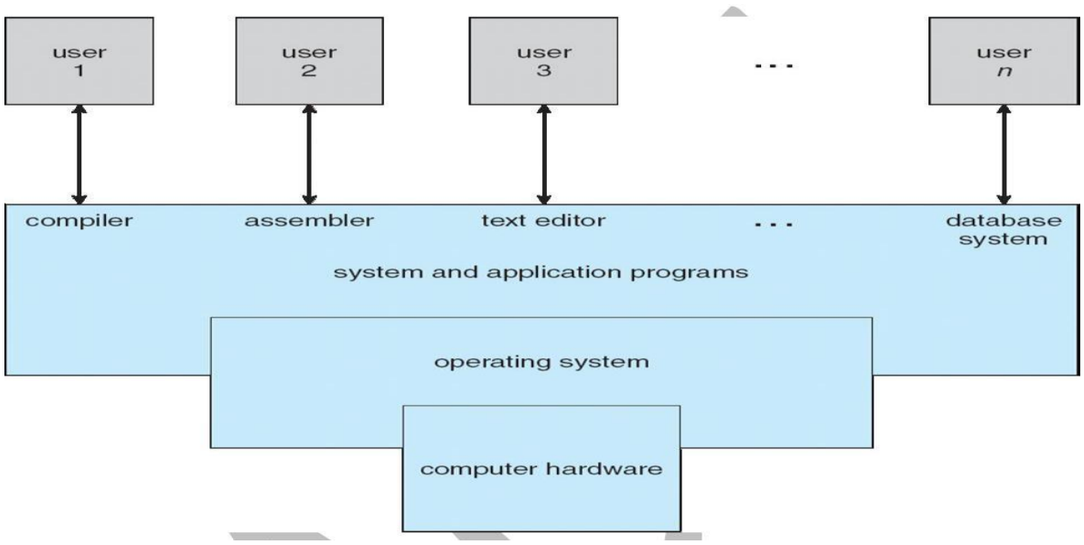
Introduction to Operating Systems

An **operating system** is a program that manages the computer hardware. It also acts as an intermediary between the computer user 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. Operating systems for handheld computers are designed to 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 some combination of the two.

1.1 What Operating Systems Do

We begin our discussion by looking at the operating system's role in theoverall computer system. A computer system can be divided roughly intofour components: the *hardware,* the *operating system,* the *application programs,*and the *users*



Abstract view of the components of a computer system.

The **hardware—**the **central processing unit (CPU),** the **memory,** and the**input/output (I/O) devices**—provides the basic computing resources for thesystem. The **application programs**—such as word processors, spreadsheets,compilers, and web browsers—define the ways in which these resources areused to solve users' computing problems. The operating system controls andcoordinates the use of the hardware among the various application programs

for the various users.

We can also view a computer system as consisting of hardware, software,and data. The operating system provides the means for proper use of theseresources in the operation of the computer system.To understand more fully the operating system's role, we next explore

operating systems from two viewpoints: that of the user and that of the system.

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

In other cases, a user sits at a terminal connected to a **mainframe** or **minicomputer.** Other users are accessing the same computer through other terminals. These users share resources and may exchange information. The operating system in such cases is designed to maximize resource utilization to assure

that all available CPU time, memory, and I/O are used efficiently and that no individual user takes more than her fair share.

In still other cases, users sit at **workstations** connected to networks of other workstations and **servers.** These users have dedicated resources at their disposal, but they also share resources such as networking and servers—file, compute, and print servers.

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.

An operating system is a control program. A **control program** manages the execution of user programs to prevent errors and improper use of the computer. It is especially concerned with the operation and control of I/O devices.

2. Storage Structure

Computer programs must be in main memory (also called **random-access memory** or **RAM)** to be executed. Main memory is the only large storage area (millions to billions of bytes) that the processor can access directly. It commonly is implemented in a semiconductor technology called **dynamic random-access memory (DRAM),** which forms an array of memory words. Each word has its own address. Interaction is achieved through a sequence of load or store instructions to specific memory addresses. The load instruction moves a word from main memory to an internal register within the CPU, whereas the store instruction moves the content of a register to main memory. Aside from explicit loads and stores, the CPU automatically loads instructions from main memory for execution.

A typical instruction-execution cycle, as executed on a system with a **von Neumann** architecture, first fetches an instruction from memory and stores that instruction in the **instruction register.** The instruction is then decoded and may cause operands to be fetched from memory and stored in some internal register. After the instruction on the operands has been executed, the result may be stored back in memory. Notice that the memory unit sees only a stream of memory addresses; it does not know how they are generated (by the instruction counter, indexing, indirection, literal addresses, or some other means) or what they are for (instructions or data). Accordingly, we can ignore *hoio* a memory address is generated by a program. We are interested only in the sequence of memory addresses generated by the running program. Ideally, we want the programs and data to reside in main memory permanently. This arrangement usually is not possible for the following tworeasons:

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

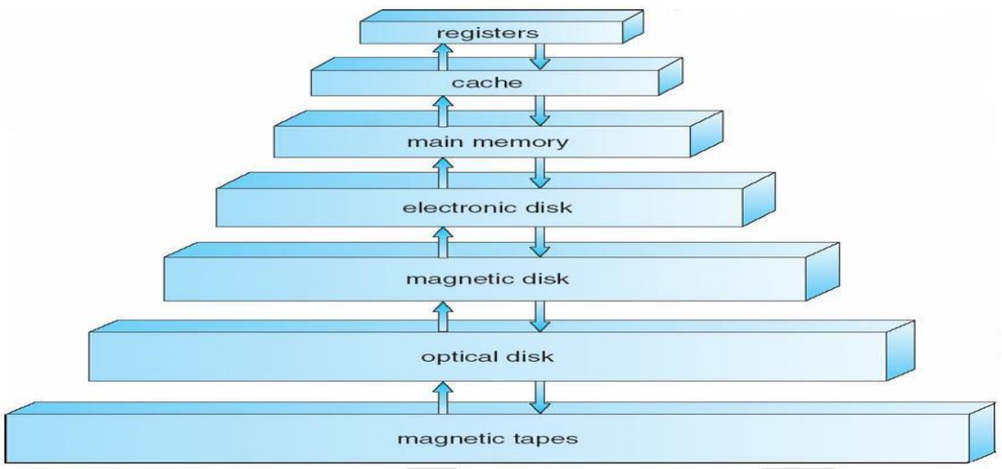
Thus, most computer systems provide **secondary storage** as an extension of main memory. The main requirement for secondary storage is that it be able to hold large quantities of data permanently.

The most common secondary-storage device is a **magnetic disk,** which provides storage for both programs and data. Most programs (web browsers, compilers, word processors, spreadsheets, and so on) are stored on a disk until they are loaded into memory. Many programs then use the disk as both a source and a destination of the information for their processing

In a larger sense, however, the storage structure that we have described consisting of registers, main memory, and magnetic disks—is only one of many possible storage systems. Others include cache memory, CD-ROM, magnetic tapes, and so on. Each storage system provides the basic functions of

storing a datum and of holding that datum until it is retrieved at a later time. The main differences among the various storage systems lie in speed, cost, size,and volatility.

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. 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 the storage systems above the electronic disk are volatile, whereas those below are nonvolatile. An **electronic disk** can be designed to be either volatile or nonvolatile. During normal operation, the electronic disk stores data in a large DRAM array, which is volatile. But many electronic-disk devices contain a hidden magnetic hard disk and a battery for backup power. If external power is interrupted, the electronic-disk controller copies the data from RAM to the magnetic disk. When external power is restored, the controller copies the data back into the RAM. Another form of electronic disk is flash memory, which is popular in cameras and **personal digital assistants (PDAs),** in robots, and increasingly as removable storage on general-purpose computers. Flashmemory 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 but has a limited duration in which it is nonvolatile.

3. Computer-System Architecture

A computer system may be organized in a number of different ways, which we can categorize roughly according to the number of general-purpose processors used.

3.1. Single-Processor Systems

Most systems use a single processor. On a single-processor system, there is one main CPU capableof executing a general-purpose instruction set, including instructions fromuser processes. Almost all systems have other special-purpose processors aswell. They may come in the form of device-specific

processors, such as disk,keyboard, and graphics controllers; or, on mainframes, they may come in theform of more general-purpose processors, such as I/O processors that movedata rapidly among the components of the system.

For example, a disk-controller microprocessorreceives a sequence of requests from the main CPU and implements its own diskqueue and scheduling algorithm. This arrangement relieves the main CPU ofthe overhead of disk scheduling. PCs contain a microprocessor in the keyboardto convert the keystrokes into codes to be sent to the CPU. The use of special-purposemicroprocessors is common and does not turn a single-processor system intoa multiprocessor.

3.2. Multiprocessor Systems

Although single-processor systems are most common, **multiprocessor systems**(also known as **parallel systems** or **tightly coupled systems)** are growingin importance. Such systems have two or more processors in close communication,sharing the computer bus and sometimes the clock, memory, andperipheral devices.Multiprocessor systems have three main advantages:

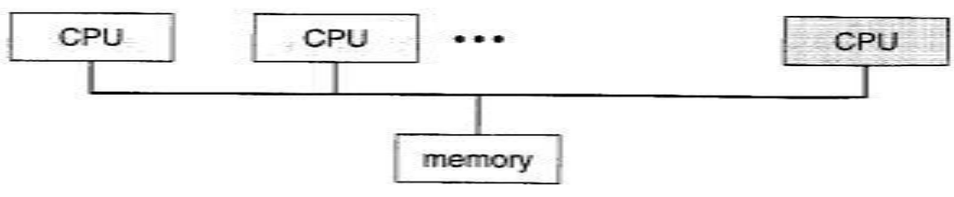
1. Increased throughput. By increasing the number of processors, we expectto get more work done in less time. The speed-up ratio with *N* processorsis not *N*, however; rather, it is less than N. When multiple processorscooperate on a task, a certain amount of overhead is incurred in keepingall the parts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

2. Economy of scale. Multiprocessor systems can cost less than equivalentmultiple single-processor systems, because they can share peripherals,mass storage, and power supplies. If several programs operate on thesame set of data, it is cheaper to store those data on one disk and to haveall the processors share them than to have many computers with localdisks and many copies of the data.

3. **Increased reliability.** If functions can be distributed properly amongseveral processors, then the failure of one processor will not halt thesystem, only slow it down. If we have ten processors and one fails, theneach of the remaining nine processors can pick up a share of the work ofthe 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. Somesystems use **asymmetric multiprocessing,** in which each processor is assigneda specific task. A master processor controls the system; the other processorseither look to the master for instruction or have predefined tasks. This schemedefines a master-slave relationship. The master processor schedules andallocates work to the slave processors.

The most common systems use **symmetric multiprocessing (SMP),** inwhich each processor performs all tasks within the operating system. SMPmeans that all processors are peers; no master-slave relationship existsbetween processors. Figure illustrates a typical SMP architecture. An example of the SMP system is Solaris.



Symmetric multiprocessing architecture

The benefit of this model is that many processescan run simultaneously—*N* processes can run if there are *N* CPUs—withoutcausing a significant deterioration of performance. However, we must carefullycontrol I/O to ensure that the data reach the appropriate processor. Also, sincethe CPUs are separate, one may be sitting idle while another is overloaded,resulting in inefficiencies. These

inefficiencies can be avoided if the processorsshare certain data structures. A multiprocessor system of this form will allowprocesses and resources—such as memory—to be shared dynamically amongthe various processors and can lower the variance among the processors. Allmodern operating systems—including Windows, Windows XP, Mac OS X, andLinux—now provide support for SMP.

The difference between symmetric and asymmetric multiprocessing mayresult from either hardware or software. Special hardware can differentiate themultiple processors, or the software can be written to allow only one master andmultiple slaves.

3.3. Clustered Systems

Another type of multiple-CPU system is the **clustered system.** Like multiprocessorsystems, clustered systems gather together multiple CPUs to accomplishcomputational work. Clustered systems differ from multiprocessor systems,however, in that they are composed of two or more individual systemscoupled together. The definition of the term *clustered* is not concrete. The generally accepted definition is that clusteredcomputers share storage and are closely linked via a **local-area network (LAN)**Clustering is usually used to provide **high-availability** service; that is,service will continue even if one or more systems in the cluster fail.

Clustering can be structured asymmetrically or symmetrically.

In **asymmetric clustering,** one machine is in **hot-standby mode** while the other is running the applications. The hot-standby host machine does nothing but monitor the active server. If that server fails, the hot-standby host becomes the active server.

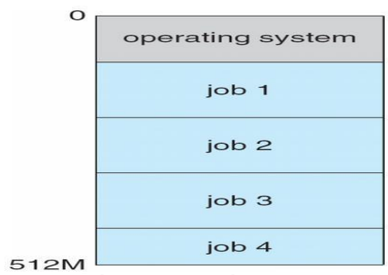
In **symmetric mode,** two or more hosts are running applications, and are monitoring each other. This mode is obviously more efficient, as it uses all of the available hardware. It does require that more than one application be available to run.

