

Operating System

UNIT-I

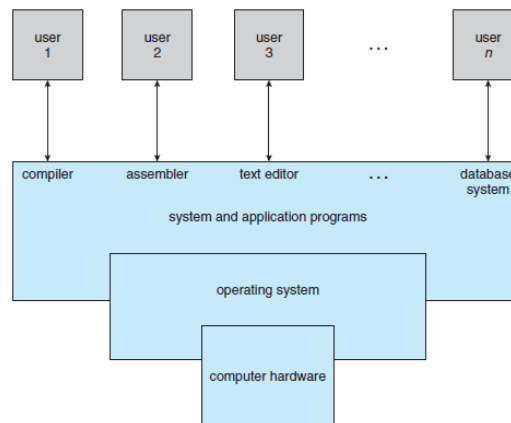
1.1 introduction

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

An **operating system** is a program that manages a computer's hardware. It also provides a basis for application programs and acts as an intermediary between the computer user and the computer hardware. An amazing aspect of operating systems is how they vary in accomplishing these tasks. 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. Operating systems for mobile computers provide an environment in which a user can easily interface with the computer to execute programs. Thus, some operating systems are designed to be *convenient*, others to be *efficient*, and others to be some combination of the two.

1.2 Computer system overview

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



1.1.1 User View

The user's view of the computer varies according to the interface being used. Most computer users sit in front of a PC, consisting of a monitor, keyboard, mouse, and system unit. Such a system is designed for one user to monopolize its resources. The goal is to maximize the work (or play) that the user is performing. In this case, the operating system is designed mostly for **ease of use**, with some attention paid to performance and none paid to **resource utilization**—how various hardware and software resources are shared.

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.

1.1.2 System View

From the computer's point of view, the operating system is the program most intimately involved with the hardware. In this context, we can view an operating system as a **resource allocator**. 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. As we have seen, resource allocation is especially important where many users access the same mainframe or minicomputer.

1.3 Basic elements

A modern general-purpose computer system consists of one or more CPUs and a number of device controllers connected through a common bus that provides access to shared memory (Figure 1.2). Each device controller is in charge of a specific type of device (for example, disk drives, audio devices, or video displays). The CPU and the device controllers can execute in parallel, competing for memory cycles. To ensure orderly access to the shared memory, a memory controller synchronizes access to the memory.

For a computer to start running—for instance, when it is powered up or rebooted—it needs to have an initial program to run. This initial program, or **bootstrap program**, tends to be simple. Typically, it is stored within the computer hardware in read-only memory (**ROM**) or electrically erasable programmable read-only memory (**EEPROM**), known by the general term **firmware**. It initializes all aspects of the system, from CPU registers to device controllers to memory contents. The bootstrap program must know how to load the operating system and how to start executing that system. To accomplish this goal, the bootstrap program must locate the operating-system kernel and load it into memory. Once the kernel is loaded and executing, it can start providing services to the system and its users. Some services are provided outside of the kernel, by system programs that are loaded into memory at boot time to become **system processes**, or **system daemons** that run the entire time the kernel is running.

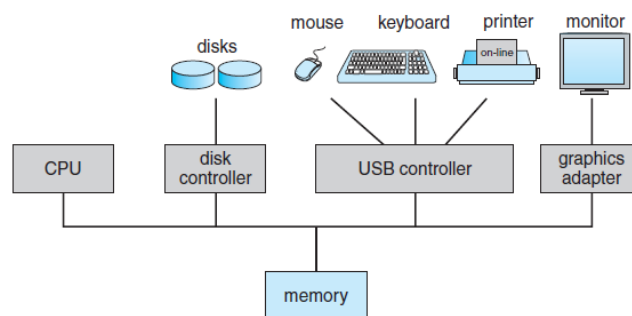


Figure 1.2 A modern computer system.

1.4 Instruction Execution and Interrupts

The occurrence of an event is usually signaled by an **interrupt** from either the hardware or the software. Hardware may trigger an interrupt at any time by sending a signal to the CPU, usually by way of the

system bus. Software may trigger an interrupt by executing a special operation called a **system call** (also called a **monitor call**).

