Operating Systems

Chapter 1

What is an operating system?

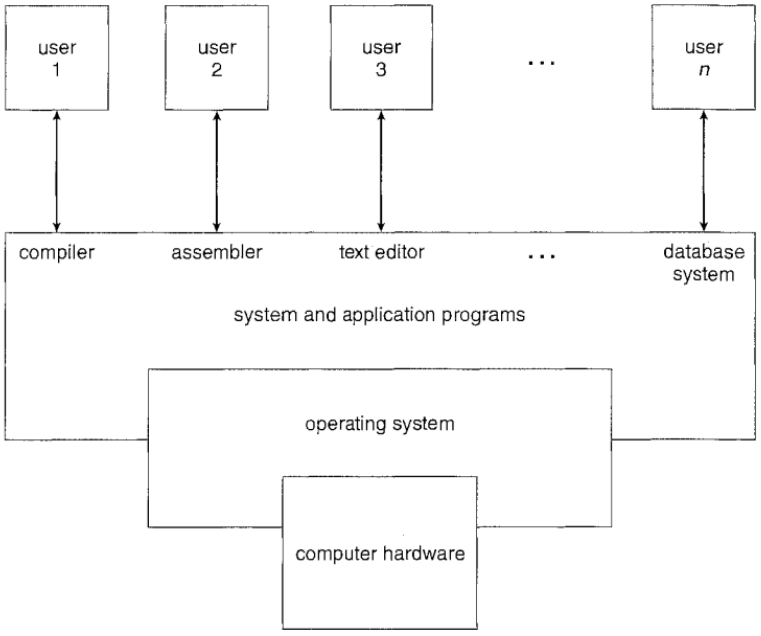
An operating system acts as an intermediary between the user of a computer and the computer hardware.

Mainframe operating systems are designed primarily to optimize utilization of hardware. Personal computer (PC) operating systems support complex games, business applications, and everything in between.

Purpose of an operating system

The purpose of an operating system is to provide an environment in which a user can execute programs in a convenient and efficient manner.

Components of an operating system



A computer system has roughly four components:

- hardware
- operating system
- application programs
- users

The operating system provides the means for proper use of these resources in the operation of the computer system.

Personal computers designed for **ease of use.**

We can view the operating system as a **resource allocator.**

An operating system is 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.

A simple viewpoint is that the operating system includes everything a vendor ships when you order "the operating system." The features included, however, vary greatly across systems.

A more common definition, and the one that we usually follow, is that the operating system is the one program running at all times on the computer-usually called the **kernel**. (Along with the kernel, there are two other types of programs: system programs, which are associated with the operating system but are not part of the kernel, and application programs, which include all programs not associated with the operation of the system.)

Computer System Organization

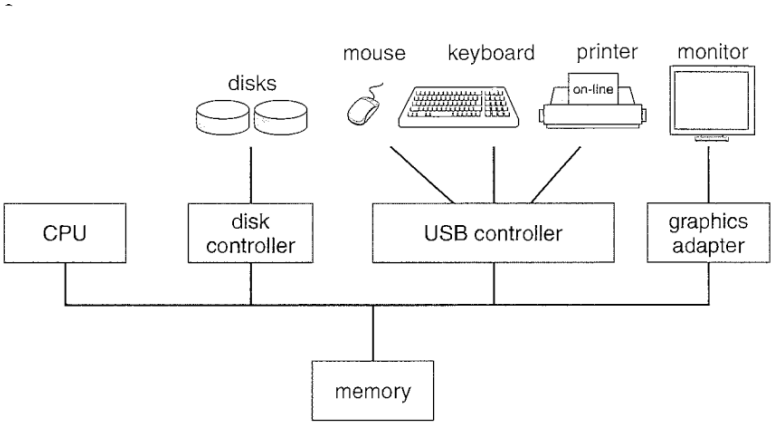


Figure 1.2 A modern computer system.

For a computer to start running-for instance, when it is powered up or rebooted-it needs to have an initial program to run. This initial program, or **bootstrap program**, tends to be simple. Typically, it is stored in read-only memory **ROM** or electrically erasable programmable read-only memory **EEPROM** known by the general term **firmware** within the computer hardware.

The **bootstrap program** must know how to load the operating system and how to start executing that system. To accomplish this goal, the bootstrap program must **locate** and **load into memory** the operating system **kernel**. The operating system then starts executing the first process, such as "init," and waits for some event to occur.

The occurrence of an event is usually signaled by an **interrupt** from either the hardware or the software.

Hardware may trigger an **interrupt** at any time by sending a signal to the CPU, usually by way of the system bus. Software may trigger an interrupt by executing a special operation called a **system call** (also called a **monitor call**).

When the CPU is interrupted, it stops what it is doing and **immediately transfers execution to a fixed location**. The fixed location usually contains the **starting address** where the service routine for the interrupt is located. The interrupt service routine executes; **on completion, the CPU resumes the interrupted computation**.

The interrupt must transfer control to the **appropriate interrupt service routine**. The straightforward method for handling this transfer would be to invoke a **generic routine** to **examine the interrupt information**; the routine, in turn, **would call the interrupt-specific handler**.

However, **interrupts must be handled quickly**. Since only a predefined number of interrupts is possible, **a table of pointers to interrupt routines can be used** instead to provide the necessary speed.

The **interrupt routine** is called indirectly through the table, with no intermediate routine needed. Generally, the table of pointers is **stored in low memory** (the first hundred or so locations). These locations **hold the addresses of the interrupt service routines** for the various devices.

This array, or **interrupt vector**, of addresses is then indexed by a unique device number, given with the interrupt request, to provide the address of the interrupt service routine for the interrupting device. Operating systems as different as Windows and UNIX dispatch interrupts in this manner.