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.

4 Operating-System Structure

An operating system provides the environment within which programs areexecuted. One of the most important aspects of operating systems is the ability tomultiprogram. A single user cannot, in general, keep either the CPU or theI/O devices busy at all times. **Multiprogramming** increases CPU utilization byorganizing jobs (code and data) so that the CPU always has one to execute.

The idea is as follows: The operating system keeps several jobs in memorysimultaneously (Figure 1.7). This set of jobs can be a subset of the jobs kept inthe job pool—which contains all jobs that enter the system—since the numberof jobs that can be kept simultaneously in memory is usually smaller thanthe number of jobs that can be kept in the job pool. The operating systempicks and begins to execute one of the jobs in memory. Eventually, the jobmay have to wait for some task, such as an I/O operation, to complete.



Memory layout for a multiprogramming system.

In a non-multiprogrammed system, the CPU would sit idle. In a multiprogrammedsystem, 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 atleast one job needs to execute, the CPU is never idle.

Multiprogrammed systems provide an environment in which the varioussystem resources (for example, CPU, memory, and peripheral devices) areutilized effectively, but they do not provide for user interaction with thecomputer system.

Time sharing (or **multitasking)** is a logical extension ofmultiprogramming. In time-sharing systems, the CPU executes multiple jobsby switching among them, but the switches occur so frequently that the userscan 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. Theuser gives instructions to the operating system or to a program directly, using ainput device such as a keyboard or a mouse, and waits for immediate results onan output device. Accordingly, the **response time** should be short—typicallyless than one second.

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

Time-sharing and multiprogramming require several jobs to be keptsimultaneously in memory. Since in general main memory is too small toaccommodate 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 mainmemory. If several jobs are ready to be brought into memory, and if there isnot enough room for all of them, then the system must choose among them.Making this decision is **job scheduling,** Whenthe operating system selects a job from the job pool, it loads that job intomemory for execution. if several jobs are ready to run at the same time, the systemmust choose among them. Making this decision is **CPU scheduling.** Finally, running multiple jobs concurrently requiresthat their ability to affect one another be limited in all phases of the operatingsystem, including process scheduling, disk storage, and memory management.

5 Operating-System Operations

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

Since the operating system and the users share the hardware and software resources of the computer system, With sharing, many processes could be adversely affected by a bug in one program. For example, if a process gets stuck in an infinite loop, this loop could prevent the correct operation of many other processes. More subtle errors can occur in a multiprogramming system, where one erroneous program might modify another program, the data of another program, or even the operating system itself.

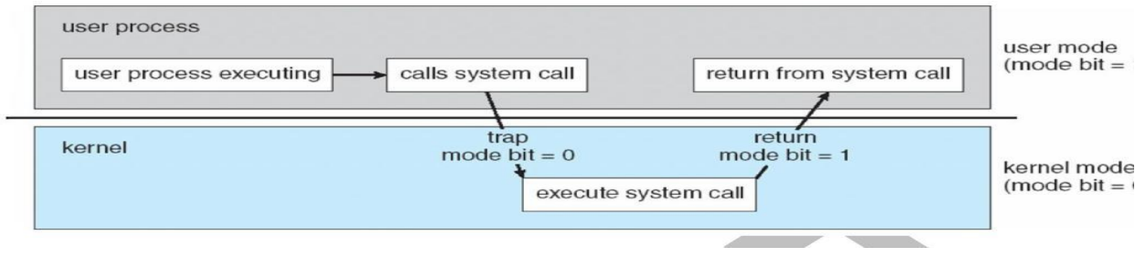
A properly designed operating system must ensure that an incorrect (or malicious) program cannot cause other programs to execute incorrectly.

5.1 Dual-Mode Operation

In order to ensure the proper execution of the operating system, we must be able to distinguish between the execution of operating-system code and userdefined code. The approach taken by most

computer systems is to provide hardware support that allows us to differentiate among various modes of execution.

At the very least, we need two separate **modes** of operation: **user mode**and **kernel mode** (also called **supervisor mode, system mode, or privilegedmode).** A bit, called the **mode bit,** is added to the hardware of the computer toindicate the current mode: kernel (0) or user (1). With the mode bit, we are ableto distinguish between a task that is executed on behalf of the operating systemand one that is executed on behalf of the user. When the computer system isexecuting on behalf of a user application, the system is in user mode. However,when a user application requests a service from the operating system (via asystem call), it must transition from user to kernel mode to fulfill the request.



This is shown in Figure. As we shall see, At system boot time, the hardware starts in kernel mode. The operatingsystem is then loaded and starts user applications in user mode. Whenever atrap 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 operatingsystem gains control of the computer, it is in kernel mode. The system alwaysswitches to user mode (by setting the mode bit to 1) before passing control toa user program.

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

5.2 Timer

We must prevent a user program from getting stuck in an infinite loop or not calling system services and never returning control to the operating system.To accomplish this goal, we can use a **timer.** A timer can be set to interrupt the computer after a specified period. The period may be fixed (for example, 1/60 second) or variable (for example, from 1 millisecond to 1 second). A **variable timer** is generally implemented by a fixed-rate clock and a counter. The operating system sets the counter. Every time the clock ticks, the counter is decremented. When the counter reaches 0, an interrupt occurs

Before turning over control to the user, the operating system ensures that the timer is set to interrupt. If the timer interrupts, control transfers automatically to the operating system, which may treat the interrupt as a fatal error or may give the program more time. Clearly, instructions that modify thecontent of the timer are privileged.Thus, we can use the timer to prevent a user program from running toolong.

6 OPERATING SYSTEM FUNCTIONS

Process Management

- A process is a program in execution. It is a unit of work within the system. Program is a *passive entity*, process is an *active entity*.
- Process needs resources to accomplish its task
- CPU, memory, I/O, files

- Initialization data
- Process termination requires reclaim of any reusable resources
- Single-threaded process has one **program counter** specifying location of next instruction to execute
- Process executes instructions sequentially, one at a time, until completion
- Multi-threaded process has one program counter per thread
- Typically system has many processes, some user, some operating system running concurrently on one or more CPUs
- Concurrency by multiplexing the CPUs among the processes / threads

Process Management Activities

- The operating system is responsible for the following activities in connection with process management:
- Creating and deleting both user and system processes
- Suspending and resuming processes
- Providing mechanisms for process synchronization
- Providing mechanisms for process communication
- Providing mechanisms for deadlock handling

Memory Management

- All data in memory before and after processing
- All instructions in memory in order to execute
- Memory management determines what is in memory when
- Optimizing CPU utilization and computer response to users
- **Memory management activities**
- 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

- OS provides uniform, logical view of information storage
- Abstracts physical properties to logical storage unit - **file**
- Each medium is controlled by device (i.e., disk drive, tape drive)
- Varying properties include access speed, capacity, data-transfer rate, access method (sequential or random)
- File-System management
- Files usually organized into directories
- Access control on most systems to determine who can access what
- **OS activities include**
- Creating and deleting files and directories
- Primitives to manipulate files and dirs
- Mapping files onto secondary storage
- Backup files onto stable (non-volatile) storage media

Mass-Storage Management

- Usually disks used to store data that does not fit in main memory or data that must be kept for a ―long‖ period of time
- Proper management is of central importance
- Entire speed of computer operation hinges on disk subsystem and its algorithms
- **MASS STORAGE activities**
- Free-space management
- Storage allocation
- Disk scheduling
- Some storage need not be fast
- Tertiary storage includes optical storage, magnetic tape
- Still must be managed
- Varies between WORM (write-once, read-many-times) and RW (read-write)

I/O Subsystem

- One purpose of OS is to hide peculiarities of hardware devices from the user
- I/O subsystem responsible for
- Memory management of I/O including buffering (storing data temporarily while it is being transferred), caching (storing parts of data in faster storage for performance), spooling (the overlapping of output of one job with input of other jobs)
- General device-driver interface
- Drivers for specific hardware devices

Protection and Security

Protection – any mechanism for controlling access of processes or users to resources defined by the OS

Security – defense of the system against internal and external attacks

- Huge range, including denial-of-service, worms, viruses, identity theft, theft of service
- Systems generally first distinguish among users, to determine who can do what
- User identities (**user IDs**, security IDs) include name and associated number, one per user
- User ID then associated with all files, processes of that user to determine access control
- Group identifier (**group ID**) allows set of users to be defined and controls managed, then also associated with each process, file
- **Privilege escalation** allows user to change to effective ID with more rights

7 . Protection and Security

If a computer system has multiple users and allows the concurrent execution of multiple rocesses, then access to data must be regulated. For that purpose, mechanisms ensure that files, memory segments, CPU, and other resources can be operated on by only those processes that have gained proper authorization from the operating system.

Protection, then, is any mechanism for controlling the access of processes or users to the resources defined by a computer system. This mechanism must provide means for specification of the controls to be imposed and means for enforcement. Protection can improve reliability by detecting latent errors at the interfaces between component subsystems.

A system can have adequate protection but still be prone to failure and allow inappropriate access. Consider a user whose authentication information (her means of identifying herself to the system) is stolen. Her data could be copied or deleted, even though file and memory protection are working. It is the job of **security** to defend a system from external and internal attacks. Such attacks spread across a

huge range and include viruses and worms, denial-ofservice attacks , identity theft, and theft of service (unauthorized use of a system).

8. Distributed Systems

A distributed system is a collection of physically separate, possibly heterogeneous computer systems that are networked to provide the users with access to the various resources that the system maintains. Access to a shared resource increases computation speed, functionality, data availability, and reliability.

A **network,** in the simplest terms, is a communication path between two or more systems. Distributed systems depend on networking for their functionality. Networks vary by the protocols used, the distances between nodes, and the transport media. TCP/IP is the most common network protocol,although ATM and other protocols are in widespread use.

Networks are characterized based on the distances between their nodes. A **local-area network (LAN)** connects computers within a room, a floor, or a building. A **wide-area network (WAN)** usually links buildings, cities, or countries. A global company may have a WAN to connect its offices worldwide. **metropolitan-area network** (MAN) could link buildings within a city. BlueTooth and 802.11 devices use wireless technology to communicate over a distance of several feet, in essence creating a **small-area network** such as might be found in a home.

The media to carry networks are equally varied. They include copper wires, fiber strands, and wireless transmissions between satellites, microwave dishes, and radios.

A **network operating system** is an operating system that provides features such as file sharing across the network and that includes a communication scheme that allows different processes on different computers to exchange messages. A computer running a network operating system acts autonomously from all other computers on the network, although it is aware of the network and is able to communicate with other networked computers.

9. Special-Purpose Systems

The discussion thus far has focused on general-purpose computer systems that we are all familiar with. There are, however, different classes of computer systems whose functions are more limited and whose objective is to deal with limited computation domains.

9.1 Real-Time Embedded Systems

Embedded computers are the most prevalent form of computers in existence. These devices are found everywhere, from car engines and manufacturing robots to VCRs and microwave ovens. They tend to have very specific tasks. The use of embedded systems continues to expand.

Embedded systems almost always run **real-time operating systems.** A real-time system is used when rigid time requirements have been placed on the operation of a processor or the flow of data; thus, it is often used as a control device in a dedicated application. Sensors bring data to the computer. The computer must analyze the data and possibly adjust controls to modify the sensor inputs. Systems that control scientific experiments, medical imaging systems, industrial control systems, and certain display systems are realtime systems. Some automobile-engine fuel-injection systems, home-appliancecontrollers, and weapon systems are also real-time systems.

A real-time system has well-defined, fixed time constraints. Processing *mustbe* done within the defined constraints, or the system will fail. A real-time system functions correctly only if it returns the correct result within its time constraints. Contrast this system with a time-sharing system, where it is desirable (but not mandatory) to respond quickly, or a batch system, which may have no time constraints at all.

9.2 Multimedia Systems

Most operating systems are designed to handle conventional data such as text files, programs, word-processing documents, and spreadsheets. However, a recent trend in technology is the incorporation of **multimedia data** into computer systems. Multimedia data consist of audio and video files as well as conventional files. These data differ from conventional data in that multimedia data—such as frames of video—must be delivered (streamed) according to certain time restrictions (for example, 30 frames per second).

Multimedia describes a wide range of applications that are in popular use today. These include audio files such as MP3 DVD movies, video conferencing, and short video clips of movie previews or news stories downloaded over theInternet. Multimedia applications may also include live webcasts Multimedia applications need not be either audio or video; rather, amultimedia application often includes a combination of both

9.3 Handheld Systems

Handheld systems include personal digital assistants (PDAs), such as Palmand Pocket-PCs, and cellular telephones, many of which use special-purposeembedded operating systems. Developers of handheld systems and applicationsface many challenges, most of which are due to the limited size of suchdevices. For example, a PDA is typically about 5 inches in height and 3 inchesin width, and it weighs less than one-half pound. Because of their size, mosthandheld devices have a small amount of memory, slow processors, and smalldisplay screens.