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

Interrupts are an important part of computer architecture. Each computer design has its own interrupt mechanism, but several functions are common. The interrupt must transfer control to the appropriate interrupt service routine. The straightforward method for 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.

Chapter 1 Introduction

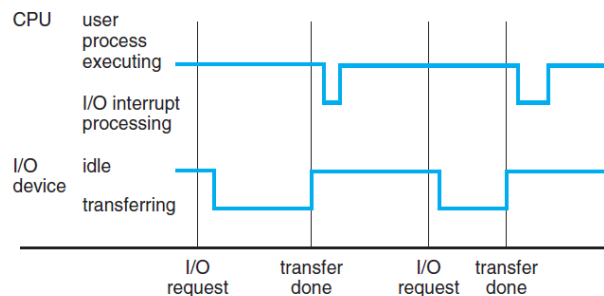


Figure 1.3 Interrupt timeline for a single process doing output.

1.5 Memory Hierarchy

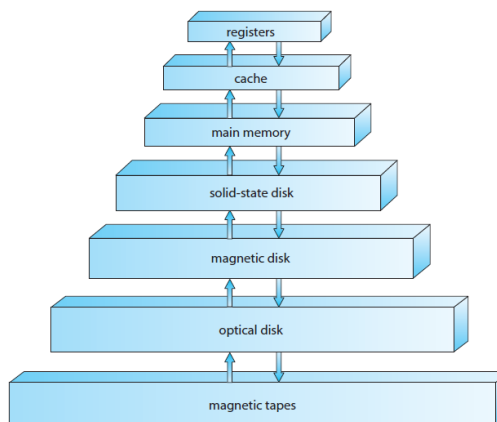


Figure 1.4 Storage-device hierarchy.

The wide variety of storage systems can be organized in a hierarchy (Figure 1.4) according to speed and cost. The higher levels are expensive, but they are fast. As we move down the hierarchy, the cost per bit generally decreases, whereas the access time generally increases. This trade-off is reasonable; if a given storage system were both faster and less expensive than another—other properties being the same—then there would be no reason to use the slower, more expensive memory. In fact, many early storage devices, including paper tape and core memories, are relegated to museums now that magnetic tape and **semiconductor memory** have become faster and cheaper. The top four levels of memory in Figure 1.4 may be constructed using semiconductor memory.

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

Solid-state disks have several variants but in general are faster than magnetic disks and are nonvolatile. One type of solid-state disk stores data in a large DRAM array during normal operation but also contains a hidden magnetic hard disk and a battery for backup power. If external power is interrupted, this solid-state disk's controller copies the data from RAM to the magnetic disk. When external power is restored, the controller copies the data back into RAM. Another form of solid-state disk is flash memory, which is popular in cameras and **personal digital assistants (PDAs)**, in robots, and increasingly for storage on general-purpose computers. Flash memory is slower than DRAM but needs no power to retain its contents. Another form of nonvolatile storage is **NVRAM**, which is DRAM with battery backup power. This memory can be as fast as DRAM and (as long as the battery lasts) is nonvolatile.

1.6 Cache Memory

- Important principle, performed at many levels in a computer (in hardware, operating system, software)
- Information in use copied from slower to faster storage temporarily
- Faster storage (cache) checked first to determine if information is there
 - ➔ If it is, information used directly from the cache (fast)
 - ➔ If not, data copied to cache and used there
- Cache smaller than storage being cached
 - ➔ Cache management important design problem
 - ➔ Cache size and replacement policy

1.7 Direct Memory Access

Interrupt-driven I/O is fine for moving small amounts of data but can produce high overhead when used for bulk data movement such as disk I/O. To solve this problem, **direct memory access (DMA)** is used. After setting up buffers, pointers, and counters for the I/O device, the device controller transfers an entire block of data directly to or from its own buffer storage to memory, with no intervention by the CPU. Only one interrupt is generated per block, to tell the device driver that the operation has completed, rather than the one interrupt per byte generated for low-speed devices. While the device controller is performing these operations, the CPU is available to accomplish other work.

Some high-end systems use switch rather than bus architecture. On these systems, multiple components can talk to other components concurrently, rather than competing for cycles on a shared bus. In this case, DMA is even more effective. Figure 1.5 shows the interplay of all components of a computer system.