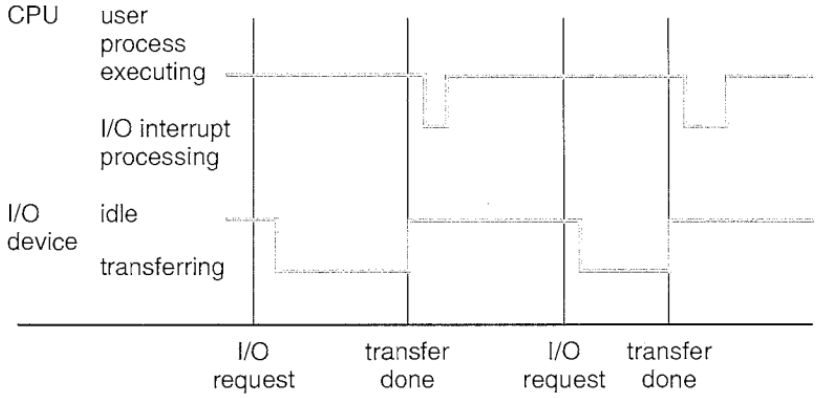


Figure 1.3 Interrupt time line for a single process doing output.

The interrupt architecture must also save the **address of the interrupted instruction**.

If the interrupt routine needs to **modify the processor state-for instance**, by modifying register values, **it must explicitly save the current state and then restore that state before returning**.

After the interrupt is serviced, **the saved return address is loaded into the program counter**, and the interrupted computation resumes as though the interrupt had not occurred.

General-purpose computers run most of their programs from **rewritable memory**, called main memory (also called **random access memory**, **RAM**). Main memory commonly is implemented in a semiconductor technology called **dynamic random access memory (DRAM)**. Because ROM cannot be changed. EEPROM cannot be changed frequently that's why mostly static programs are here.

Accordingly, **we can ignore how a memory address is generated by a program**. **We are interested only in the sequence of memory addresses generated** by the running program.

A second storage is used because:

- Main memory is usually too small to store all needed programs and data permanently.
- Main memory is a volatile storage device that loses its contents when power is turned off or otherwise lost.

The wide variety of storage systems in a computer system can be organized in a hierarchy according to speed and cost. **The higher levels are expensive, but they are fast. As we move down the hierarchy, the cost per bit generally decreases, whereas the access time generally increases**.

In addition to differing in speed and cost, the various storage systems are either volatile or nonvolatile. As mentioned earlier, **volatile storage** loses its contents when the power to the device is removed. (volatile -> RAM, ROM etc.)

In the absence of expensive battery and generator backup systems, data must be written to **nonvolatile storage** for safekeeping.

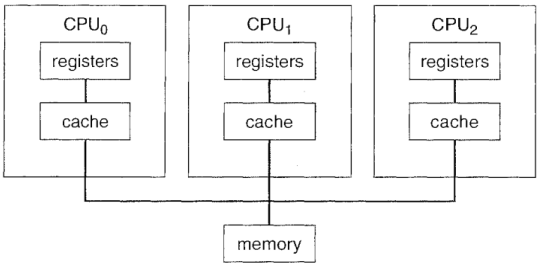


Figure 1.6 Symmetric multiprocessing architecture.

The benefit of this model is that **many processes can run simultaneously -N processes can run if there are N CPUs-without causing a significant deterioration of performance**. However, we must carefully control I/0 to ensure that the data reaches the appropriate processor.

Also, since the CPUs are separate, one may be sitting idle while another is overloaded, **resulting in inefficiencies**. These inefficiencies **can be avoided if the processors share certain data structures**.

Multiprocessing adds CPUs to increase computing power. If the CPU has an integrated memory controller, then adding CPUs can also **increase the amount of memory addressable in the system**.

Either way, multiprocessing can cause a system to change its memory access model from **uniform memory access** (UMA) to **non-uniform memory access** (NUMA). **UMA** is defined as the situation in which access to any RAM from any CPU takes the same amount of time. With **NUMA**, some parts of memory may take longer to access than other parts, creating a performance penalty. Operating systems can minimize the NUMA penalty through resource management.

A recent trend in CPU design is to **include multiple computing cores on a single chip**. They can be more efficient than multiple chips with single cores **because on-chip communication is faster than between-chip communication**. In addition, **one chip with multiple cores uses significantly less power than multiple single-core chips**. As a result, multicore systems are especially well suited for server systems such as database and Web servers.

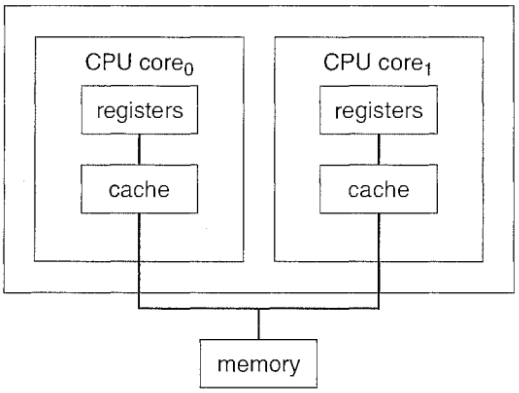


Figure 1.7 A dual-core design with two cores placed on the same chip.

In this design, each core has its own register set as well as its own local cache; other designs might use a shared cache or a combination of local and shared caches.

Aside from architectural considerations, such as cache, memory, and bus contention, these multicore CPUs appear to the operating system as N standard processors. This tendency puts pressure on operating system designers-and application programmers-to make use of those CPUs.

Finally, **blade servers** are a recent development in which multiple processor boards, I/0 boards, and networking boards are placed in the same chassis. The difference between these and traditional multiprocessor systems is that **each blade-processor board boots independently and runs its own operating system**.