The amount of physical memory in a handheld depends upon the device,but typically is is somewhere between 512 KB and 128 MB. (Contrast this with atypical PC or workstation, which may have several gigabytes of memory!)As a result, the operating system and applications must manage memoryefficiently.

A second issue of concern to developers of handheld devices is the speedof the processor used in the devices. Processors for most handheld devicesrun at a fraction of the speed of a processor in a PC. Faster processors requiremore power. To include a faster processor in a handheld device would requirea larger battery, which would take up more space and would have to bereplaced (or recharged) more frequently. Most handheld devices use smaller,slower processors that consume less power. Therefore, the operating systemand applications must be designed not to tax the processor.

The last issue confronting program designers for handheld devices is I/O.A lack of physical space limits input methods to small keyboards, handwritingrecognition, or small screen-based keyboards. The small display screens limitoutput options. Whereas a monitor for a home computer may measure up to30 inches, the display for a handheld device is often no more than 3 inchessquare. Familiar tasks, such as reading e-mail and browsing web pages, mustbe condensed into smaller displays. One approach for displaying the contentin web pages is **web clipping,** where only a small subset of a web page isdelivered and displayed on the handheld device.

10. Operating-System Services

An operating system provides certain services to the users of those programs.. These operating-system services are provided for the convenience of the programmer, to make the programming task easier.

One set of operating-system services provides functions that are helpful to the user.

• **User interface.** Almost all operating systems have a **user interface** (UI). This interface can take several forms. One is a **command-line interface (CLI),** which uses text commands and a method for entering them (say, a program to allow entering and editing of commands **graphical user interface (GUI)** is used.

Here, the interface is a window system with a pointing device to direct I/O, choose from menus, and make selections and a keyboard to enter text. Some systems provide two or all three of these variations.

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

• **I/O operations.** A running program may require I/O, which may involve a file or an I/O device. For specific devices, special functions may be desired (such as recording to a CD or DVD drive or blanking a CRT screen). For efficiency and protection, users usually cannot control I/O devices directly. Therefore, the operating system must provide a means to do I/O.

• **File-system manipulation.** The file system is of particular interest. Obviously, programs need to read and write files and directories. They also need to create and delete them by name, search for a given file, and list file information. Finally, some programs include permissions management to allow or deny access to files or directories based on file ownership.

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

• **Error detection.** The operating system needs to be constantly 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. Debugging facilities can greatlyenhance the user's and programmer's abilities to use the system efficiently.

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.

Resource allocation. When there are multiple users or multiple jobs running at the same time, resources must be allocated to each of Them. Many different types of resources are managed by the operating system. 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. There may also be routines to allocate printers, modems, USB storage drives, and other peripheral devices.

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. Usage statistics may be a valuable tool for researchers who wish to reconfigiire the system to improve computing services.

Protection and security. The owners of information stored in a multiuser or networked computer system may want to control use of that information. When several separate processes execute concurrently, it should not be possible for one process to interfere with the others or with the operating system itself. Protection involves ensuring that all access to system resources is controlled. Security of the system from outsiders is also important. Such security starts with requiring each user to authenticate himself or herself to the system, usually by means of a password, to gain access to system resources. It extends to defending external I/O devices, including modems and network adapters, from invalid access attempts and to

recording all such connections for detection of break-ins. If a system is to be protected and secure, precautions must be instituted throughout it. A chain is only as strong as its weakest link.

11. System Calls

System calls provide an interface to the services made available by an operating system.
System calls can be grouped roughly into five major categories:

Processcontrol, file manipulation, device manipulation, information maintenance,and **communications.**

Most ofthese system calls support, or are supported by, concepts and functions thatare discussed in later chapters

. Process control

o end, abort

o load, execute

o create process, terminate process

o get process attributes, set process

attributes o wait for time

o wait event, signal event o

allocate and free memory

• File management

• create file, delete file

o open, close

• read, write, reposition

o get file attributes, set file attributes

• Device management

o request device, release device

° read, write, reposition

o get device attributes, set device attributes

° logically attach or detach devices

• Information maintenance

o get time or date, set time or date

o get system data, set system data

o get process, file, or device attributes

o set process, file, or device attributes

• Communications

0 create, delete communication connection

° send, receive messages

o transfer status information

o attach or detach remote devices

Process Control

A running program needs to be able to halt its execution either normally (end)or abnormally (abort). If a system call is made to terminate the currentlyrunning program abnormally, or if the program runs into a problem andcauses an error trap, and an errormessage generated.

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 (rewinding or skipping to the end of the file, for example). Finally, we need

to close the file, indicating that we are no longer using it. We may need these same sets of operations for directories if we have a

directory structure for organizing files in the file system. In addition, for either files or directories, we need to be able to determine the values of various attributes and perhaps to reset them if necessary. File attributes include the file name, a file type, protection codes, accounting information, and so on. At least two system calls, get file attribute and set file attribute, 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 willhave to wait until sufficient resources are available.

The various resources controlled by the operating sysstem can be thoughtof as devices. Some of these devices are physical devices (for example, tapes),while others can be thought of as abstract or virtual devices (for example,files). If there are multiple users of the system, the system may require us tofirst request the device, to ensure exclusive use of it. After we are finishedwith the device, we release it. These functions are similar to the open andclose system calls for files. Other operating systems allow unmanaged access

to devices.

Once the device has been requested (and allocated to us), we can read,write, and (possibly) reposition the device, just as we can with files. In fact,the similarity between I/O devices and files is so great that many operatingsystems, including UNIX, merge the two into a combined file-device structure.

Information Maintenance

Many system calls exist simply for the purpose of transferring informationbetween the user program and the operating system. For example, mostsystems have a system call to return the current t i m e and d a t e . Other systemcalls may return information about the system, such as the number of currentusers, the version number of the operating system, the amount of free memoryor disk space, and so on.

In addition, the operating system keeps information about all its processes,and svstem calls are used to access this information. Generally, calls arealso used to reset the process information (get process a t t r i b u t e s ands e t process a t t r i b u t e s) . In Section 3.1.3, we discuss what information isnormally kept.

Communication

There are two common models of interprocess communication:

the messagepassingmodel and the shared-memory model. In the message-passing model,the communicating processes exchange messages with one another to transferinformation. Messages can be exchanged between the processes either directlyor indirectly through a common mailbox. Before communication can takeplace, a connection must be opened. The name of the other communicator

must be known, be it another process on the same system or a process onanother computer connected by a communications network. Each computerin a network has a *host name* by which it is commonly known. A host alsohas a network identifier, such as an IP address. Similarly, each process hasa *process name,* and this name is translated into an identifier by which theoperating system can refer to the process. The get host id and get processed system calls do this translation. The identifiers are then passed to the generalpurposeopen and close calls provided by the file system or to specificopen connection and close

connection system calls, depending on thesystem's model of communication. The recipient process usually must give itspermission for communication to take place with an accept connection call.

In the shared-memory model, processes use shared memory create andshared memory attach system calls to create and gain access to regions ofmemory owned by other processes. Recall that, normally, the operating systemtries to prevent one process from accessing another process's memory. Sharedmemory requires that two or more processes agree to remove this restriction.They can then exchange information by reading and writing data in the sharedareas.

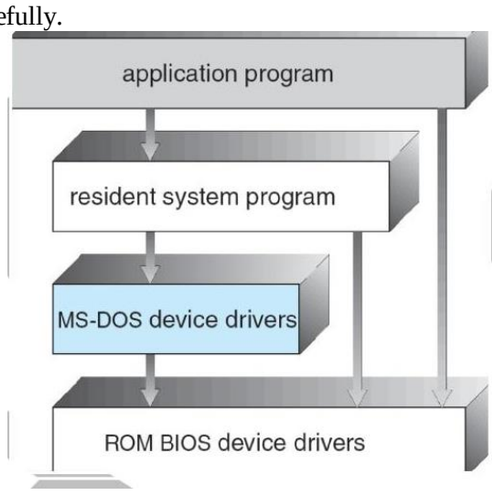
12. Operating-System Structure

A system as large and complex as a modern operating system must beengineered carefully if it is to function properly and be modified easily. Acommon approach is to partition the task into small components rather than

have one monolithic system.

12.1 Simple Structure

Many commercial systems do not have well-defined structures. Frequently,such operating systems started as small, simple, and limited systems and thengrew beyond their original scope. MS-DOS is an example of such a system. It wasoriginally designed and implemented by a few people who had no idea that itwould become so popular. It was written to provide the most functionality in the least space, so it was not divided into modules carefully.

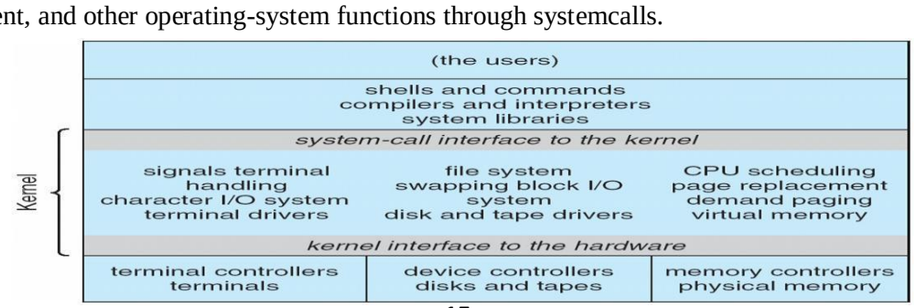


MS-DOS layer structure.

In MS-DOS, the interfaces and levels of functionality are not well separated.For instance, application programs are able to access the basic I/O routinesto write directly to the display and disk drives. Such freedom leaves MS-DOSvulnerable to errant (or malicious) programs, causing entire system crasheswhen user programs fail.

Another example of limited structuring is the original UNIX operatingsystem. UNIX is another system that initially was limited by hardware functionality.It consists of two separable parts: the kernel and the system programs.

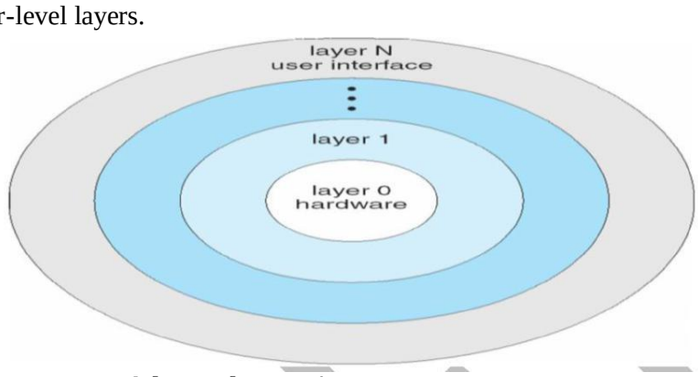
The kernel is further separated into a series of interfaces and device drivers,which have been added and expanded over the years as UNIX has evolved. Wecan view the traditional UNIX operating system as being layered, as shown inFigure. Everything below the system call interface and above the physicalhardware is the kernel. The kernel provides the file system, CPU scheduling,memory management, and other operating-system functions through systemcalls.



12.2 Layered Approach

With proper hardware support, operating systems can be broken into piecesthat are smaller and more appropriate than those allowed by the originalMS-DOS or UNIX systems. The operating system can then retain much greater
control over the computer and over the applications that make use of thatcomputer.

A system can be made modular in many ways. One method is the **layeredapproach,** in which the operating system is broken up into a number of layers(levels). The bottom layer (layer 0) is the hardware; the highest (layer N) is theuser interface. This layering structure is depicted in Figure.An operating-system layer is an implementation of an abstract object madeup of data and the operations that can manipulate those data. A typicaloperating-system layer—say, layer M—consists of data structures and a setof routines that can be invoked by higher-level layers. Layer *M*, in turn, can
invoke operations on lower-level layers.



A layered operating system.

The main advantage of the layered approach is simplicity of constructionand debugging. The layers are selected so that each uses functions (operations)and services of only lower-level layers. This approach simplifies debuggingand system verification. The first layer can be debugged without any concernfor the rest of the system, because, by definition, it uses only the basic hardware(which is assumed correct) to implement its functions. Once the first layer isdebugged, its correct functioning can be assumed while the second layer isdebugged, and so on. If an error is found during the debugging of a particularlayer, the error must be on that layer, because the layers below it are alreadydebugged. Thus, the design and implementation of the system is simplified.Each layer is implemented with only those operations provided by lowerlevellayers.

The major difficulty with the layered approach involves appropriatelydefining the various layers. Because a layer can use only lower-level layers,careful planning is necessary. For example, the device driver for the backingstore (disk space used by virtual-memory algorithms) must be at a lowerlevel than the memory-management routines, because memory managementrequires the ability to use the backing store.

A final problem with layered implementations is that they tend to be lessefficient than other types. For instance, when a user program executes an I/Ooperation, it executes a system call that is trapped to the I/O layer, which callsthe memory-management layer, which in turn calls the CPU-scheduling layer,which is then passed to the hardware. At each layer, the parameters may bemodified, data may need to be passed, and so on. Each layer adds overhead tothe system call; the net result is a system call that takes longer than does oneon a nonlayered system.