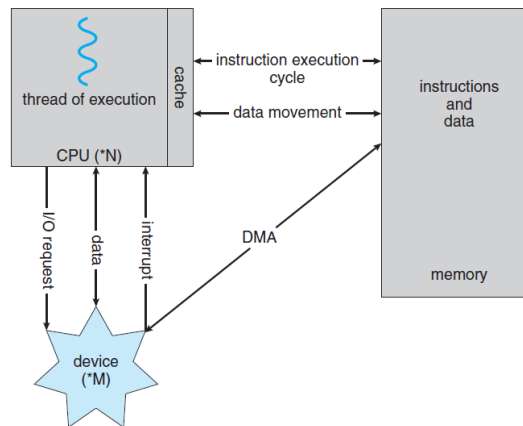


Figure 1.5 How a modern computer system works.

1.7 Multiprocessor and Multicore Organization

1.7.1 Multiprocessor Organization

Within the past several years, **multiprocessor systems** (also known as **parallel systems** or **multicore systems**) have begun to dominate the landscape of computing. Such systems have two or more processors in close communication, sharing the computer bus and sometimes the clock, memory, and peripheral devices. Multiprocessor systems first appeared prominently in servers and have since migrated to desktop and laptop systems. Recently, multiple processors have appeared on mobile devices such as smart phones and tablet computers. Multiprocessor systems have three main advantages:

1. 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. Similarly, N programmers working closely together do not produce N times the amount of work a single programmer would produce.

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

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. Some systems use **asymmetric multiprocessing**, in which each processor is assigned a specific task. A **boss** processor controls the system; the other processors either look to the boss for instruction or have predefined tasks. This scheme defines a boss–

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

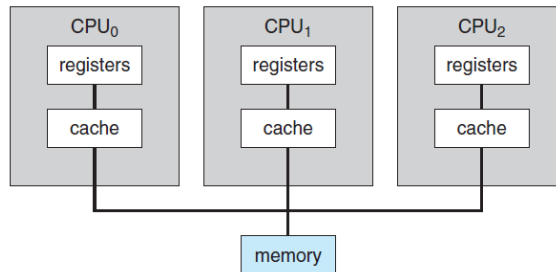


Figure 1.6 Symmetric multiprocessing architecture.

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

1.7.2 Multicore Organization

Earlier in the history of computer design, in response to the need for more computing performance, single-CPU systems evolved into multi-CPU systems. A more recent, similar trend in system design is to place multiple computing cores on a single chip. Each core appears as a separate processor to the operating system (Section 1.3.2). Whether the cores appear across CPU chips or within CPU chips, we call these systems **multicore** or **multiprocessor** systems. Multithreaded programming provides a mechanism for more efficient use of these multiple computing cores and improved concurrency. Consider an application with four threads. On a system with a single computing core, concurrency merely means that the execution of the threads will be interleaved over time (Figure 4.3), because the processing core is capable of executing only one thread at a time. On a system with multiple cores, however, concurrency means that the threads can run in parallel, because the system can assign a separate thread to each core (Figure 4.4).

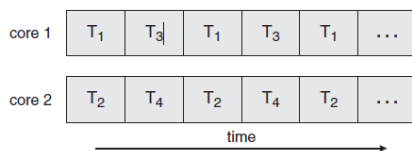


Figure 4.4 Parallel execution on a multicore system.

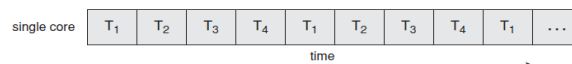


Figure 4.3 Concurrent execution on a single-core system.

Notice the distinction between *parallelism* and *concurrency* in this discussion. 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. Thus, it is possible to have concurrency without parallelism. Before the advent of SMP and multicore architectures, most computer systems had only a single processor. CPU schedulers were designed to provide the illusion of parallelism by rapidly switching between processes in the system, thereby allowing each process to make progress. Such processes were running concurrently, but not in parallel.

As systems have grown from tens of threads to thousands of threads, CPU designers have improved system performance by adding hardware to improve thread performance. 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. Multicore computers will no doubt continue to increase in core counts and hardware thread support.