Clustered Systems

Another type of multiple-CPU system is the **clustered system**. Like multiprocessor systems, clustered systems gather together multiple CPUs to accomplish computational work. Clustered systems differ from multiprocessor systems, however, in that they are composed of two or more individual systems-or nodes-joined together. The generally accepted definition is that clustered computers share storage and are closely linked via a **local area network (LAN)** or a faster interconnect, such as InfiniBand.

Clustering is usually used to provide **high availability** service; that is, service will continue even if one or more systems in the cluster fail. High availability is generally obtained by adding a level of redundancy in the system. A layer of cluster software runs on the cluster nodes. Each node can monitor one or more of the others (over the LAN). If the monitored machine fails, the monitoring machine can take ownership of its storage and restart the applications that were running on the failed machine. The users and clients of the applications see only a brief interruption of service.

Beowulf Clusters

Beowulf clusters are designed for solving high-performance computing tasks. These clusters are built using commodity hardware such as. personal computers-that are connected via a simple local area network. Beowulf clusters consist of a set of open-source software libraries that allow the computing nodes in the cluster to communicate with one another. In fact, some Beowulf clusters built from collections of discarded personal computers are using hundreds of computing nodes to solve computationally expensive problems in scientific computing.

In asymmetrical 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 symmetrical clustering, two or more hosts are running applications and are monitoring each other. **This mode is obviously more efficient, as it uses all of the available hardware**. **It does require that more than one application be available to run**.

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 are capable of running an application concurrently on all computers in the cluster**. However, applications must be written to take advantage of the cluster by using a technique known as **parallelization** which **consists of dividing a program into separate components that run in parallel on individual computers in the cluster**. 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**.

Other forms of clusters include parallel clusters and clustering over a **wide-area network (WAN)**. **Parallel clusters allow multiple hosts to access the same data on the shared storage**. Because most operating systems lack support for simultaneous data access by multiple hosts, parallel clusters are usually accomplished by use of special versions of software and special releases of applications. Each machine has full access to all data in the database. To provide this shared access to data, the system must also supply access control and locking to ensure that no conflicting operations occur. This function, commonly known as a **distributed lock machine (DLM)** is included in some cluster technology.

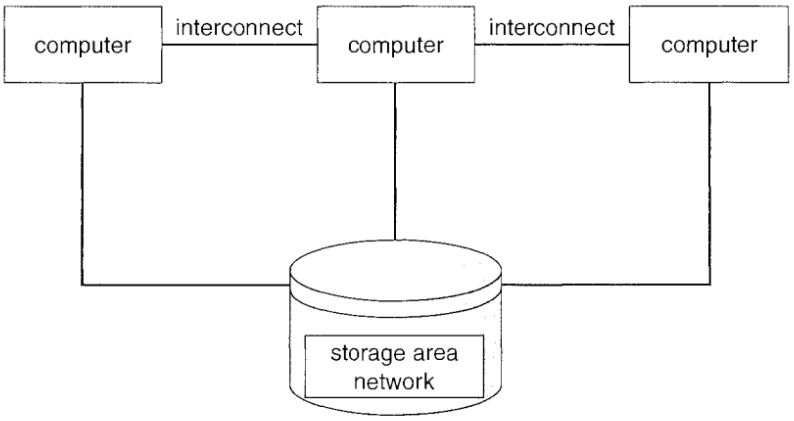


Figure 1.8 General structure of a clustered system.

Operating System Structure

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

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

The set of jobs in memory can be a subset of the jobs kept in the job pool. **The operating system picks and begins to execute one of the jobs in the memory**. Eventually, the job may have to wait for some task, such as an I/O operation to complete.

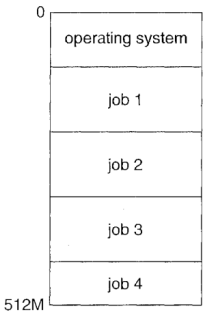


Figure 1.9 Memory layout for a multiprogramming system.

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 is switched to another job, and so on. Eventually the first job finishes waiting and gets the CPU back. As long as at least one job needs to be executed, 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** (**or hands-on**) computer system, **which provides direct communication between the user and the system**. The user gives instructions to the operating system or to a program directly, using an input device such as a keyboard or a mouse and waits for immediate results on an output device. Accordingly, the response time should be short - typically less than one second.

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

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 seperate program in memory**. A program loaded into memory and executed is called a **process**. When a process executes, it typically executes for only a short time it either finishes or needs to perform I/0. I/0 may be interactive; that is, output goes to a display for the user, and input comes from a user keyboard, mouse, or other device. Since interactive I/0 typically runs at "people speeds," it may take a long time to complete. Input, for example, may be bounded by the user's typing speed; seven characters per second is fast for people but incredibly slow for computers. Rather than let the CPU sit idle as this interactive input takes place, the operating system will rapidly switch the CPU to 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 is **job scheduling**. If several jobs are ready to run at the same time, making this decision is **CPU scheduling**.

In a time-sharing system, the operating system must ensure reasonable response time, which is sometimes accomplished through **swapping** where processes are swapped in and out of main memory to the disk. A more common method for achieving this goal is **virtual memory,** a technique that allows the execution of a process that is not completely in memory. The main advantage of the virtual-memory scheme is that it enables users to run programs that are larger than actual physical memory.

Operating System Operations

Modern operating systems are **interrupt driven**.

Events are almost always signaled by the occurrence of an **interrupt** or a **trap**.

A trap (or exception) is a software generated interrupt either by an error or by a specific request from a user program that an operating system service be performed.

The interrupt-driven nature of an operating system defines that system's general structure. For each type of interrupt, separate segments of code in the operating system determine what action should be taken. An interrupt service routine is provided that is responsible for dealing with the interrupt.

