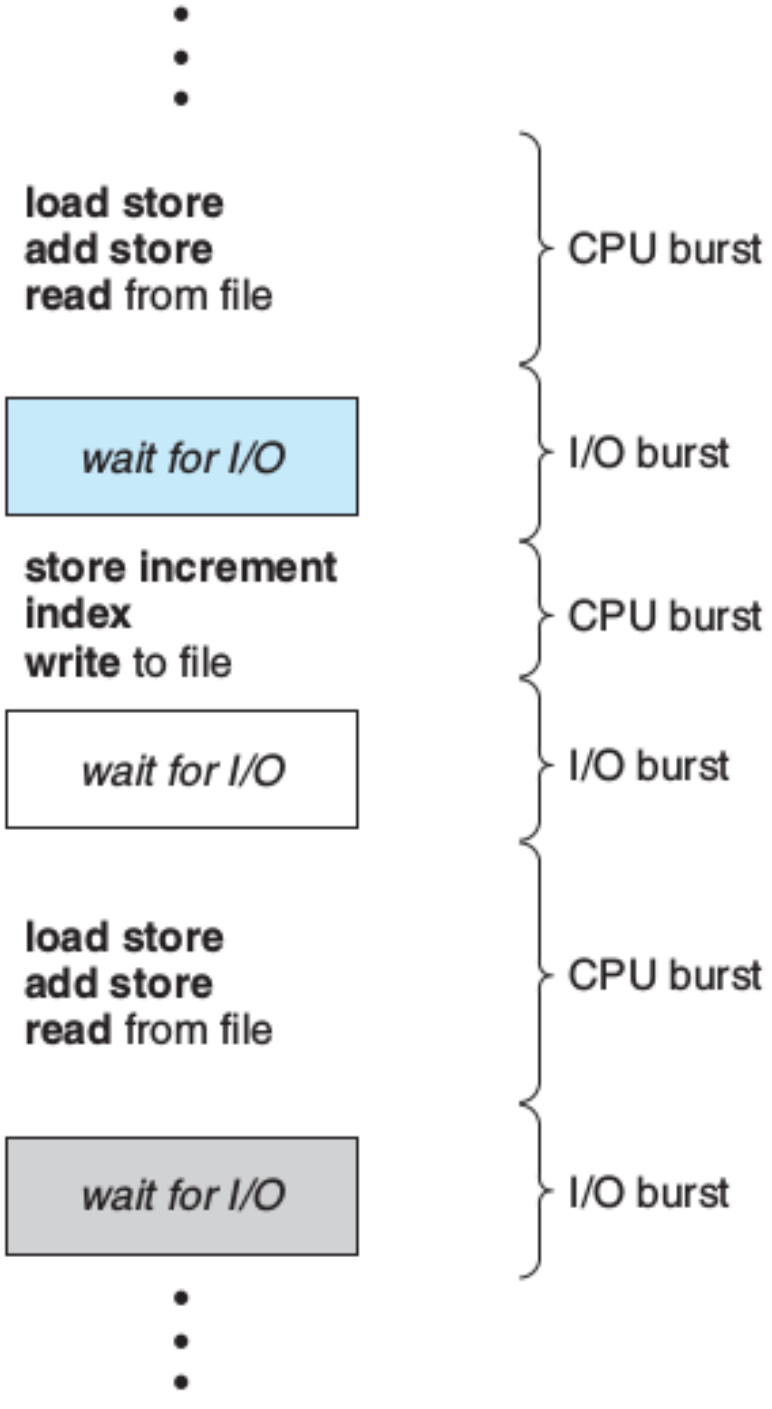


Figure 6.1 Alternating sequence of CPU and I/O bursts.

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

Pre-emptive Scheduling

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

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

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

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

  4. When a process terminates

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

- When scheduling takes place only under circumstances 1 and 4, we say that the scheduling scheme is **nonpreemptive** or **cooperative**.
- Otherwise, it is **preemptive**.
- Under nonpreemptive scheduling, once the CPU has been allocated to a process, the process keeps the CPU until it releases the CPU either by terminating or by switching to the waiting state.
- Unfortunately, preemptive scheduling can result in race conditions when data are shared among several processes.
- Consider the case of two processes that share data. While one process is updating the data, it is preempted so that the second process can run.
- The second process then tries to read the data, which are in an inconsistent state.

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

  1. Switching context

  2. Switching to user mode

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

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

Scheduling Criteria

- **CPU utilization**. We want to keep the CPU as busy as possible.
- **Throughput**. One measure of work is the number of processes that are completed per time unit, called throughput. For long processes, this rate may be one process per hour; for short transactions, it may be ten processes per second.
- **Turnaround time**. The interval from the time of submission of a process to the time of completion is the turnaround time. Turnaround time is the sum of the periods spent waiting to get into memory, waiting in the ready queue, executing on the CPU , and doing I/O .
- **Waiting time**. Waiting time is the sum of the periods spent waiting in the ready queue.
- **Response time**. In an interactive system, a process can produce some output fairly early and can continue computing new results while previous results are being output to the user. Thus, another measure is the time from the submission of a request until the first response is produced.