1.8 Operating system overview-objectives and functions.

Objectives and Functions

- A program that is executed by the processor that frequently relinquishes control and must depend on the processor to regain control.
 - A program that mediates between application programs and the hardware
 - A set of procedures that enable a group of people to use a computer system.
 - A program that controls the execution of application programs
 - An interface between applications and hardware

Functions

Usage

Computer system

Control

Support

Usage

- ❖ Users of a computer system:
 - ❖ Programs - use memory, use CPU time, use I/O devices
 - ❖ Human users
 - ❖ Programmers - use program development tools such as debuggers, editors and users - use application programs, e.g. Internet explorer

Computer system

hardware + software

OS is a part of the computer software, it is a program. It is a very special program, that is the first to be executed when the computer is switched on, and is supposed to control and support the execution of other programs and the overall usage of the computer system.

Control

The operating system controls the usage of the computer resources - hardware devices and software utilities. We can think of an operating system as a *Resource Manager*. Here are some of the resources managed by the OS:

- Processors,
- Main memory,
- Secondary Memory,
- Peripheral devices,
- Information.

Support

- ✓ The operating system provides a number of services to assist the users of the computer system:

For the programmers:

Utilities - debuggers, editors, file management, etc.

For the end users - provides the interface to the application programs

For programs - loads instructions and data into memory, prepares I/O devices for usage, handles interrupts and error conditions.

1.9 Evolution of Operating System

1.9.1 **Serial Processing** - 1940's – 1950's programmer interacted directly with hardware. No operating system.

Problems

Scheduling - users sign up for machine time. Wasted computing time

Setup Time- Setup included loading the compiler, source program, saving compiled program, and loading and linking. If an error occurred - start over.

1.9.2 Simple Batch Systems

Improve the utilization of computers.

Jobs were submitted on cards or tape to an operator who batches jobs together sequentially. The program that controls the execution of the jobs was called **monitor** - a simple version of an operating system. The interface to the monitor was accomplished through Job Control Language (JCL). For example, a JCL request could be to run the compiler for a particular programming language, then to link and load the program, then to run the user program.

Hardware features:

Memory protection: do not allow the memory area containing the monitor to be altered

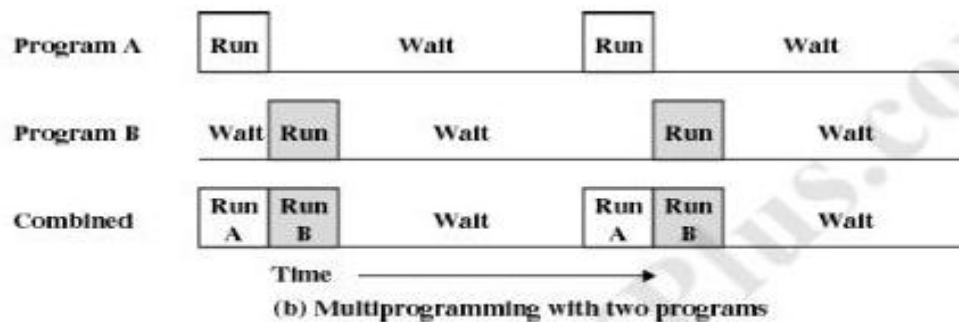
Timer: prevents a job from monopolizing the system

Problems:

Bad utilization of CPU time - the processor stays idle while I/O devices are in use.

1.9.3 Multiprogrammed Batch Systems

More than one program resides in the main memory. While a program A uses an I/O device the processor does not stay idle, instead it runs another program B.



New features:

Memory management - to have several jobs ready to run, they must be kept in main memory

Job scheduling - the processor must decide which program to run.

1.9.4 Time-Sharing Systems

Multiprogramming systems: Several programs use the computer system.

Time-sharing systems: Several (human) users use the computer system interactively.

Characteristics:

- Using multiprogramming to handle multiple interactive jobs
- Processor's time is shared among multiple users
- Multiple users simultaneously access the system through terminals

1.9.5 Operating-System Services

The OS provides certain services to programs and to the users of those programs.

1. 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).

2. I/O operations:

A running program may require I/O. This I/O may involve a file or an I/O device.

3. File-system manipulation:

The program needs to read, write, create and delete files.