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 modes: **user** and **kernel** (or supervisor mode, system mode and privileged mode). A bit called **mode bit** is added to hardware to indicate the current mode: **kernel (0) or user (1)**. With mode bit, we are able to distinguish between a task for os and user. However, when a user application requests a service from the operating system (via a system call), **it must transition from user to kernel mode** to fulfill the request.

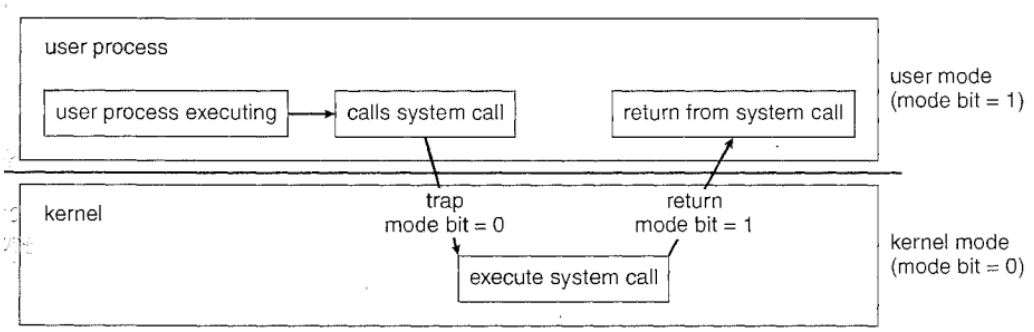


Figure 1.10 Transition from user to kernel mode.

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

The hardware allows privileged instructions to be executed only in kernel mode. Control is switched back to the operating system via an **interrupt, a trap, or a system call**.

system call: operating system will execute a task on the user program's behalf. A system call usually takes the form of a trap to a **specific location in the interrupt vector.** This trap can be executed by a **generic trap instruction**. When a system call occurs, it is treated as a software interrupt. **Control passes through the interrupt vector to service routine** in the operating system, **and the mode bit is set to kernel**. Kernel verifies the parameters, executes the request and returns control to the instruction following the system call.

Timer Operation

We must have control over the CPU. We cannot let a user program get stuck in a loop or fail to call system services and never return the control back to the operating system. To accomplish this goal we can use a **timer**. A timer can be set to interrupt the computer after a period. The counter will be decreased each time the clock ticks, and when it reaches zero an interrupt occurs.

Process Management

A program does nothing unless its instructions are executed by a CPU. **A program in execution, as mentioned, is a process**. A process **needs certain resources**, including CPU time, memory, files and I/O devices to accomplish its task. We emphasize that a **program by itself is not a process**; a **program is a passive entity**, like contents stored in a disk, where the **process is an active entity**. A single threaded process **has one program counter** specifying the next instruction to execute. The execution of such a process must be sequential. Further, at any time, one instruction at most is executed on behalf of the process. Thus, although two processes may be associated with the same program, they are nevertheless considered two separate execution sequences.

A process is the unit of work in a system. Such a system consists of a collection of processes, some of which are operating-system processes (those that execute system code) and the rest of which are user processes (those that execute user code). All these processes can potentially execute concurrently by multiplexing on a single CPU.

The operating system is responsible for the following activities in connection with process management:

- Scheduling processes and threads on the CPUs.
- Creating and deleting both user and system processes.
- Suspending and resuming processes.
- Providing mechanisms for process synchronization.
- Providing mechanisms for process communication.

Memory Management

Main memory is a repository of quickly accessible data shared by the CPU and I/0 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. As noted earlier, 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, that data must first be transferred to main n"lemory by CPU-generated I/0 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. We must take into account many factors-especially the hardware design of the system. Each algorithm requires its own hardware support.

The operating system is responsible for the following activities in connection with memory management:

- Keeping track of which parts of memory are currently being used and by whom.
- Deciding which processes (or parts thereof) and data to move into and out of memory.
- Allocating and deallocating memory space as needed.

Storage Management

A file is a collection of related information defined by its creator. Commonly, files represent programs (both source and object forms) and data.

The operating system is responsible for the following activities in connection with file management:

- Creating and deleting files.
- Creating and deleting directories to organize files.
- Supporting primitives for manipulating files and directories.
- Mapping files onto secondary storage.
- Backing up files on stable (nonvolatile) storage media.

Mass Storage Management

The operating system is responsible for the following activities in connection with disk management:

- Free-space management
- Storage allocation
- Disk scheduling

Caching

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.

Level	1	2	3	4
Name	registers	cache	main memory	disk storage
Typical size	< 1 KB	< 16 MB	< 64 GB	> 100 GB
Implementation technology	custom memory with multiple ports, CMOS	on-chip or off-chip CMOS SRAM	CMOS DRAM	magnetic disk
Access time (ns)	0.25 – 0.5	0.5 – 25	80 – 250	5,000.000
Bandwidth (MB/sec)	20,000 – 100,000	5000 – 10,000	1000 – 5000	20 – 150
Managed by	compiler	hardware	operating system	operating system
Backed by	cache	main memory	disk	CD or tape

Figure 1.11 Performance of various levels of storage.

Chapter 2

Operating Services

An operating system provides an environment for the execution of programs. It provides certain services to programs and to the users of those programs.

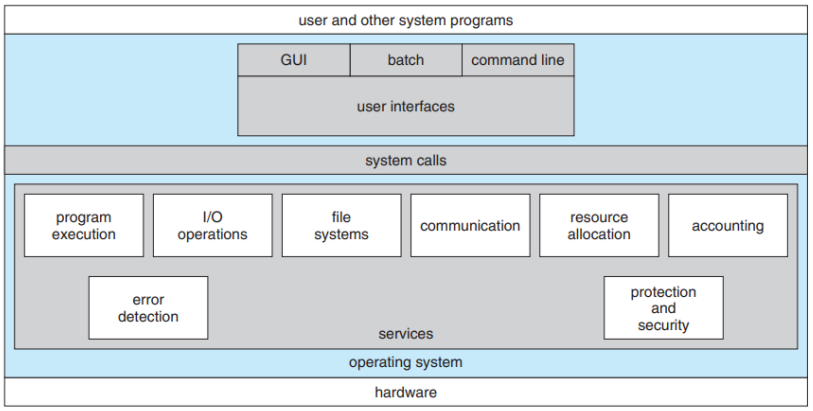


Figure 2.1 A view of operating system services.