- It is desirable to maximize CPU utilization and throughput and to minimize turnaround time, waiting time, and response time.

Scheduling Algorithms

1. First-Come, First-Served Scheduling

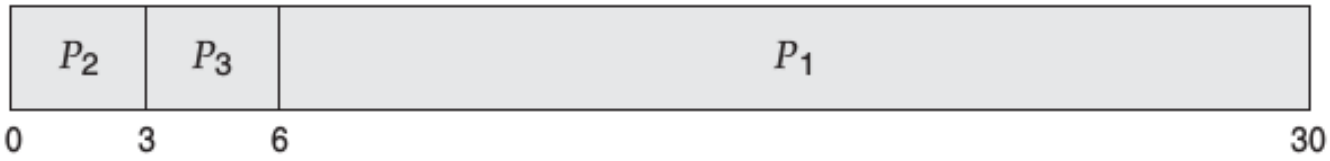
- With this scheme, the process that requests the CPU first is allocated the CPU first.
- The implementation of the FCFS policy is easily managed with a FIFO queue.
- When a process enters the ready queue, its PCB is linked onto the tail of the queue.
- When the CPU is free, it is allocated to the process at the head of the queue. The running process is then removed from the queue.

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

Process	Burst Time
P_1	24
P_2	3
P_3	3

- If the processes arrive in the order P1 , P2 , P3 , and are served in FCFS order, we get the result shown in the following **Gantt chart**, which is a bar chart that illustrates a particular schedule, including the start and f
proce

- The waiting time is 0 milliseconds for process P1 , 24 milliseconds for process P2 , and 27 milliseconds for process P3 .
- Thus, the average waiting time is (0 + 24 + 27)/3 = 17 milliseconds. If the processes arrive in the order P2 , P3 , P1



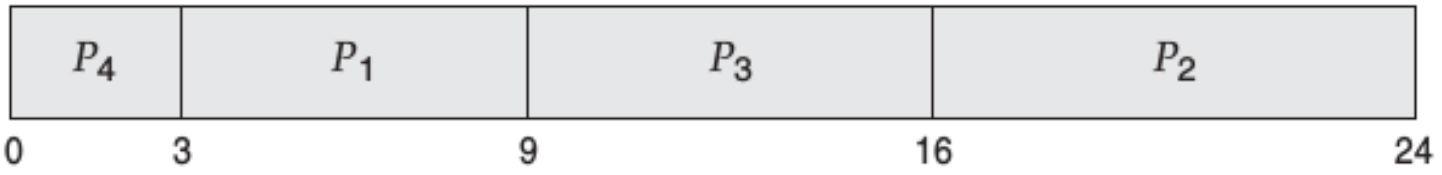
- The average waiting time is now (6 + 0 + 3)/3 = 3 milliseconds.
- The FCFS scheduling algorithm is nonpreemptive. Once the CPU has been allocated to a process, that process keeps the CPU until it releases the CPU , either by terminating or by requesting I/O .

- Assume we have one CPU -bound process and many I/O -bound processes. The CPU -bound process will get and hold the CPU . During this time, all the other processes will finish their I/O and will move into the ready queue, waiting for the CPU .
- While the processes wait in the ready queue, the I/O devices are idle.
- Eventually, the CPU -bound process finishes its CPU burst and moves to an I/O device. All the I/O -bound processes, which have short CPU bursts, execute quickly and move back to the I/O queues.
- At this point the CPU sits idle.
- The CPU -bound process will then move back to the ready queue and be allocated the CPU . Again, all the I/O processes end up waiting in the ready queue until the CPU -bound process is done.
- There is a **convoy effect** as all the other processes wait for the one big process to get off the CPU .
- This effect results in lower CPU and device utilization than might be possible if the shorter processes were allowed to go first.

2. Shortest-Job-First Scheduling

- When the CPU is available, it is assigned to the process that has the smallest next CPU burst.
- If the next CPU bursts of two processes are the same, FCFS scheduling is used to break the tie.
- It is also called **shortest-next-CPU -burst** algorithm, because scheduling depends on the length of the next CPU burst of a process, rather than its total length.