4. Communications :

In many circumstances, one process needs to exchange information with another process. Such communication can occur in two major ways. The first takes place between processes that are executing on the same computer; the second takes place between processes that are executing on different computer systems that are tied together by a computer network.

5. Error detection:

The operating system constantly needs to be aware of possible errors. 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 tape, 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.

6. Resource allocation:

Different types of resources are managed by the Os.

When there are multiple users or multiple jobs running at the same time, resources must be allocated to each of them.

7. Accounting:

We want to keep track of which users use how many and which kinds of computer resources. This record keeping may be used for accounting or simply for accumulating usage statistics.

8. Protection:

The owners of information stored in a multiuser computer system may want to control use of that information. Security of the system is also important.

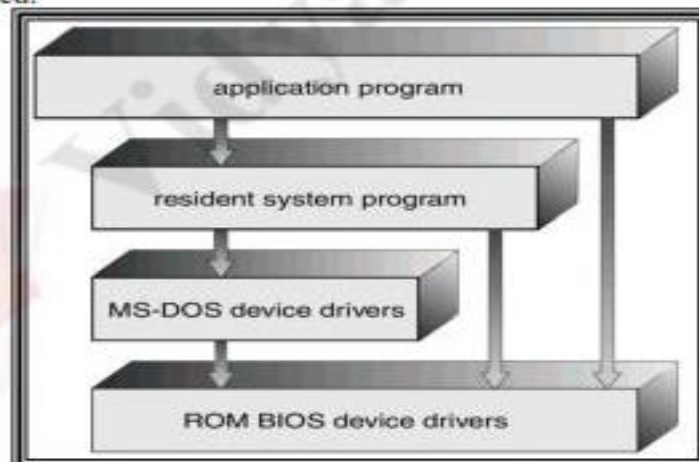
1.10 Computer System Organization

1.10.1 Operating System Structure and Operations

1.10.1 Operating System Structure

1.10.1.1 MS-DOS System Structure

- ✓ MS-DOS – written to provide the most functionality in the least space.
- ✓ Not divided into modules.
- ✓ Although MS-DOS has some structure, its interfaces and levels of functionality are not well separated.



1.10.1.2 Unix System Structure

- **UNIX** – limited by hardware functionality, the original UNIX operating system had limited structuring. The UNIX OS consists of two separable parts.

- **Systems programs** – use kernel supported system calls to provide useful functions such as compilation and file manipulation.
- **The kernel** - Consists of everything below the system-call interface and above the physical hardware

1.10.1.3 Layered Approach

- ✓ The operating system is divided into a number of layers (levels), each built on top of lower layers. The bottom layer (layer 0), is the hardware; the highest (layer N) is the user interface.
- ✓ An OS layer is an implementation of an abstract object that is the encapsulation of data and operations that can manipulate those data. These operations (routines) can be invoked by higher-level layers. The layer itself can invoke operations on lower-level layers.
- ✓ Layered approach provides modularity. With modularity, layers are selected such that each layer uses functions (operations) and services of only lower-level layers.
- ✓ Each layer is implemented by using only those operations that are provided lower level layers.
- ✓ The major difficulty is appropriate definition of various layers.

1.10.1.4 Microkernel System Structure

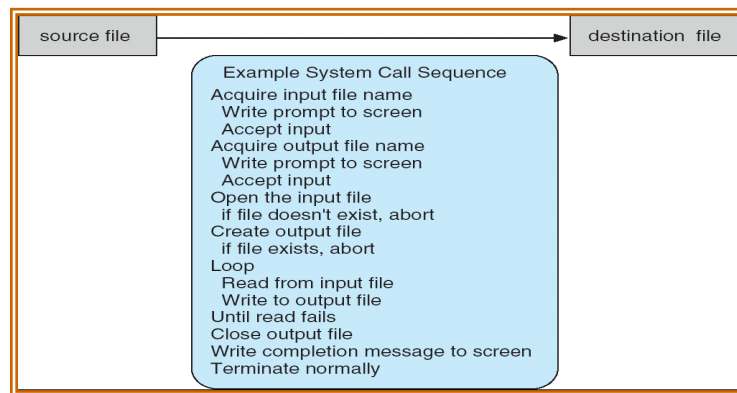
- ✓ Moves as much from the kernel into “user” space.
- ✓ Communication takes place between user modules using message passing.
 - ❖ Benefits:
 - Easier to extend a microkernel
 - Easier to port the operating system to new architectures