1. User Interface
    a. Command Line Interface
    b. Batch Interface
    c. Graphical User Interface
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
5. Communications

There are many circumstances in which one process needs to exchange information with another process. Communications may be implemented via **shared memory**, in which two or more processes read and write to a shared section of memory, or **message passing**.

6. Error Detection

The operating system needs to be detecting and correcting errors constantly. Errors may occur in the CPU and memory hardware. For each type of error, the operating system should take the appropriate action to ensure correct and consistent computing.

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

7. Resource Allocation

When there are multiple users or multiple jobs running at the same time, resources must be allocated to each of them. For instance, in determining how best to use the CPU, operating systems have CPU-scheduling routines that take into account the speed of the CPU, the jobs that must be executed, the number of registers available, and other factors.

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

The owners of information stored in a multiuser or networked computer system may want to control use of that information. When several separate processes execute concurrently, it should not be possible for one process to interfere with the others or with the operating system itself. Protection involves ensuring that all access to system resources is controlled.

Operating Services

User and Operating-System Interface

Here, we discuss two fundamental approaches. One provides a command-line interface, or command interpreter, that allows users to directly enter commands to be performed by the operating system. The other allows users to interface with the operating system via a graphical user interface, or GUI.

Command Interpreters

On systems with multiple command interpreters to choose from, the interpreters are known as shells. The main function of the command interpreter is to get and execute the next user-specified command.

1. In one approach, the command interpreter itself contains the code to execute the command. For example, a command to delete a file may cause the command interpreter to jump to a section of its code that sets up the parameters and makes the appropriate system call.
2. An alternative approach—used by UNIX, among other operating systems implements most commands through system programs. In this case, the command interpreter does not understand the command in any way; it merely uses the command to identify a file to be loaded into memory and executed.

Graphical User Interfaces

A second strategy for interfacing with the operating system is through a user-friendly graphical user interface, or GUI. Here, rather than entering commands directly via a command-line interface, users employ a mouse-based window and menu system characterized by a desktop metaphor.

Because a mouse is impractical for most mobile systems, smartphones and handheld tablet computers typically use a touchscreen interface. Here, users interact by making gestures on the touchscreen. For example, pressing and swiping fingers across the screen.

Choice of Interface

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

System Calls

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

Many system calls: first to write a prompting message on the screen and then to read from the keyboard the characters that define the two files. Once the two file names have been obtained, the program must open the input file and create the output file. Each of these operations requires another system call. Possible error conditions for each operation can require additional system calls. When the program tries to open the input file, for example, it may find that there is no file of that name or that the file is protected against access. In these cases, the program should print a message on the console (another sequence of system calls) and then terminate abnormally (another system call). If the input file exists, then we must create a new output file. We may find that there is already an output file with the same name. This situation may cause the program to abort (a system call), or we may delete the existing file (another system call) and create a new one (yet another system call). Another option, in an interactive system, is to ask the user (via a sequence of system calls to output the prompting message and to read the response from the terminal) whether to replace the existing file or to abort the program

When both files are set up, we enter a loop that reads from the input file (a system call) and writes to the output file (another system call). Each read and write must return status information regarding various possible error conditions. Finally, after the entire file is copied, the program may close both files (another system call), write a message to the console or window (more system calls), and finally terminate normally (the final system call).

As you can see, even simple programs may make heavy use of the operating system. Frequently, systems execute thousands of system calls per second.

Typically, application developers design programs according to an **application programming interface (API)**. The API specifies a set of functions that are available to an application programmer, including the parameters that are passed to each function and the return values the programmer can expect.

Behind the scenes, the functions that make up an API typically invoke the actual system calls on behalf of the application programmer. For example, the Windows function CreateProcess() (which unsurprisingly is used to create a new process) actually invokes the NTCreateProcess() system call in the Windows kernel. **Actual system calls can often be more detailed and difficult to work with than the API available to an application programmer.**

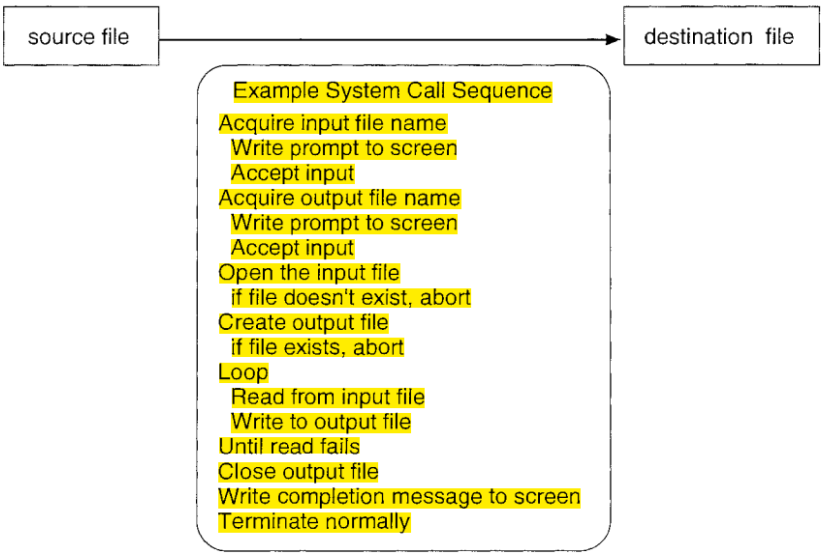


Figure 2.4 Example of how system calls are used.