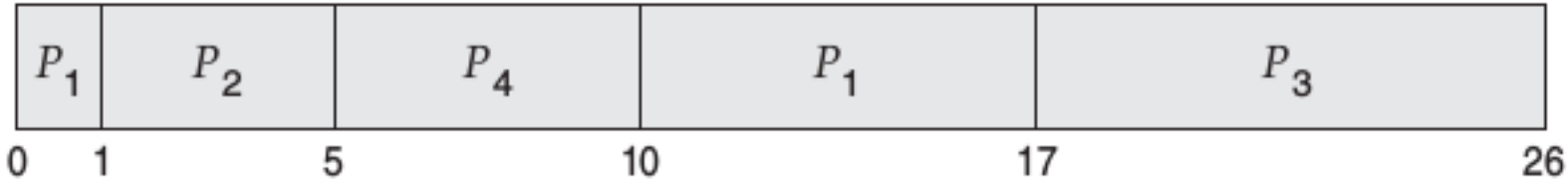
Process	Burst Time
P_1	6
P_2	8
P_3	7
P_4	3



- The waiting time is 3 milliseconds for process P1 , 16 milliseconds for process P2 , 9 milliseconds for process P3 , and 0 milliseconds for process P4 .
- Thus, the average waiting time is (3 + 16 + 9 + 0)/4 = 7 milliseconds.
- By comparison, if we were using the FCFS scheduling scheme, the average waiting time would be 10.25 milliseconds.
- Moving a short process before a long one decreases the waiting time of the short process more than it increases the waiting time of the long process. Consequently, the average waiting time decreases.
- The real difficulty with the SJF algorithm is knowing the length of the next CPU request.

- The SJF algorithm can be either preemptive or nonpreemptive.
- When a new process arrives at the ready queue while a previous process is still executing, the next CPU burst of the newly arrived process may be shorter than what is left of the currently executing process.
- A preemptive SJF algorithm will preempt the currently executing process, whereas a nonpreemptive SJF algorithm will allow the currently running process to finish its CPU burst.
- Preemptive SJF scheduling is sometimes called **shortest-remaining-time-first** scheduling.

Process	Arrival Time	Burst Time
P_1	0	8
P_2	1	4
P_3	2	9
P_4	3	5

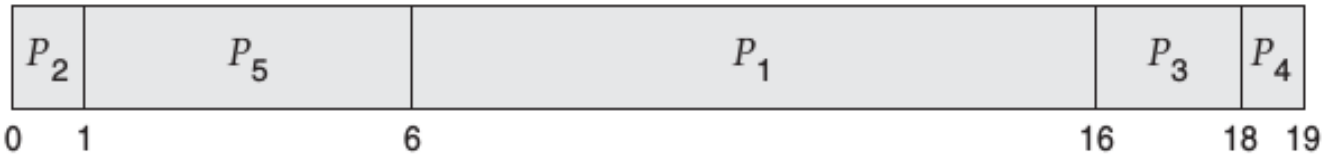


- Process P1 is started at time 0, since it is the only process in the queue. Process P2 arrives at time 1.
- The remaining time for process P1 (7 milliseconds) is larger than the time required by process P2 (4 milliseconds), so process P1 is preempted, and process P2 is scheduled.
- The average waiting time for this example is [(10 − 1) + (1 − 1) + (17 − 2) + (5 − 3)]/4 = 26/4 = 6.5 milliseconds.
- Nonpreemptive SJF scheduling would result in an average waiting time of 7.75 milliseconds.

3. Priority Scheduling

- A priority is associated with each process, and the CPU is allocated to the process with the highest priority.
- Equal-priority processes are scheduled in FCFS order.
- An SJF algorithm is simply a priority algorithm where the priority (p) is the inverse of the (predicted) next CPU burst. The larger the CPU burst, the lower the priority, and vice versa.
- For the following example, the average waiting time is 8.2 milliseconds.

Process	Burst Time	Priority
P_1	10	3
P_2	1	1
P_3	2	4
P_4	1	5
P_5	5	2



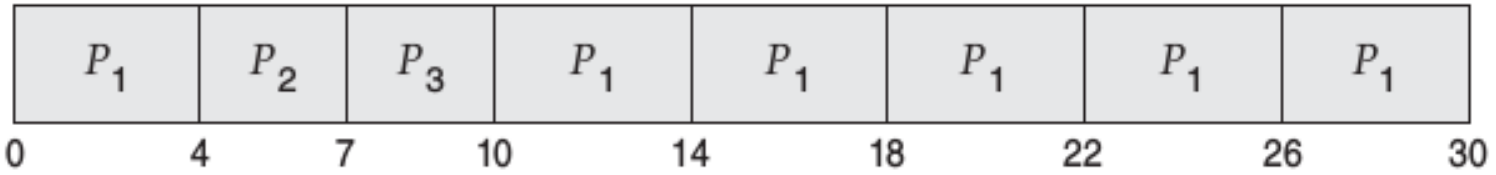
- Priorities can be defined either internally or externally.
- Internally defined priorities are time limits, memory requirements, the number of open files, and the ratio of average I/O burst to average CPU burst have been used in computing priorities.
- External priorities are set by criteria outside the operating system, such as the importance of the process, the type and amount of funds being paid for computer use, the department sponsoring the work, and other, often political, factors.
- Priority scheduling can be either preemptive or nonpreemptive. When a process arrives at the ready queue, its priority is compared with the priority of the currently running process.
- A preemptive priority scheduling algorithm will preempt the CPU if the priority of the newly arrived process is higher than the priority of the currently running process.
- A nonpreemptive priority scheduling algorithm will simply put the new process at the head of the ready queue.