1.10.2 Operating-System Operations

- ✓ If there are no processes to execute, no I/O devices to service, and no users to whom to respond, an operating system will sit quietly, waiting for something to happen. Events are almost always signaled by the occurrence of an interrupt or a trap.
- ✓ A trap (or an exception) is a software-generated interrupt caused either by an error (for example, division by zero or invalid memory access) or by a specific request from a user program that an operating-system service be performed. The interrupt-driven nature of an operating system defines that system’s general structure.
- ✓ 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.

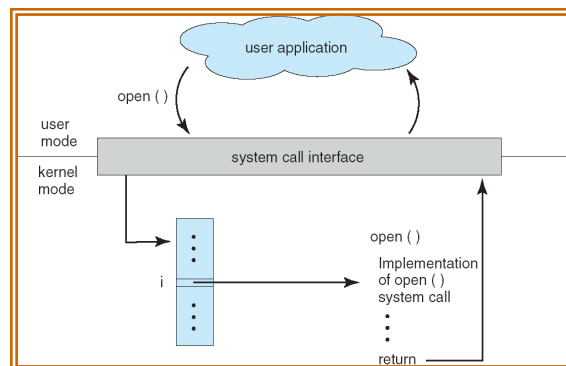
✓ System Calls

- **Programming interface to the services provided by the OS**
- **Typically written in a high-level language (C or C++)**
- **Mostly accessed by programs via a high-level Application Program Interface (API) rather than direct system call use**
- **Three most common APIs are Win32 API for Windows, POSIX API for POSIX-based systems (including virtually all versions of UNIX, Linux, and Mac OS X), and Java API for the Java virtual machine (JVM)**
- **System call sequence to copy the contents of one file to another file**



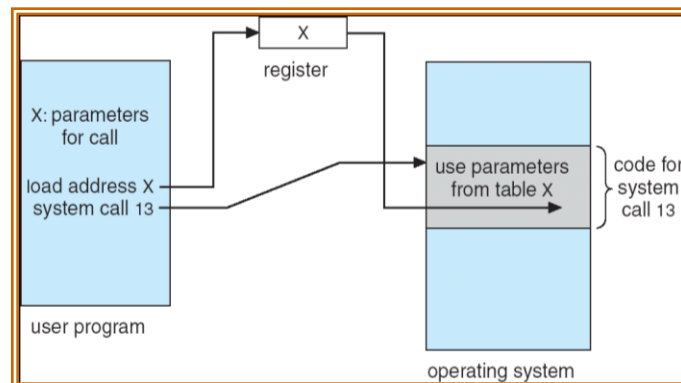
1.11.1 System Call Implementation

- **Typically, a number associated with each system call**
- ✓ **System-call interface maintains a table indexed according to these numbers**
- **The system call interface invokes intended system call in OS kernel and returns status of the system call and any return values**
- **The caller need know nothing about how the system call is implemented**
- ✓ **Just needs to obey API and understand what OS will do as a result call**
- ✓ **Most details of OS interface hidden from programmer by API**
- ✓ **Managed by run-time support library (set of functions built into libraries included with compiler)**



1.11.2 System Call Parameter Passing

- Often, more information is required than simply identity of desired system call
 - Exact type and amount of information vary according to OS and call
- Three general methods used to pass parameters to the OS
 - Simplest: pass the parameters in *registers*
 - ▶ In some cases, may be more parameters than registers
 - Parameters stored in a *block*, or table, in memory, and address of block passed as a parameter in a register
 - ▶ This approach taken by Linux and Solaris
 - Parameters placed, or *pushed*, onto the *stack* by the program and *popped* off the stack by the operating system
 - Block and stack methods do not limit the number or length of parameters being passed



Types of System Calls

- **Process control**
- **File management**
- **Device management**
- **Information maintenance**
- **Communications**

1.12 System Programs

- System programs provide a convenient environment for program development and execution. They can be divided into:
 - File manipulation
 - Status information
 - File modification