.**12.3 Microkernels**

In the mid-1980s, researchers at Carnegie MellonUniversity developed an operating system called **Mach** that modularizedthe kernel using the **microkernel** approach. This method structures theoperating system by removing all nonessential components from the kernel andimplementing them as system and user-level programs. The result is a smallerkernel. There is little consensus regarding which services

should remain in thekernel and which should be implemented in user space. Typically, however,microkernels provide minimal process and memory management, in additionto a communication facility.

The main function of the microkernel is to provide a communication facilitybetween the client program and the various services that are also runningin user space. Communication is provided by *message passing*

For example, if the client program wishes to accessa file, it must interact with the file server. The client program and service neverinteract directly. Rather, they communicate indirectly by exchanging messageswith the microkernel.

One benefit of the microkernel approach is ease of extending the operating system. All new services are added to user space and consequently do notrequire modification of the kernel. When the kernel does have to be modified,the changes tend to be fewer, because the microkernel is a smaller kernel.The resulting operating system is easier to port from one hardware designto another.

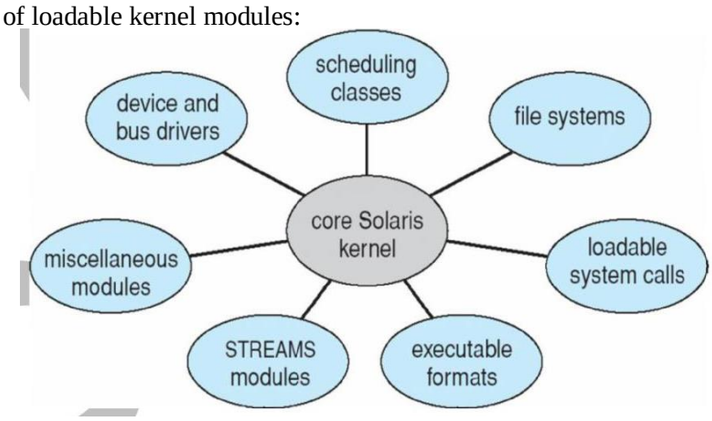
The microkernel also provides more security and reliability, sincemost services are running as user—rather than kernel—processes. If a servicefails, the rest of the operating system remains untouched.

Unfortunately, microkernels can suffer from performance decreases dueto increased system function overhead.

12.4 Modules

The best current methodology for operating-system design involvesusing object-oriented programming techniques to create a modular kernel.Here, the kernel has a set of core components and dynamically links inadditional services either during boot time or during run time. Such astrategy uses dynamically loadable modules and is common in modernimplementations of UNIX, such as Solaris, Linux, and Mac OS X. For example, theSolaris operating system structure is organized arounda core kernel with seven types of loadable kernel modules:

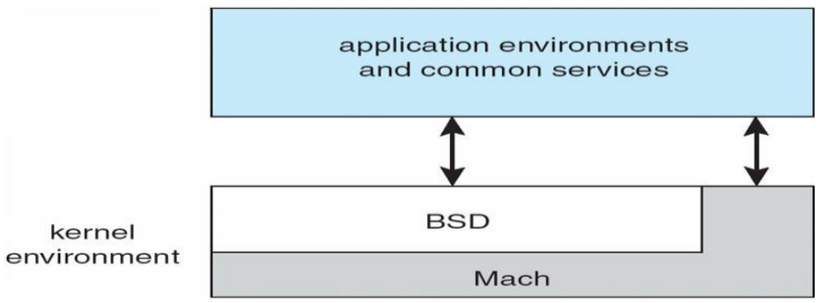
1. Scheduling classes
2. File systems



Solaris loadable modules.

1. Loadable system calls
2. Executable formats
3. STREAMS modules
4. Miscellaneous
5. Device and bus drivers

Such a design allows the kernel to provide core services yet also allowscertain features to be implemented dynamically. For example, device andbus drivers for specific hardware can be added to the kernel, and supportfor different file systems can be added as loadable modules. The overallresult resembles a layered system in that each kernel section has defined,protected interfaces; but it is more flexible than a layered system in that anymodule can call any other module. Furthermore, the approach is like themicrokernel approach in that the primary module has only core functionsand knowledge of how to load and communicate with other modules; but itis more efficient, because modules do not need to invoke message passing inorder to communicate.


application environments
and common services
BSD
Mach
kernel
environment

The Apple Macintosh Mac OS X operating system uses a hybrid structure.Mac OS X (also known as *Danvin)* structures the operating system using alayered technique where one layer consists of the Mach microkernel. Thestructure of Mac OS . The top layers include application environments and a set of servicesproviding a graphical interface to applications. Below these layers is the kernelenvironment, which consists primarily of the Mach microkernel and the BSDkernel. Mach provides memory management; support for remote procedurecalls (RPCs) and interprocess communication (IPC) facilities, including messagepassing; and thread scheduling. The BSD component provides a BSD commandline interface, support for networking and file systems. In addition to Mach and BSD, the kernelenvironment provides an I/O kit for development of device drivers anddynamically loadable modules (which Mac OS X refers to as **kernel extensions).**As shown in the figure, applications and common services can make use ofeither the Mach or BSD facilities directly

13. Operating-System Generation

It is possible to design, code, and implement an operating system specificallyfor one machine at one site. More commonly, however, operating systemsare designed to run on any of a class of machines at a variety of sites with a variety of peripheral configurations.

The system must then be configuredor generated for each specific computer site, a process sometimes known as **system generation** (SYSGEN).The operating system is normally distributed on disk or CD-ROM. Togenerate a system, we use a special program. The SYSGEN program reads froma given file, or asks the operator of the system for information concerning thespecific configuration of the hardware system, or probes the hardware directlyto determine what components are there. The following kinds of informationmust be determined.

• What CPU is to be used? What options (extended instruction sets, floatingpointarithmetic, and so on) are installed? For multiple CPU systems, eachCPU must be described.

• How much memory is available? Some systems will determine this valuethemselves by referencing memory location after memory location until an"illegal address" faul is generated. This procedure defines the final legaladdress and hence the amount of available memory.

• What devices are available? The system will need to know how to addresseach device (the device number), the device interrupt number, the device'stype and model, and any special device characteristics.

• What operating-system options are desired, or what parameter values areto be used? These options or values might include how many buffers ofwhich sizes should be used, what type of CPU-scheduling algorithm isdesired, what the maximum number of processes to be supported is, andso on.

Once this information is determined, it can be used in several ways. At one

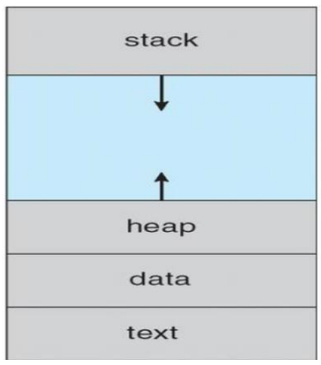
extreme, a system administrator can use it to modify a copy of the source code ofthe operating system. The operating system then is completely compiled. Datadeclarations, initializations, and constants, along with conditional compilation,produce an output object version of the operating system that is tailored to thesystem described.

Process Management

1. Process Concept

The Process

A process is a program in execution. A processis 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 **programcounter** and the contents of the processor's registers. A process generally alsoincludes the process **stack,** which contains temporary data (such as functionparameters, return addresses, and local variables), and a **data section,** whichcontains global variables. A process may also include a **heap,** which is memorythat is dynamically allocated during process run time. The structure of a process in memory is shown in Figure.



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 calledan **executable file),** whereas a process is an *active* entity, with a program counterspecifying the next instruction to execute and a set of associated resources. Aprogram becomes a process when an executable file is loaded into memory.

Process State

As a process executes, it changes **state.** The state of a process is defined inpart by the current activity of that process. Each process may be in one of thefollowing states:

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

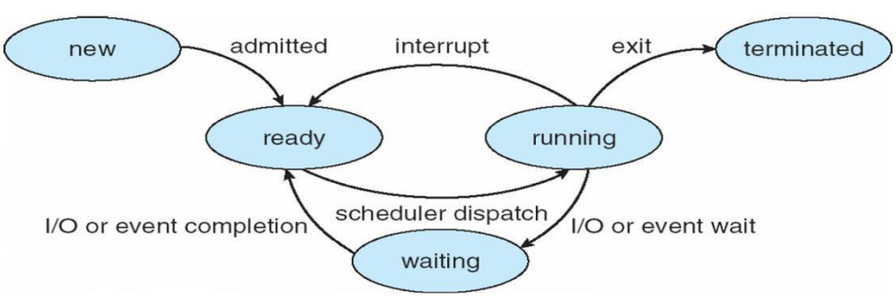
• **Running.** Instructions are being executed.

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

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

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

It is important to realizethat only one process can be *running* on any processor at any instant. Many processes may be *ready* and *limiting*, however. The state diagram correspondingto these states is presented in Figure.



Process Control Block

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

• **Process state.** The state may be new, ready, running, waiting, halted, andso on.

• **Program counter.** The counter indicates the address of the next instructionto be executed for this process.

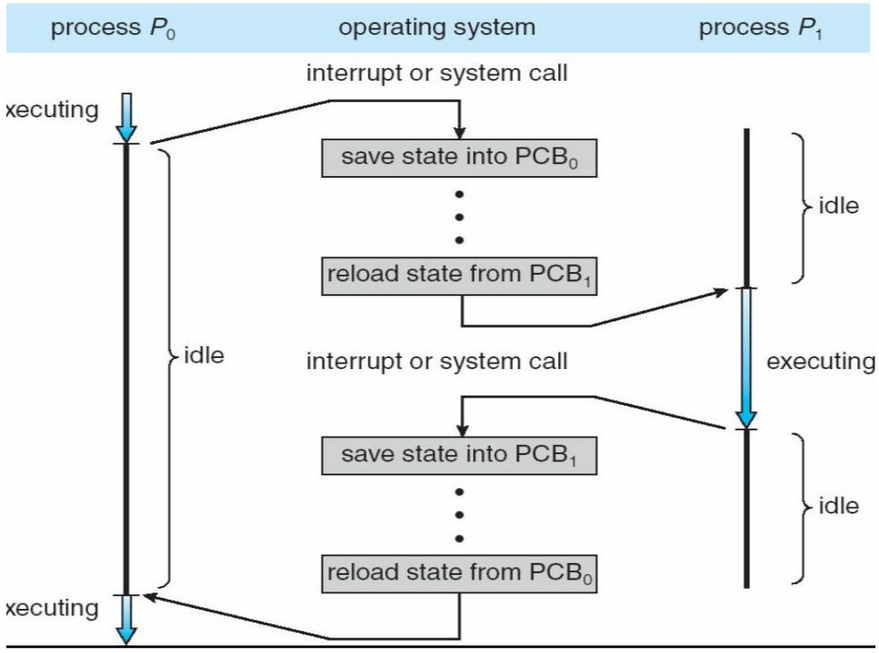
• CPU **registers.** The registers vary in number and type, depending onthe computer architecture. They include accumulators, index registers,stack pointers, and general-purpose registers, plus any condition-codeinformation. Along with the program counter, this state information mustbe saved when an interrupt occurs, to allow the process to be continuedcorrectly 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 suchinformation as the value of the base and limit registers, the page tables,or the segment tables, depending on the memory system used by the operating system.

• **Accounting information.** This information includes the amount of CPUand real time used, time limits, account mimbers, job or process numbers,and so on.

• **I/O status information.**This information includes the list of I/O devicesallocated to the process, a list of open files, and so on.In brief,the PCB simply serves as the repository for any information that mayvary from process to Process.



CPU Switch from Process to Process

2. Process Scheduling

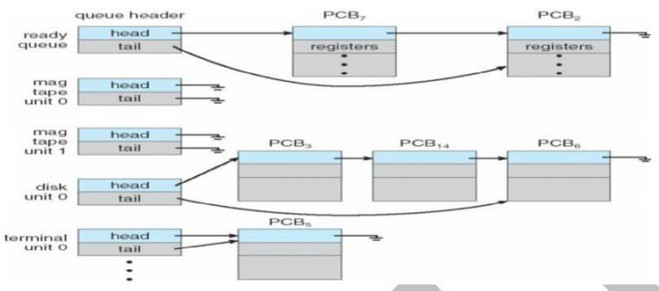
The objective of multiprogramming is to have some process running at alltimes, to maximize CPU utilization. The objective of time sharing is to switch theCPU among processes so frequently that users can interact with each programwhile it is running. To meet these objectives, the **process scheduler** selectsan available process (possibly from a set of several available processes) forprogram execution on the CPU. For a single-processor system, there will neverbe more than one running process. If there are more processes, the rest willhave to wait until the CPU is free and can be rescheduled.