- A major problem with priority scheduling algorithms is **indefinite blocking, or starvation**.
- A priority scheduling algorithm can leave some low- priority processes waiting indefinitely.
- In a heavily loaded computer system, a steady stream of higher-priority processes can prevent a low-priority process from ever getting the CPU .
- A solution to the problem of indefinite blockage of low-priority processes is **aging**.
- Aging involves gradually increasing the priority of processes that wait in the system for a long time.
- For example, if priorities range from 127 (low) to 0 (high), we could increase the priority of a waiting process by 1 every 15 minutes.
- Eventually, even a process with an initial priority of 127 would have the highest priority in the system and would be executed.

4. Round-Robin Scheduling

- The round-robin (RR) scheduling algorithm is designed especially for time-sharing systems.
- It is similar to FCFS scheduling, but preemption is added to enable the system to switch between processes. A small unit of time, called a **time quantum** or **time slice**, is defined.
- The ready queue is treated as a circular queue.
- One of two things will then happen. The process may have a CPU burst of less than 1 time quantum. In this case, the process itself will release the CPU voluntarily. The scheduler will then proceed to the next process in the ready queue.
- If the CPU burst of the currently running process is longer than 1 time quantum, the timer will go off and will cause an interrupt to the operating system. A context switch will be executed, and the process will be put at the tail of the ready queue. The CPU scheduler will then select the next process in the ready queue.
- The average waiting time under the RR policy is often long.

Process	Burst Time
P_1	24
P_2	3
P_3	3



- If we use a time quantum of 4 milliseconds, then process P1 gets the first 4 milliseconds. Since it requires another 20 milliseconds, it is preempted after the first time quantum, and the CPU is given to the next process in the queue, process P2 . The RR scheduling algorithm is thus preemptive.
- Process P2 does not need 4 milliseconds, so it quits before its time quantum expires. The CPU is then given to the next process, process P3 .
- Once each process has received 1 time quantum, the CPU is returned to process P1 for an additional time quantum.

- P1 waits for 6 milliseconds (10 - 4), P2 waits for 4 milliseconds, and P3 waits for 7 milliseconds.
- Thus, the average waiting time is 17/3 = 5.66 milliseconds.
- If there are n processes in the ready queue and the time quantum is q, then each process gets 1/n of the CPU time in chunks of at most q time units.
- Each process must wait no longer than (n − 1) × q time units until its next time quantum.
- For example, with five processes and a time quantum of 20 milliseconds, each process will get up to 20 milliseconds every 100 milliseconds.

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

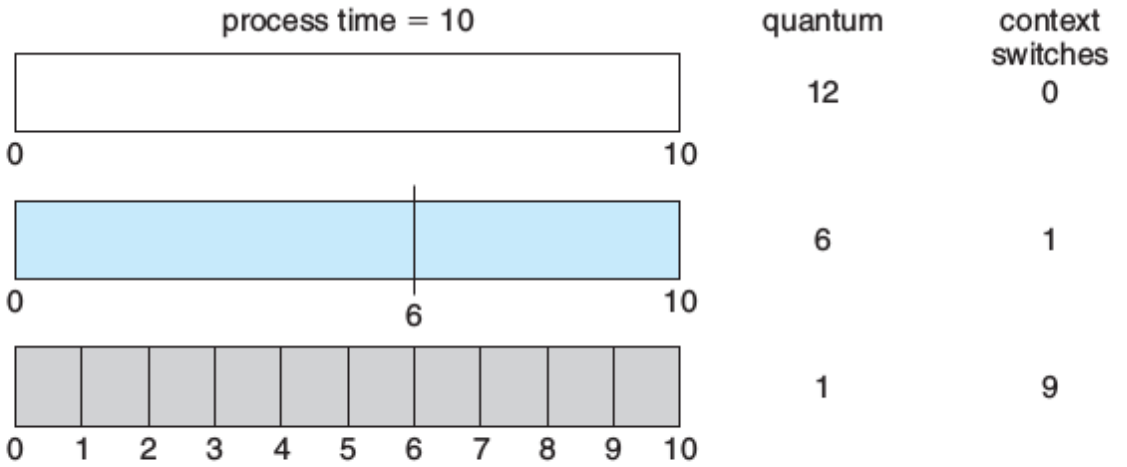


Figure 6.4 How a smaller time quantum increases context switches.