- Programming language support
 - Program loading and execution
 - Communications
 - Application programs
- Most users' view of the operation system is defined by system programs, not the actual system calls
 - Provide a convenient environment for program development and execution
 - Some of them are simply user interfaces to system calls; others are considerably more complex
 - File management - Create, delete, copy, rename, print, dump, list, and generally manipulate files and directories
 - Status information
 - Some ask the system for info - date, time, amount of available memory, disk space, number of users
 - Others provide detailed performance, logging, and debugging information
 - Typically, these programs format and print the output to the terminal or other output devices
 - Some systems implement a registry - used to store and retrieve configuration information
 - File modification
 - Text editors to create and modify files
 - Special commands to search contents of files or perform transformations of the text
 - Programming-language support - Compilers, assemblers, debuggers and interpreters sometimes provided
 - Program loading and execution- Absolute loaders, relocatable loaders, linkage editors, and overlay-loaders, debugging systems for higher-level and machine language
 - Communications - Provide the mechanism for creating virtual connections among processes, users, and computer systems
 - Allow users to send messages to one another's screens, browse web pages, send electronic-mail messages, log in remotely, transfer files from one machine to another

1.12 OS Generation and System Boot.

1.12.1 OS Generation

Historically operating systems have been tightly related to the computer architecture, it is good idea to study the history of operating systems from the architecture of the computers on which they run.

Operating systems have evolved through a number of distinct phases or generations which corresponds roughly to the decades.

The 1940's - First Generations

The earliest electronic digital computers had no operating systems. Machines of the time were so primitive that programs were often entered one bit at time on rows of mechanical switches (plug boards). Programming languages were unknown (not even assembly languages). Operating systems were unheard of .

The 1950's - Second Generation

By the early 1950's, the routine had improved somewhat with the introduction of punch cards. The General Motors Research Laboratories implemented the first operating systems in early 1950's for their IBM 701. The system of the 50's generally ran one job at a time. These were called single-stream batch processing systems because programs and data were submitted in groups or batches.

The 1960's - Third Generation

The systems of the 1960's were also batch processing systems, but they were able to take better advantage of the computer's resources by running several jobs at once. So operating systems designers developed the concept of multiprogramming in which several jobs are in main memory at once; a processor is switched from job to job as needed to keep several jobs advancing while keeping the peripheral devices in use.

For example, on the system with no multiprogramming, when the current job paused to wait for other I/O operation to complete, the CPU simply sat idle until the I/O finished. The solution for this problem that evolved was to partition memory into several pieces, with a different job in each partition. While one job was waiting for I/O to complete, another job could be using the CPU.

Another major feature in third-generation operating system was the technique called spooling (simultaneous peripheral operations on line). In spooling, a high-speed device like a disk interposed between a running program and a low-speed device involved with the program in input/output. Instead of writing directly to a printer, for example, outputs are written to the disk. Programs can run to completion faster, and other programs can be initiated sooner when the printer becomes available, the outputs may be printed.

Note that spooling technique is much like thread being spun to a spool so that it may be later be unwound as needed.

Another feature present in this generation was time-sharing technique, a variant of multiprogramming technique, in which each user has an on-line (i.e., directly connected) terminal. Because the user is

present and interacting with the computer, the computer system must respond quickly to user requests, otherwise user productivity could suffer. Timesharing systems were developed to multiprogram large number of simultaneous interactive users.

Fourth Generation

With the development of LSI (Large Scale Integration) circuits, chips, operating system entered in the system entered in the personal computer and the workstation age. Microprocessor technology evolved to the point that it become possible to build desktop computers as powerful as the mainframes of the 1970s. Two operating systems have dominated the personal computer scene: MS-DOS, written by Microsoft, Inc. for the IBM PC and other machines using the Intel 8088 CPU and its successors, and UNIX, which is dominant on the large personal computers using the Motorola 6899 CPU family.

12.2 System Boot.

- Operating system must be made available to hardware so hardware can start it
 - Small piece of code – **bootstrap loader**, locates the kernel, loads it into memory, and starts it
 - Sometimes two-step process where **boot block** at fixed location loads bootstrap loader
 - When power initialized on system, execution starts at a fixed memory location
 - ▶ Firmware used to hold initial boot code