Scheduling Queues

As processes enter the system, they are put into a **job queue,** which consistsof all processes in the system. The processes that are residing in main memoryand 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 containspointers to

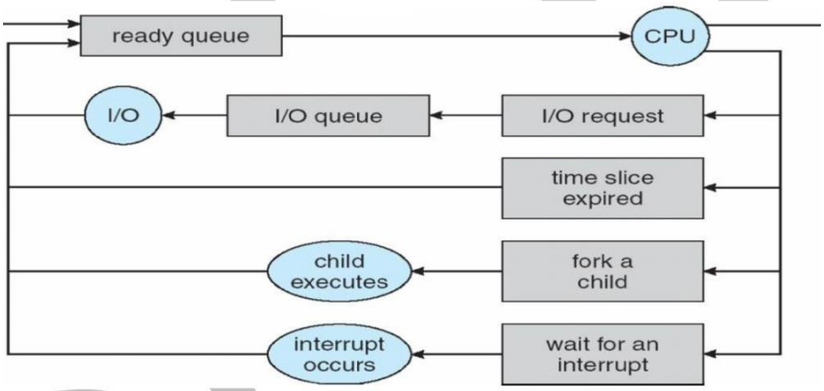
the first and final PCBs in the list. Each PCB includes a pointer fieldthat points to the next PCB in the ready queue.

The system also includes other queues. When a process is allocated theCPU, it executes for a while and eventually quits, is interrupted, or waits forthe occurrence of a particular event, such as the completion of an I/O request.Suppose the process makes an I/O request to a shared device, such as a disk.Since there are many processes in the system, the disk may be busy with theI/O request of some other process. The process therefore may have to wait forthe disk. The list of processes waiting for a particular I/O device is called adevice **queue.**
Each device has its own device queue (Figure).



The ready queue and various I/O device queues.

A common representation for a discussion of process scheduling is a**queueing diagram,** such as that in Figure . Each rectangular box representsa queue. Two types of queues are present: the ready queue and a set of devicequeues. The circles represent the resources that serve the queues, and thearrows indicate the flow of processes in the system.



Queueing-diagram representation of process scheduling.

A new process is initially put in the ready queue. It waits there tmtil it isselected for execution, or is **dispatched.** Once the process is allocated the CPUand is executing, one of several events could occur:

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

• The process could create a new subprocess and wait for the subprocess'stermination.

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

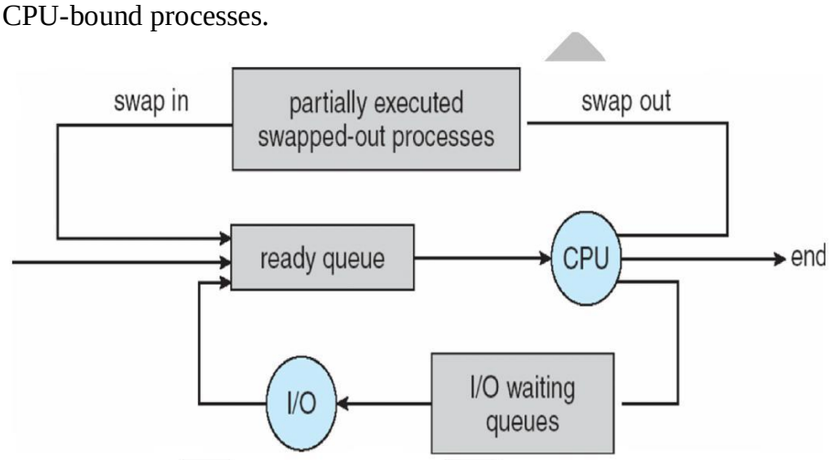
Schedulers

A process migrates among the various scheduling queues throughout itslifetime. The operating system must select, for scheduling purposes, processesfrom these queues in some fashion. The selection process is carried out by theappropriate **scheduler.**

The **long-term scheduler,** or **jobscheduler,** selects processes from this pool and loads them into memory forexecution. The **short-term scheduler, or** CPU **scheduler,** selects from amongthe processes that are ready to execute and allocates the CPU to one of them.

The primary distinction between these two schedulers lies in frequencyof execution. The short-term scheduler must select a new process for the CPUfrequently. A process may execute for only a few milliseconds before waitingfor an I/O request. Often, the short-term scheduler executes at least once every100 milliseconds. Because of the short time between executions, the short-termscheduler must be fast. The long-term scheduler executes much less frequently; minutes may separatethe creation of one new process and the next. The long-term schedulercontrols the **degree of multiprogramming**

It is important that the long-term scheduler make a careful selection. Ingeneral, most processes can be described as either L/O bound or CPU bound. An**I/O-bound process** is one that spends more of its time doing I/O than it spendsdoing computations. A **CPU-bound process,** in contrast, generates I/O requestsinfrequently, using more of its time doing computations. It is important that thelong-term scheduler select a good **process mix** of I/O-bound and CPU-bound processes.



Addition of medium-term scheduling to the queueing diagram.

If all processes are I/O bound, the ready queue will almost alwaysbe empty, and the short-term scheduler will have little to do. If all processesare CPU bound, the I/O waiting queue will almost always be empty, devices

will go unused, and again the system will be unbalanced. The system with thebest performance will thus have a combination of CPU-bound and I/O-boundprocesses.

Some operating systems, such as time-sharing systems, may introduce anadditional, intermediate level of scheduling. This **medium-term scheduler** isdiagrammed in Figure . The key idea behind a medium-term scheduler is

that sometimes it can be advantageous to remove processes from memory(and from active contention for the CPU) and thus reduce the degree ofmultiprogramming. Later, the process can be reintroduced into memory, and itsexecution can be continued where it left off. This scheme is called swapping.The process is swapped out, and is later swapped in, by the medium-termscheduler. Swapping may be necessary to improve the process mix or because

a change in memory requirements has overcommitted available memory,requiring memory to be freed up.

Context Switch

An interrupts cause the operating system to change a CPUfrom its current task and to run a kernel routine. Such operations happenfrequently on general-purpose systems. When an interrupt occurs, the systemneeds to save the current **context** of the process currently running on theCPU so that it can restore that context when its processing is done, essentiallysuspending the process and then resuming it.

The context is represented inthe PCB of the process; it includes the value of the CPU registers, the process

state (see Figure), and memory-management information. Generically, weperform a **state save** of the current state of the CPU, be it in kernel or user mode,and then a **state restore** to resume operations.

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

3. Operations on Processes

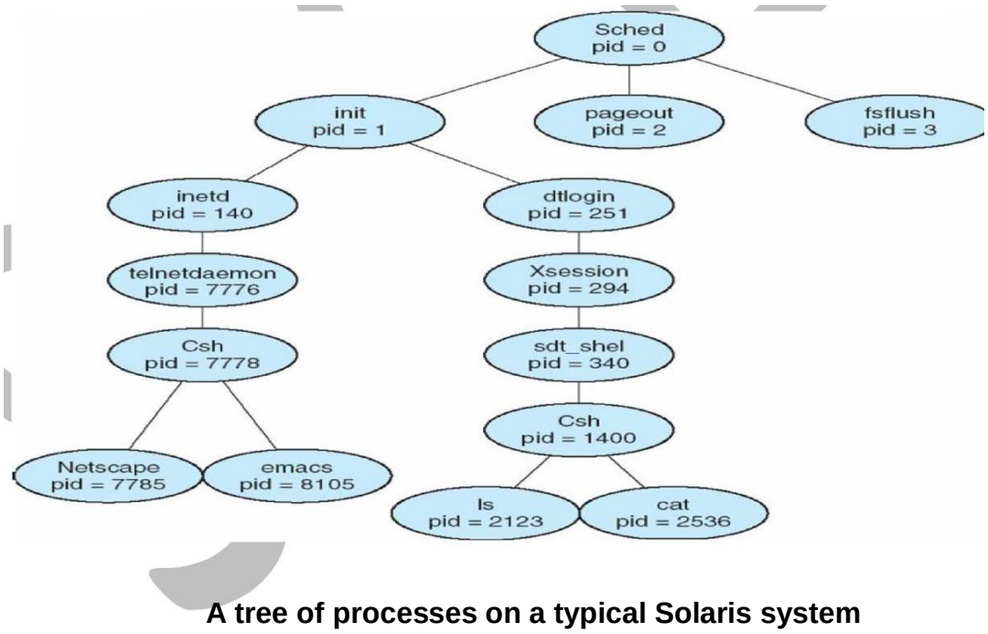
The processes in most systems can execute concurrently, and they may be created and deleted dynamically. Thus, these systems must provide a mechanism for process creation and termination.

Process Creation

A process may create several new processes, via a create-process system call, during the course of execution. The creating process is called a **parent** process, and the new processes are called the **children** of that process. Each of these

new processes may in turn create other processes, forming a **tree** of processes.

Most operating systems identify processes according to a unique **process identifier** (or **pid),** which is typically an integer number. Figure illustrates a typical process tree for the Solaris operating system, showing the name of each process and its pid. In Solaris, the process at the top of the tree is the sched process, with pid of 0. The sched process creates several children processes—including pageout and f sf lush. These processes are responsible for managing memory and file systems. The sched process also creates the i n i t process, which serves as the root parent process for all user processes.



A tree of processes on a typical Solaris system

In general, a process will need certain resources (CPU time, memory, files, I/O devices) to accomplish its task. When a process creates a subprocess, that subprocess may be able to obtain its resources directly from the operatiiigsystem, or it may be constrained to a subset of the resources of the parent process. The parent may have to partition its resources among its children, or it may be able to share some resources (such as memory or files) among several of its children. Restricting a child process to a subset of the parent's resources prevents any process from overloading the system by creating too many subprocesses.

When a process creates a new process, two possibilities exist in terms ofexecution:

3. The parent continues to execute concurrently with its children.
4. The parent waits until some or all of its children have terminated.

There are also two possibilities in terms of the address space of the new process:

6. The child process is a duplicate of the parent process (it has the sameprogram and data as the parent).

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

Process Termination

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

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

• The child has exceeded its usage of some of the resources that it has beenallocated. (To determine whether this has occurred, the parent must havea mechanism to inspect the state of its children.)

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

• The parent is exiting, and the operating system does not allow a child tocontinue if its parent terminates.

Some systems, including VMS, do not allow a child to exist if its parenthas terminated. In such systems, if a process terminates (either normally orabnormally), then all its children must also be terminated. This phenomenon,referred to as cascading termination, is normally initiated by the operatingsystem.

To illustrate process execution and termination, consider that, in UNIX, wecan terminate a process by using the

e x i t() system call; its parent processmay wait for the termination of a child process by using the waitO systemcall. The wait () system call returns the process identifier of a terminated childso that the parent can tell which of its possibly many children has terminated.

4. Interprocess Communication

Processes executing concurrently in the operating system may be either independent processes or cooperating processes. A process is **independent** if it cannot affect or be affected by the other processes executing in the system.Any process that does not share data with any other process is independent.

A process is **cooperating** if **it** can affect or be affected by the other processes executing in the system. Clearly, any process that shares data with other processes is a cooperating process.

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

• **Information sharing.** Since several users may be interested in the same piece of information (for instance, a shared file), we must provide an environment to allow concurrent access to such information.

• **Computation speedup.** If we want a particular task to run faster, we must break it into subtasks, each of which will be executing in parallel with the others.

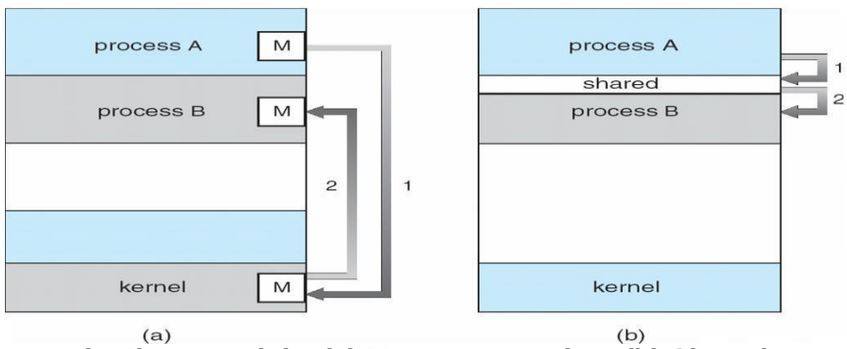
• **Modularity.** We may want to construct the system in a modular fashion, dividing the system functions into separate processes or threads.

• **Convenience.** Even an individual user may work on many tasks at the same time. For instance, a user may be editing, printing, and compiling in parallel.

Cooperating processes require an **interprocess communication (IPC)** mechanism that will allow them to exchange data and information. There are two fundamental models of interprocess communication: **(1) shared memory** and (2) **message passing.**

In the shared-memory model, a region of memory that is shared by cooperating processes is established. Processes can then exchange information by reading and writing data to the shared region. In the

messagepassingmodel, communication takes place by means of messages exchanged between the cooperating processes. The two communications models are contrasted in Figure.



Communications models, (a) Message passing, (b) Shared memory.

Both of the models just discussed are common in operating systems, and many systems implement both. Message passing is useful for exchanging smaller amounts of data, because no conflicts need be avoided. Message passing is also easier to implement than is shared memory for intercomputer communication. Shared memory allows maximum speed and convenience of communication, as it can be done at memory speeds when within a computer.