EXAMPLE OF STANDARD API

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

```
man read
```

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

```
#include <unistd.h>

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

return value — function name — parameters

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

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

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

For most programming languages, 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**. Then the system-call interface invokes the intended system call in the operating-system kernel and returns the status of the system call and any return values.

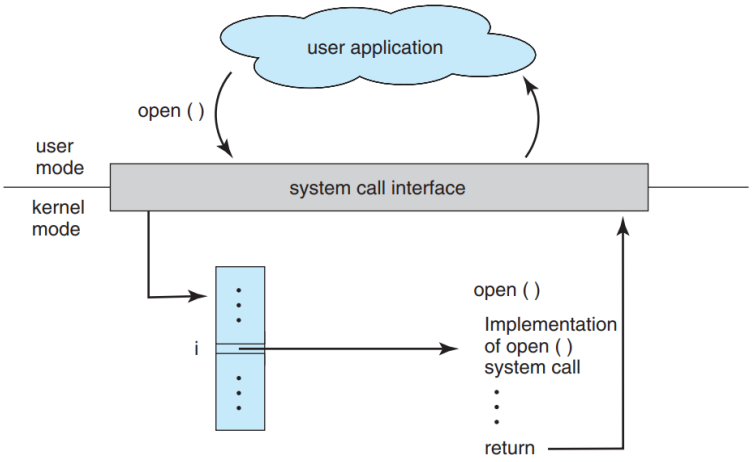


Figure 2.6 The handling of a user application invoking the `open()` system call.

Thus, most of the details of the operating-system interface are hidden from the programmer by the API and are managed by the run-time support library.

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

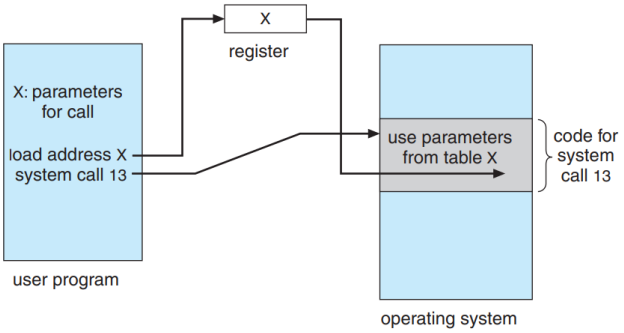


Figure 2.7 Passing of parameters as a table.

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.

- Process control
  - end, abort
  - load, execute
  - create process, terminate process
  - get process attributes, set process attributes
  - wait for time
  - wait event, signal event
  - allocate and free memory

EXAMPLES OF WINDOWS AND UNIX SYSTEM CALLS

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

Process Control

1) End, Abort

A running program needs to be able to halt its execution either normally (end()) or abnormally (abort()). If a system call is made to terminate the currently running program abnormally, or if the program runs into a problem and causes an error trap, a dump of memory is sometimes taken and an error message generated. The dump is written to disk and may be examined by a debugger—a system program designed to aid the programmer in finding and correcting errors, or bugs—to determine the cause of the problem. Under either normal or abnormal circumstances, the operating system must transfer control to the invoking command interpreter.

It is then possible to combine normal and abnormal termination by defining a normal termination as an error at level 0. The command interpreter or a following program can use this error level to determine the next action automatically.

2) Load, Execute

A process or job executing one program may want to load() and execute() another program. This feature allows the command interpreter to execute a program as directed by, for example, a user command, the click of a mouse, or a batch command. An interesting question is where to return control when the loaded program terminates.

3) Create Process

If control returns to the existing program when the new program terminates, we must save the memory image of the existing program; thus, we have effectively created a mechanism for one program to call another program. If both programs continue concurrently, we have created a new job or process to be multiprogrammed. Often, there is a system call specifically for this purpose (create process() or submit job()).

4) Get Process Attributes, Set Process Attributes, Terminate Process

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

5) Wait For Time, Wait Event, Signal Event

Having created new jobs or processes, we may need to wait for them to finish their execution. We may want to wait for a certain amount of time to pass (wait time()). More probably, we will want to wait for a specific event to occur (wait event()). The jobs or processes should then signal when that event has occurred (signal event()).

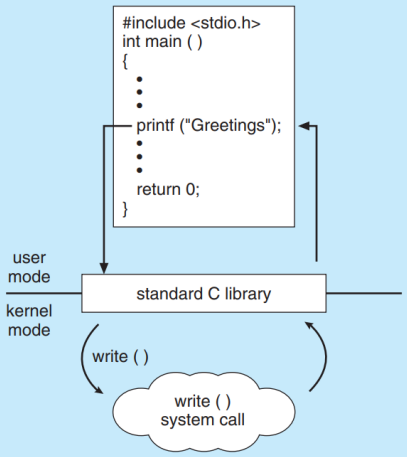
6) Lock, Release Lock

Quite often, two or more processes may share data. 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().

Error => System Trap
Terminate => System Call

EXAMPLE OF STANDARD C LIBRARY

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



The MS-DOS operating system is an example of a single-tasking system. It has a command interpreter that is invoked when the computer is started (Figure 2.9(a)). Because MS-DOS is single-tasking, it uses a simple method to run a program and does not create a new process. It loads the program into memory, writing over most of itself to give the program as much memory as possible (Figure 2.9(b)). Next, it sets the instruction pointer to the first instruction of the program. The program then runs, and either an error causes a trap, or the program executes a system call to terminate. In either case, the error code is saved in the system memory for later use. Following this action, the small portion of the command interpreter that was not overwritten resumes execution. Its first task is to reload the rest of the command interpreter from disk.

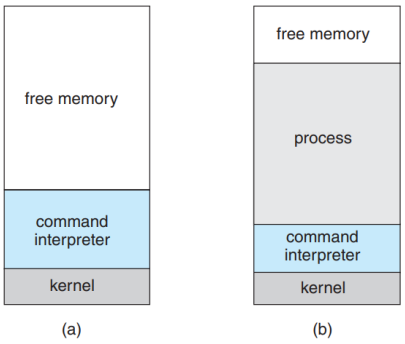


Figure 2.9 MS-DOS execution. (a) At system startup. (b) Running a program.

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

To start a new process, the shell executes a fork() system call. Then, the selected program is loaded into memory via an exec() system call, and the program is executed. Depending on the way the command was issued, the shell then either waits for the process to finish or runs the process "in the background." In the latter case, the shell immediately requests another command. When a process is running in the background, it cannot receive input directly from the keyboard, because the shell is using this resource. I/O is therefore done through files or through a GUI interface. Meanwhile, the user is free to ask the shell to run other programs, to monitor the progress of the running process, to change that program's priority, and so on. When the process is done, it executes an exit() system call to terminate, returning to the invoking process a status code of 0 or a nonzero error code. This status or error code is then available to the shell or other programs.