Shared memory is faster than message passing, as message-passing systemsare typically implemented using system calls and thus require the more timeconsumingtask of kernel intervention. In contrast, in shared-memory systems,

system calls are required only to establish shared-memory regions. Once sharedmemory is established, all accesses are treated as routine memory accesses, andno assistance from the kernel is required.

Shared-Memory Systems

Interprocess communication using shared memory requires communicatingprocesses to establish a region of shared memory. Typically, a shared-memoryregion resides in the address space of the process creating the shared-memorysegment. Other processes that wish to communicate using this shared-memorysegment must attach it to their address space. They can then exchange information by reading and writingdata in the shared areas. The form of the data and the location are determined bythese processes and are not under the operating system's control. The processes

are also responsible for ensuring that they are not writing to the same locationsimultaneously.

To illustrate the concept of cooperating processes, let's consider theproducer-consumer problem, which is a common paradigm for cooperatingprocesses. A **producer** process produces information that is consumed by a**consumer** process.

Message-Passing Systems

Message passing provides a mechanism to allow processes to communicate and to synchronize their actions without sharing the same address space and is particularly useful in a distributed environment, where the communicating

processes may reside on different computers connected by a network.

A message-passing facility provides at least two operations: send(message) and receive(message). Messages sent by a process can be of either fixed or variable size. If only fixed-sized messages can be sent, the system-level implementation is straightforward. This restriction, however, makes the task of programming more difficult. Conversely, variable-sized messages require a more complex system-level implementation, but the programming task becomes simpler. This is a common kind of tradeoff seen throughout operating system design.

If processes *P* and *Q* want to communicate, they must send messages to and receive messages from each other; a **communication link** must exist between them. This link can be implemented in a variety of ways. Here are several methods for logically implementing a link and the send()/receive () operations:

• Direct or indirect communication

• Synchronous or asynchronous communication

• Automatic or explicit buffering

We look at issues related to each of these features next.

Naming

Processes that want to communicate must have a way to refer to each other. They can use either direct or indirect communication.

Under direct communication, each process that wants to communicate must explicitly name the recipient or sender of the communication. In this scheme, the send.0 and receive() primitives are defined as:

• send(P, message)—Send a message to process P.

• receive (Q, message)—Receive a message from process Q.

A communication link in this scheme has the following properties:

• A link is established automatically between every pair of processes that want to communicate. The processes need to know only each other's identity to communicate.

• A link is associated with exactly two processes.

• Between each pair of processes, there exists exactly one link.

This scheme exhibits *symmetry* in addressing; that is, both the sender process and the receiver process must name the other to communicate. A variant of this scheme employs *asymmetry* in addressing. Here, only the sender names the recipient; the recipient is not required to name the sender. In this scheme, the send() and receive () primitives are defined as follows:

• send(P, message)—Send a message to process P.

• receive(id, message)—-Receive a message from any process; the variable *id* is set to the name of the process with which communication has taken place.

With indirect communication, the messages are sent to and received from mailboxes, or ports. A mailbox can be viewed abstractly as an object into which messages can be placed by processes and from which messages can be removed.

Each mailbox has a unique identification.

Two processes can communicate only if the processes have a shared mailbox, however. The sendC) and receive () primitives are defined as follows:

• send(A, message)—Send a message to mailbox A.

• receive(A, message)—Receive a message from mailbox A.

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

• A link is established between a pair of processes only if both members of the pair have a shared mailbox.

• A link may be associated with more than two processes.

• Between each pair of communicating processes, there may be a number of different links, with each link corresponding to one mailbox.

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

• Create a new mailbox.

• Send and receive messages through the mailbox.

• Delete a mailbox.

The process that creates a new mailbox is that mailbox's owner by default.Initially, the owner is the only process that can receive messages through thismailbox. However, the ownership and receiving privilege may be passed toother processes through appropriate system calls. Of course, this provisioncould result in multiple receivers for each mailbox.

Synchronization

Communication between processes takes place through calls to sendO andreceive () primitives. There are different design options for implementingeach primitive. Message passing may be either **blocking** or **nonblocking**also known as **synchronous** and **asynchronous.**

• **Blocking send.** The sending process is blocked until the message isreceived by the receiving process or by the mailbox.

• **Nonblocking send.** The sending process sends the message and resumesoperation.

• **Blocking receive.** The receiver blocks until a message is available.

• **Nonblocking receive.** The receiver retrieves either a valid message or anull.

Buffering

Whether communication is direct or indirect, messages exchanged by communicatingprocesses reside in a temporary queue. Basically, such queues can beimplemented in three ways:

• **Zero capacity.** The queue has a maximum length of zero; thus, the linkcannot have any messages waiting in it. In this case, the sender must blockuntil the recipient receives the message.

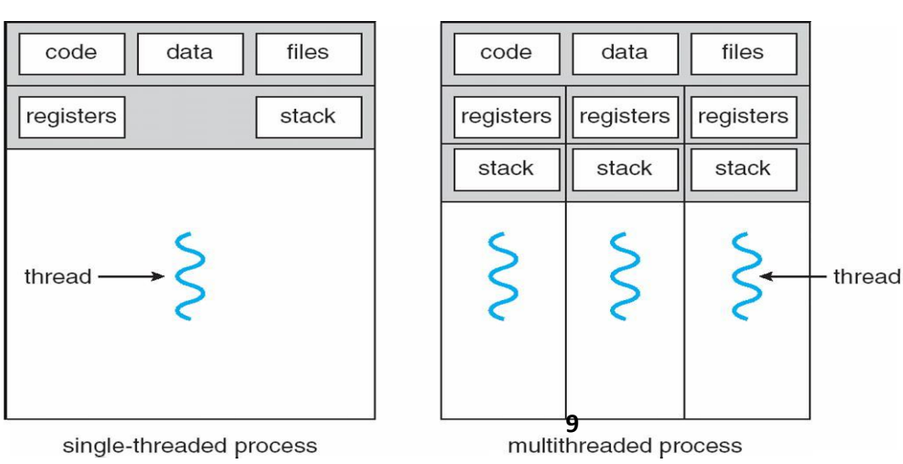
• **Bounded capacity.** The queue has finite length *n;* thus, at most *n* messagescan reside in it. If the queue is not full when a new message is sent, themessage is placed in the queue (either the message is copied or a pointerto the message is kept), and the sender can continue execution withoutwaiting. The links capacity is finite, however. If the link is full, the sendermust block until space is available in the queue.

• **Unbounded capacity.** The queues length is potentially infinite; thus, anynumber of messages can wait in it. The sender never blocks.

The zero-capacity case is sometimes referred to as a message system with nobuffering; the other cases are referred to as systems with automatic buffering.

5. Overview of Threads

A thread is a basic unit of CPU utilization; it comprises a thread ID, a programcounter, a register set, and a stack. It shares with other threads belongingto the same process its code section, data section, and other operating-systemresources, such as open files and signals. A traditional (or **heavyweight)** processhas a single thread of control. process has multiple threads of control, itcan perform more than one task at a time. Figure illustrates the differencebetween a traditional **single-threaded** process and a **multithreaded** process.



single-threaded process multithreaded process

Single-threaded and multithreaded processes.

The benefits of multithreaded programming can be broken down into fourmajor categories:

1. Responsiveness. Multithreading an interactive application may allow aprogram to continue running even if part of it is blocked or is performinga lengthy operation, thereby increasing responsiveness to the user. Forinstance, a multithreaded web browser could still allow user interactionin one thread while an image was being loaded in another thread.

2. Resource sharing. By default, threads share the memory and theresources of the process to which they belong. The benefit of sharingcode and data is that it allows an application to have several differentthreads of activity within the same address space.

3. Economy. Allocating memory and resources for process creation is costly.Because threads share resources of the process to which they belong, itis more economical to create and context-switch threads. Empiricallygauging the difference in overhead can be difficult, but in general it ismuch more time consuming to create and manage processes than threads.

4. Utilization of multiprocessor architectures. The benefits of multithreadingcan be greatly increased in a multiprocessor architecture, wherethreads may be running in parallel on different processors. A singlethreaded process can only run on one CPU, no matter how many areavailable. Multithreading on a multi-CPU machine increases concurrency.

Multithreading Models

Supportfor threads may be provided either at the user level, for **user threads,** or by thekernel, for **kernel threads.** User threads are supported above the kernel andare managed without kernel support, whereas kernel threads are supportedand managed directly by the operating system.

Ultimately, there must exist a relationship between user threads and kernelthreads. In this section, we look at three common ways of establishing thisrelationship.

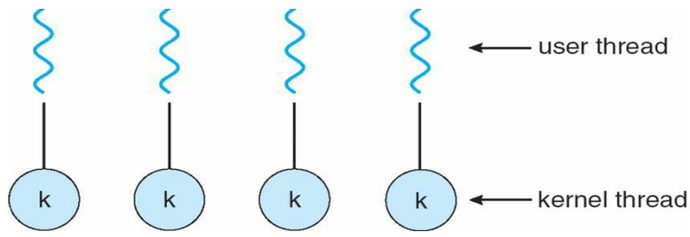
Many-to-One Model

The many-to-one model maps many user-level threads to onekernel thread. Thread management is done by the thread library in userspace, so it is efficient; but the entire process will block if a thread makes ablocking system call. Also, because only one thread can access the kernel at atime, multiple threads are unable to run in parallel on multiprocessors. **Greenthreads**—a thread library available for Solaris—uses this model, as does GNU**Portable Threads.**

One-to-One Model

The one-to-one model maps each user thread to a kernel thread. Itprovides more concurrency than the many-to-one model by allowing anotherthread to run when a thread makes a blocking system call; it also allowsmultiple threads to run in parallel on multiprocessors. The only drawback tothis model is that creating a user thread requires creating the correspondingkernel thread. Because the overhead of creating kernel threads can burden

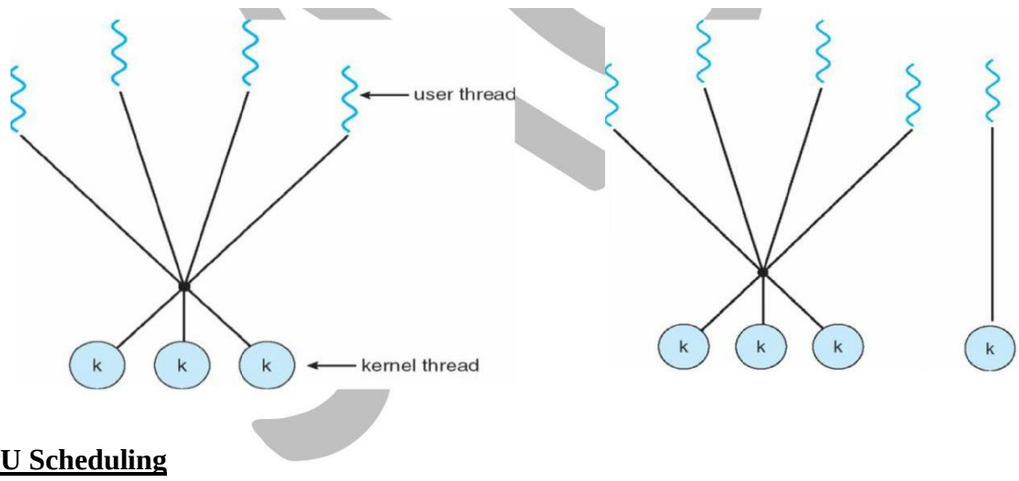
theperformance of an application, most implementations of this model restrict thenumber of threads supported by the system. Linux, along with the family ofWindows operating systems—including Windows 95, 98, NT, 2000, and XPimplement the one-to-one model.



Many-to-Many Model

The many-to-many model multiplexes many user-level threads toa smaller or equal number of kernel threads. The number of kernel threadsmay be specific to either a particular application or a particular machine (aon a uniprocessor). Whereas the many-to-one model allows the developer tocreate as many user threads as she wishes, true concurrency is not gainedbecause the kernel can schedule only one thread at a time. The one-to-onemodel allows for greater concurrency, but the developer has to be careful notto create too many threads within an application (and in some instances maybe limited in the number of threads she can create). The many-to-many modelsuffers from neither of these shortcomings: Developers can create as many userthreads as necessary, and the corresponding kernel threads can run in parallelon a multiprocessor. Also, when a thread performs a blocking system call, thekernel can schedule another thread for execution.

One popular variation on the many-to-many model still multiplexes manyuser-level threads to a smaller or equal number of kernel threads but also allowsa user-level thread to be bound to a kernel thread. This variation, sometimesreferred to as the *tivo-level model* (Figure), is supported by operating systemssuch as IRIX, HP-UX, and Tru64 UNIX. The Solaris operating system supportedthe two-level model .