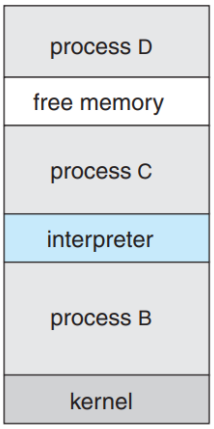


Figure 2.10 FreeBSD running multiple programs.

File Management

We first need to be able to create() and delete() files. Either system call requires the name of the file and perhaps some of the file's attributes. Once the file is created, we need to open() it and to use it. We may also read(), write(), or reposition() (rewind or skip to the end of the file, for example). Finally, we need to close() the file, indicating that we are no longer using it.

File attributes include the file name, file type, protection codes, accounting information, and so on. At least two system calls, get file attributes() and set file attributes(), are required for this function. Some operating systems provide many more calls, such as calls for file move() and copy().

Device Management

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

A system with multiple users may require us to first request() a device, to ensure exclusive use of it. After we are finished with the device, we release() it. These functions are similar to the open() and close() system calls for files. Other operating systems allow unmanaged access to devices.

Once the device has been requested (and allocated to us), we can read(), write(), and (possibly) reposition() the device, just as we can with files.

Information Maintenance

Many system calls exist simply for the purpose of transferring information between the user program and the operating system. For example, most systems have a system call to return the current time() and date().

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

Chapter 3

Process Concept

A question that arises in discussing operating systems involves what to call all the CPU activities. **A batch system executes jobs**, whereas **a time-shared system has user programs, or tasks**.

The Process

Informally, as mentioned earlier, **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.

We emphasize that a program by itself is not a process. **A program is a passive entity**, such as a file containing a list of instructions stored on disk (often called an executable file). In contrast, **a process is an active entity**, **with a program counter specifying the next instruction to execute** and a set of associated resources. A program becomes a process when an executable file is loaded into memory.

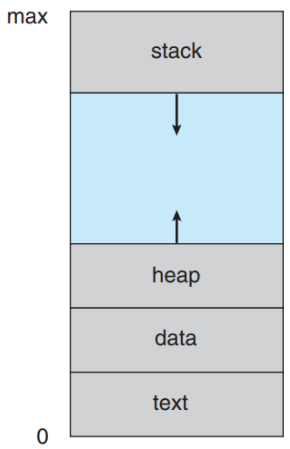


Figure 3.1 Process in memory.

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. It is also common to have a process that spawns many processes as it runs.

Note that a process itself can be an execution environment for other code. The Java programming environment provides a good example.

Process State

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

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

Processor Control Block

Each process is represented in the operating system by a process control block (PCB) also called a task control block.

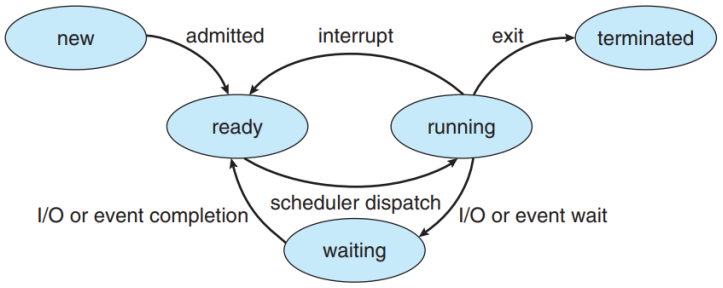


Figure 3.2 Diagram of process state.

Process State: The state may be new, ready, running, waiting, halted, and so on.

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

CPU Registers: The registers vary in number and type, depending on the computer architecture. They include accumulators, index registers, stack pointers, and general-purpose registers, plus any condition-code information. Along with the program counter, this state information must be saved when an interrupt occurs, to allow the process to be continued correctly afterward.

CPU Scheduling Information: This information includes a process priority, pointers to scheduling queues, and any other scheduling parameters.

Memory Management Information: This information may include such 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.

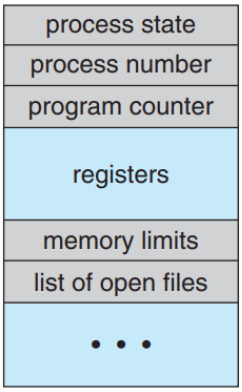


Figure 3.3 Process control block (PCB).

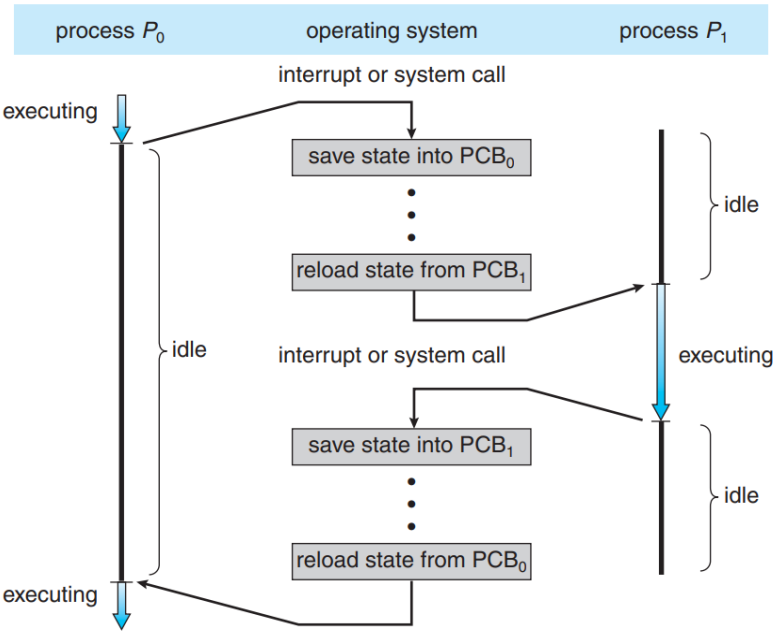


Figure 3.4 Diagram showing CPU switch from process to process.

Account Information: This information includes the amount of CPU and real time used, time limits, account numbers, job or process numbers, and so on.

I/O Status Information: This information includes the list of I/O devices allocated to the process, a list of open files, and so on.

Threads

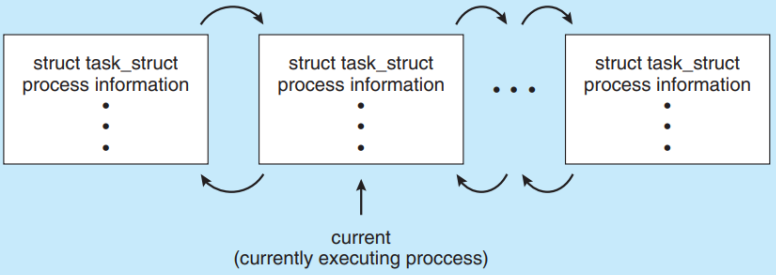
The process model discussed so far has implied that a process is a program that performs a single thread of execution. For example, when a process is running a word-processor program, a single thread of instructions is being executed. This single thread of control allows the process to perform only one task at a time. The user cannot simultaneously type in characters and run the spell checker within the same process, for example. 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. Other changes throughout the system are also needed to support threads.

PROCESS REPRESENTATION IN LINUX

The process control block in the Linux operating system is represented by the C structure `task_struct`, which is found in the `<linux/sched.h>` include file in the kernel source-code directory. This structure contains all the necessary information for representing a process, including the state of the process, scheduling and memory-management information, list of open files, and pointers to the process's parent and a list of its children and siblings. (A process's **parent** is the process that created it; its **children** are any processes that it creates. Its **siblings** are children with the same parent process.) Some of these fields include:

```
long state; /* state of the process */
struct sched_entity se; /* scheduling information */
struct task_struct *parent; /* this process's parent */
struct list_head children; /* this process's children */
struct files_struct *files; /* list of open files */
struct mm_struct *mm; /* address space of this process */
```

For example, the state of a process is represented by the field `long state` in this structure. Within the Linux kernel, all active processes are represented using a doubly linked list of `task_struct`. The kernel maintains a pointer—`current`—to the process currently executing on the system, as shown below:



As an illustration of how the kernel might manipulate one of the fields in the `task_struct` for a specified process, let's assume the system would like to change the state of the process currently running to the value `new_state`. If `current` is a pointer to the process currently executing, its state is changed with the following:

```
current->state = new_state;
```

(A process's parent is the process that created it; its children are any processes that it creates. Its siblings are children with the same parent process.)

Process Scheduling

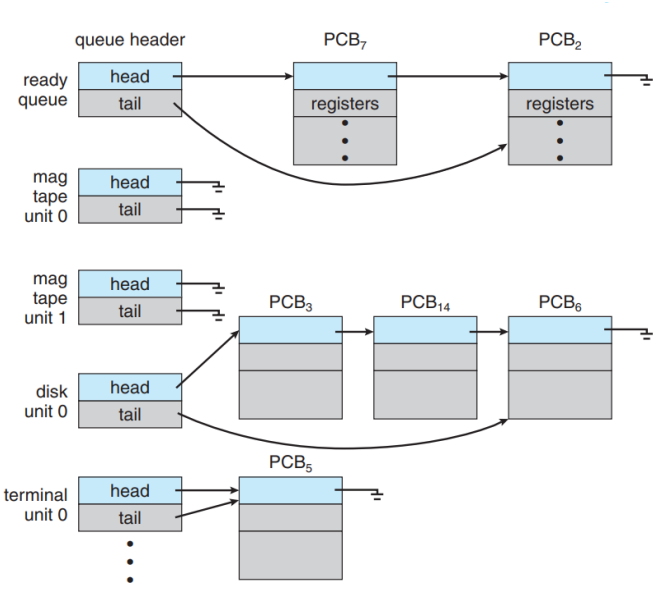


Figure 3.5 The ready queue and various I/O device queues.

Scheduling Queues

As processes enter the system, they are put into a job queue, which consists of all processes in the system. The processes that are residing in main memory and are ready and waiting to execute are kept on a list called the ready queue. This queue is generally stored as a linked list. A ready-queue header contains pointers to the first and final PCBs in the list. Each PCB includes a pointer field that points to the next PCB in the ready queue.