6. CPU Scheduling

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

1. When a process switches from the running state to the waiting state
2. When a process switches from the running state to the ready state
3. When a process switches from the waiting state to the ready state
4. When a process terminates

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

When scheduling takes place only under circumstances 1 and 4, we saythat the scheduling scheme is **nonpreemptive** or **cooperative;** otherwise, itis **preemptive.** Under nonpreemptive scheduling, once the CPU has beenallocated to a process, the process keeps the CPU until it releases the CPU eitherby terminating or by

switching to the waiting state. This scheduling methodwas used by Microsoft Windows 3.x; Windows 95 introduced preemptivescheduling, and all subsequent versions of Windows operating systems haveused preemptive scheduling.

Unfortunately, preemptive scheduling incurs a cost associated with accessto shared data. Consider the case of two processes that share data. While oneis updating the data, it is preempted so that the second process can run. Thesecond process then tries to read the data, which are in an inconsistent state.

Dispatcher

Another component involved in the CPU-scheduling function is the **dispatcher.**The dispatcher is the module that gives control of the CPU to the process selectedby the short-term scheduler. This function involves the following:

• Switching context

• Switching to user mode

• Jumping to the proper location in the user program to restart that program

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

Scheduling Criteria

Different CPU scheduling algorithms have different properties, and the choiceof a particular algorithm based onMany criteria have been suggested for comparing CPU scheduling algorithms.The criteria include thefollowing:

• **CPU utilization.** We want to keep the CPU as busy as possible. Conceptually,CPU utilization can range from 0 to 100 percent. In a real system, itshould range from 40 percent (for a lightly loaded system) to 90 percent.

• **Throughput.** One measure of CPU work is the number of processes that are completedper time unit, called *throughput.* For long processes, this rate may be oneprocess per hour; for short transactions, it may be 10 processes per second.

• **Turnaround time. It is** how long it takes to execute that process. The intervalfrom the time of submission of a process to the time of completion is the*turnaround time.* Turnaround time is the sum of the periods spent waitingto get into memory, waiting in the ready queue, executing on the CPU, anddoing I/O.

• **Waiting time.** Theamount of time that a process spends waiting in the ready queue. *Waitingtime* is the sum of the periods spent waiting in the ready queue.

• **Response time. T**he time from the submissionof a request until the first response is produced. This measure, called*response time,* is the time it takes to start responding, not the time it takesto output the response. The turnaround time is generally limited by thespeed of the output device.

It is desirable to maximize CPU utilization and throughput and to minimizeturnaround time, waiting time, and response time. In most cases, we optimizethe average measure. However, under some circumstances, it is desirableto optimize the minimum or maximum values rather than the average.

Scheduling Algorithms

CPU scheduling deals with the problem of deciding which of the processesin the ready queue is to be allocated the CPU. There are many different CPUscheduling algorithms. In this section, we describe several of them.

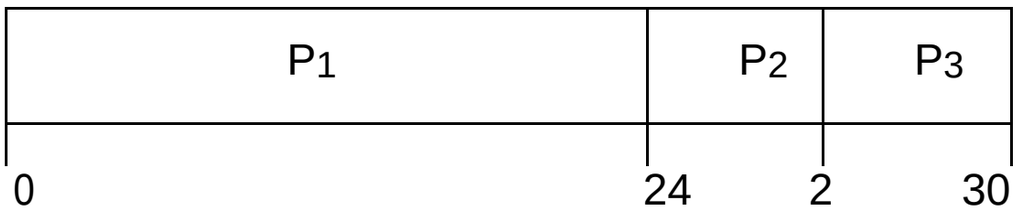
1. First-Come, First-Served Scheduling

The simplest CPU-scheduling algorithm is the **first-come, first-served(FCFS) scheduling algorithm.** With this scheme, the process that requests theCPU first is allocated the CPU first. The implementation of the

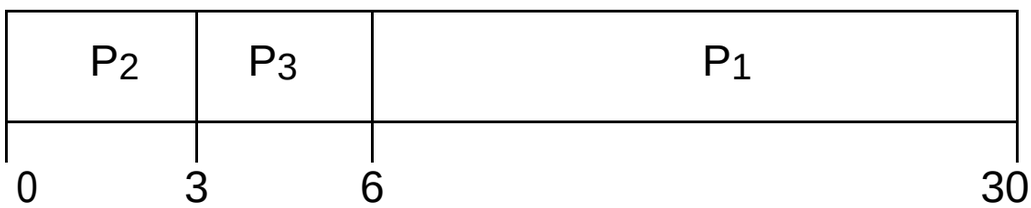
FCFS policy iseasily managed with a FIFO queue. When a process enters the ready queue, itsPCB is linked onto the tail of the queue. When the CPU is free, it is allocated tothe process at the head of the queue. The running process is then removed fromthe queue. The code for FCFS scheduling is simple to write and understand.The average waiting time under the FCFS policy, however, is often quitelong. Consider the following set of processes that arrive at time 0, with thelength of the CPU burst given in milliseconds:

Process	Burst Time
P1	24
P2	**3**
P3	3

If the processes arrive in the order P1, *P2, P3,* and are served in FCFS order,we get the result shown in the following **Gantt chart:**



The waiting time is 0 milliseconds for process Pi, 24 milliseconds for process *Pn,* and 27 milliseconds for process *Pj.* Thus, the average waiting time is (0 + 24 + 27)/3 = 17 milliseconds. If the processes arrive in the order *Pi,* P3, Pi, however, the results will be as shown in the following Gantt chart:



The average waiting time is now (6 + 0 + 3)/3 = 3 milliseconds. This reductionis substantial. Thus, the average waiting time under an FCFS policy is generallynot minimal and may vary substantially if the process's CPU burst times varygreatly.

In addition, consider the performance of FCFS scheduling in a dynamicsituation. Assume we have one CPU-bound process and many I/O-boundprocesses. As the processes flow around the system, the following scenariomay result. The CPU-bound process will get and hold the CPU. During thistime, all the other processes will finish their I/O and will move into the readyqueue, waiting for the CPU. While the processes wait in the ready queue, theI/O devices are idle. Eventually, the CPU-bound process finishes its CPU burstand moves to an I/O device. All the I/O-bound processes, which have shortCPU bursts, execute quickly and move back to the I/O queues. At this point,the CPU sits idle. The CPU-bound process will then move back to the readyqueue and be allocated the CPU. Again, all the I/O processes end up waiting inthe ready queue until the CPU-bound process is done.

There is a **convoy effect**as all the other processes wait for the one big process to get off the CPU. Thiseffect results in lower CPU and device utilization than might be possible if theshorter processes were allowed to go first.

The FCFS scheduling algorithm is nonpreemptive. Once the CPU has beenallocated to a process, that process keeps the CPU until it releases the CPU, eitherby terminating or by requesting I/O. The FCFS algorithm

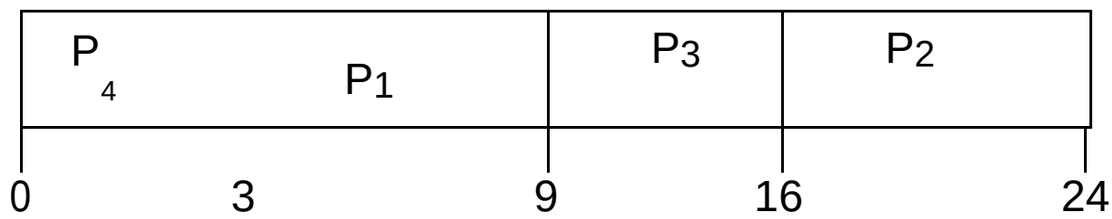
is thus particularlytroublesome for time-sharing systems, where it is important that each user geta share of the CPU at regular intervals.

2. Shortest-Job-First Scheduling

The **shortest-job-first (SJF) schedulingalgorithm** associates with each process the length of theprocess's next CPU burst. When the CPU is available, it is assigned to the processthat has the smallest next CPU burst. If the next CPU bursts of two processes arethe same, FCFS scheduling is used to break the tie. Note that a more appropriateterm for this scheduling method would be the *shortest-next-CPU-burst algorithm,*because scheduling depends on the length of the next CPU burst of a process,rather than its total length.As an example of SJF scheduling, consider the following set of processes,with the length of the CPU burst given in milliseconds:

Process	Burst Time
P1	6
P2	8
P3	7
P4	3

Using SJF scheduling, we would schedule these processes according to thefollowing Gantt chart:



The waiting time is 3 milliseconds for process *P1,* 16 milliseconds for process*P2,* 9 milliseconds for process *P3,* and 0 milliseconds for process P4. Thus, theaverage waiting time is (3 + 16 + 9 + 0)/4 - *7* milliseconds. By comparison, ifwe were using the FCFS scheduling scheme, the average waiting time wouldbe 10.25 milliseconds.

The SJF scheduling algorithm is provably *optimal,* in that it gives theminimum average waiting time for a given set of processes. Moving a shortprocess before a long one decreases the waiting time of the short process morethan it increases the waiting time of the long process. Consequently, the *average*waiting time decreases.

The real difficulty with the SJF algorithm is knowing the length of the nextCPU request. There is no way to know the length of the nextCPU burst. One approach is to try to approximate SJF scheduling. We may not*know* the length of the next CPU burst, but we may be able to *predict* its value.We expect that the next CPU burst will be similar in length to the previous ones.Thus, by computing an approximation of the length of the next CPU burst, wecan pick the process with the shortest predicted CPU burst.The next CPU burst is generally predicted as an exponential average of themeasured lengths of previous CPU bursts. Let *tn* be the length of the »th CPUburst, and let T,,+I be our predicted value for the next CPU burst. Then, for a, 0< a < 1, define

$$T_{n+1} = a t_n + (1 - a) T_n.$$

This formula defines an **exponential average.** The value of *tn* contains ourmost recent information; *in* stores the past history. The parameter *a* controlsthe relative weight of recent and past history in our prediction. If a = 0, thenTn,+I =Tn,, and recent history has no effect (current conditions are assumedto be transient); if a = 1, then Tn+1= *tn,* and only the most recent CPU burstmatters (history is assumed to be old and irrelevant). More

commonly, a =1/2, so recent history and past history are equally weighted. The initial T0 canbe defined as a constant or as an overall system average.

The SJF algorithm can be either preemptive or nonpreemptive. The choicearises when a new process arrives at the ready queue while a previous process isstill executing. The next CPU burst of the newly arrived process may be shorterthan what is left of the currently executing process. A preemptive SJF algorithmwill preempt the currently executing process, whereas a nonpreemptiTe SJFalgorithm will allow the currently running process to finish its CPU burst.Preemptive SJF scheduling is sometimes called **shortest-remaining-time-firstscheduling.**

As an example, consider the following four processes, with the length ofthe CPU burst given in milliseconds:

Process	Arrival Time	Burst Time
P1	0	8
P2	1	4
P3	2	9
P4	3	5

If the processes arrive at the ready queue at the times shown and need theindicated burst times, then the resulting preemptive SJF schedule is as depictedin the following Gantt chart:

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

3 Priority Scheduling

The SJF algorithm is a special case of the general **priority scheduling algorithm.**A priority is associated with each process, and the CPU is allocated to the processwith the highest priority. Equal-priority processes are scheduled in FCFS order.An SJF algorithm is simply a priority algorithm where the priority (p) is theinverse of the (predicted) next CPU burst. The larger the CPU burst, the lowerthe priority, and vice versa.

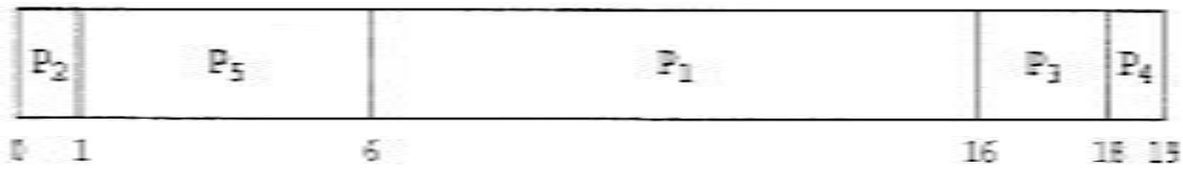
Note that we discuss scheduling in terms of *high* priority and *low* priority.Priorities are generally indicated by some fixed range of numbers, such as 0to 7 or 0 to 4,095. However, there is no general agreement on whether 0 is thehighest or lowest priority. Some systems use low numbers to represent lowpriority; others use low numbers for high priority. This difference can lead toconfusion. In this text, we assume that low numbers represent high priority.

As an example, consider the following set of processes, assumed to havearrived at time 0, in the order P1, P2, • • -, *P5,* with the length of the CPU burstgiven in milliseconds

Process	*Ps*	2
Pi		1
Pi	**Burst Time**	5
P3	10	
PA	1	**Priority**

3	4	2
1	5	

Using priority scheduling, we would schedule these processes according to thefollowing Gantt chart:



The average waiting time is 8.2 milliseconds.

Priorities can be defined either internally or externally. Internally definedpriorities use some measurable quantity or quantities to compute the priorityof a process

Priority scheduling can be either preemptive or nonpreemptive. When aprocess arrives at the ready queue, its priority is compared with the priorityof the currently running process. A preemptive priority scheduling algorithmwill preempt the CPU if the priority of the newly arrived process is higherthan the priority of the currently running process. A nonpreemptive priorityscheduling algorithm will simply put the new process at the head of the readyqueue.

A major problem with priority scheduling algorithms is **indefinite blocking,**or **starvation.** A process that is ready to run but waiting for the CPU canbe considered blocked. A priority scheduling algorithm can leave some lowpriorityprocesses waiting indefinitely. In a heavily loaded computer system, asteady stream of higher-priority processes can prevent a low-priority processfrom ever getting the CPU. Generally, one of two things will happen. Either theprocess will eventually be run or the computer system will eventually crash and lose allunfinished low-priority processes.

A solution to the problem of indefinite blockage of low-priority processesis **aging.** Aging is a technique of gradually increasing the priority of processesthat wait in the system for a long time. For example, if priorities range from127 (low) to 0 (high), we could increase the priority of a waiting process by1 every 15 minutes. Eventually, even a process with an initial priority of 127would have the highest priority in the system and would be executed. In fact,it would take no more than 32 hours for a priority-127 process to age to apriority-0 process.

4 Round-Robin Scheduling

The **round-robin (RR) scheduling algorithm** is designed especially for timesharingsystems. It is similar to FCFS scheduling, but preemption is added toswitch between processes. A small unit of time, called a **time quantum** or timeslice, is defined. A time quantum is generally from 10 to 100 milliseconds. Theready queue is treated as a circular queue. The CPU scheduler goes around theready queue, allocating the CPU to each process for a time interval of up to 1time quantum.

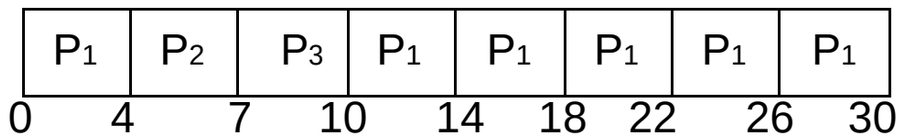
To implement RR scheduling, we keep the ready queue as a FIFO queue ofprocesses. New processes are added to the tail of the ready queue. The CPUscheduler picks the first process from the ready queue, sets a timer to interruptafter 1 time quantum, and dispatches the process.One of two things will then happen.

The process may have a CPU burst ofless than 1 time quantum. In this case, the process itself will release the CPUvoluntarily. The scheduler will then proceed to the next process in the readyqueue. Otherwise, if the CPU burst of the currently running process is longerthan 1 time quantum, the timer will go off and will cause an interrupt to theoperating system. A context switch will be executed, and the process will beput at the **tail** of the ready queue. The CPU scheduler will then select the nextprocess in the ready queue.

The average waiting time under the RR policy is often long. Consider thefollowing set of processes that arrive at time 0, with the length of the CPU burstgiven in milliseconds:

Process	Burst Time
Pi	24
Pi	3
P3	3

If we use a time quantum of 4 milliseconds, then process P1 gets the first4 milliseconds. Since it requires another 20 milliseconds, it is preempted afterthe first time quantum, and the CPU is given to the next process in the queue,process *P2.* Since process P2 does not need 4 milliseconds, it quits before itstime quantum expires. The CPU is then given to the next process, process P3.Once each process has received 1 time quantum, the CPU is returned to processP1 for an additional time quantum. The resulting RR schedule is



The average waiting time is 17/3 = 5.66 milliseconds.

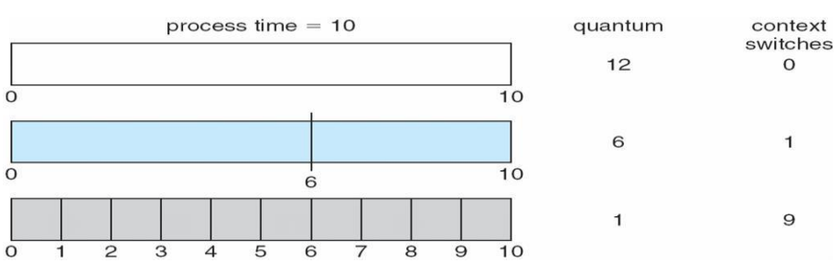
In the RR scheduling algorithm, no process is allocated the CPU for morethan 1 time quantum in a row (unless it is the only runnable process). If aprocess's CPU burst exceeds 1 time quantum, that process is *preempted* and isput back in the ready queue. The RR scheduling algorithm is thus preemptive.If there are *n* processes in the ready queue and the time quantum is *q*,then each process gets *1/n* of the CPU time in chunks of at most *q* time units.

Each process must wait no longer than *(n* — 1) x *q* time units until itsnext time quantum. For example, with five processes and a time quantum of 20milliseconds, each process will get up to 20 milliseconds every 100 milliseconds.

The performance of the RR algorithm depends heavily on the size of thetime quantum. At one extreme, if the time quantum is extremely large, the RRpolicy is the same as the FCFS policy If the time quantum is extremely small(say, 1 millisecond), the RR approach is called **processor sharing** and (in theory)creates the appearance that each of *n* processes has its own processor runningat *1/n* the speed of the real processor.

In software, we need also to consider the effect of context switching on theperformance of RR scheduling. Let us assume that we have only one process of10 time units. If the quantum is 12 time units, the process finishes in less than 1time quantum, with no overhead. If the quantum is 6 time units, however, theprocess requires 2 quanta, resulting in a context switch. If the time quantum is1 time unit, then nine context switches will occur, slowing the execution of theprocess accordingly.

Thus, we want the time quantum to be large with respect to the contextswitchtime. If the context-switch time is approximately 10 percent of thetime quantum, then about 10 percent of the CPU time will be spent in contextswitching. In practice, most modern systems have time quanta ranging from10 to 100 milliseconds. The time required for a context switch is typically lessthan 10 microseconds; thus, the context-switch time is a small fraction of thetime quantum.



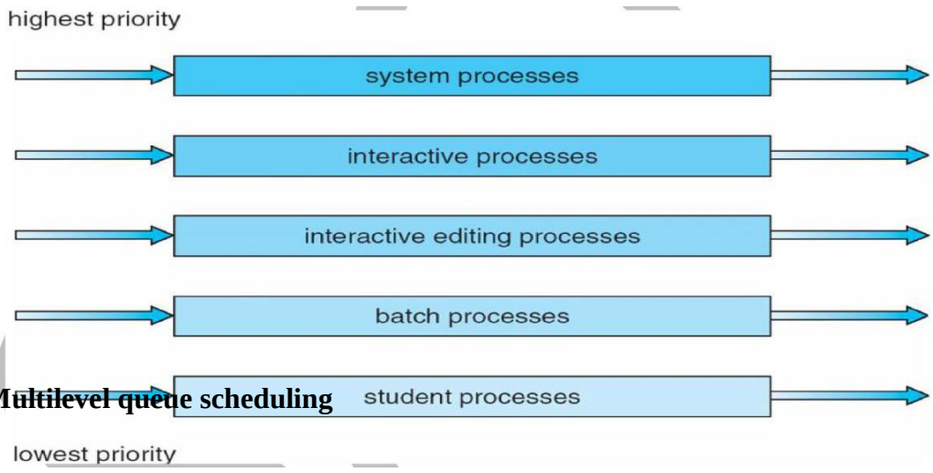
Time Quantum and Context Switch Time

Turnaround time also depends on the size of the time quantum. As we cansee from Figure, the average turnaround time of a set of processes doesnot necessarily improve as the time-quantum size increases. In general, theaverage turnaround time can be improved if most processes finish their nextCPU burst in a single time quantum. For example, given three processes of 10time units each and a quantum of 1 time unit, the average turnaround time is29. If the time quantum is 10, however, the average turnaround time drops to20. If context-switch time is added in, the average turnaround time increasesfor a smaller time quantum, since more context switches are required.Although the time quantum should be large compared with the contextswitchtime, it should not be too large. If the time quantum is too large, RRscheduling degenerates to FCFS policy. A rule of thumb is that 80 percent of theCPU bursts should be shorter than the time quantum.

5 Multilevel Queue Scheduling

Another class of scheduling algorithms has been created for situations inwhich processes are easily classified into different groups. For example, acommon division is made between **foreground** (interactive) processes and**background** (batch) processes. These two types of processes have differentresponse-time requirements and so may have different scheduling needs. Inaddition, foreground processes may have priority (externally defined) overbackground processes.

A **multilevel queue scheduling algorithm** partitions the ready queue intoseveral separate queues (Figure 5.6). The processes are permanently assigned toone queue, generally based on some property of the process, such as memorysize, process priority, or process type. Each queue has its own scheduling algorithm.



Multilevel queue scheduling

For example, separate queues might be used for foreground andbackground processes. The foreground quetie might be scheduled by an RRalgorithm, while the background queue is scheduled by an FCFS algorithm.In addition, there must be scheduling among the queues, which is commonlyimplemented as fixed-priority preemptive scheduling. For example, theforeground queue may have absolute priority over the background queue.Let's look at an example of a multilevel queue scheduling algorithm withfive queues, listed below in order of priority:

1. System processes
2. Interactive processes
3. Interactive editing processes
4. Batch processes
5. Student processes

Each queue has absolute priority over lower-priority queues. No process in thebatch queue, for example, could run unless the queues for system processes,interactive processes, and interactive editing processes were

all empty. If aninteractive editing process entered the ready queue while a batch process wasrunning, the batch process would be preempted.

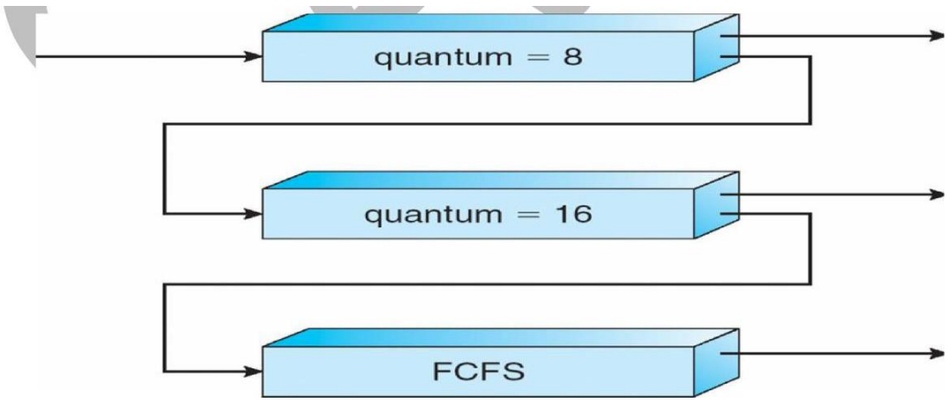
6 Multilevel Feedback-Queue Scheduling

Normally, when the multilevel queue scheduling algorithm is used, processesare permanently assigned to a queue when they enter the system. If thereare separate queues for foreground and background processes, for example,processes do not move from one queue to the other, since processes do notchange their foreground or background nature. This setup has the advantageof low scheduling overhead, but it is inflexible.

The **multilevel feedback-queue scheduling algorithm,** in contrast, allowsa process to move between queues. The idea is to separate processes accordingto the characteristics of their CPU bursts. If a process uses too much CPU time,it will be moved to a lower-priority queue. This scheme leaves I/O-bound andinteractive processes in the higher-priority queues. In addition, a process thatwaits too long in a lower-priority queue may be moved to a higher-priorityqueue. This form of aging prevents starvation.

For example, consider a multilevel feedback-queue scheduler with threequeues, numbered from 0 to 2. The scheduler first executes allprocesses in queue 0. Only when queue 0 is empty will it execute processes in queue 1. Similarly, processes in queue 2 will only be executed if queues 0and 1 are empty. A process that arrives for queue 1 will preempt a process inqueue 2. A process in queue 1 will in turn be preempted by a process arrivingfor queue 0.A process entering the ready queue is put in queue 0. A process in queue 0is given a time quantum of 8 milliseconds. If it does not finish within this time,it is moved to the tail of queue 1. If queue 0 is empty, the process at the headof queue 1 is given a quantum of 16 milliseconds. If it does not complete, it ispreempted and is put into queue 2. Processes in queue 2 are run on an FCFSbasis but are run only when queues 0 and 1 are empty.

This scheduling algorithm gives highest priority to any process with a CPUburst of 8 milliseconds or less. Such a process will quickly get the CPU, finishits CPU burst, and go off to its next I/O burst. Processes that need more than8 but less than 24 milliseconds are also served quickly, although with lowerpriority than shorter processes. Long processes automatically sink to queue2 and are served in FCFS order with any CPU cycles left over from queues 0 and 1.



Multilevel Feedback Queues

In general, a multilevel feedback-queue scheduler is defined by thefollowing parameters:

• The number of queues

• The scheduling algorithm for each queue

• The method used to determine when to upgrade a process to a higherpriorityqueue

• The method used to determine when to demote a process to a lowerpriorityqueue

• The method used to determine which queue a process will enter when thatprocess needs service

The definition of a multilevel feedback-queue scheduler makes it the mostgeneral CPU-scheduling algorithm.

7. Algorithm Evaluation

The first problem is defining the criteria to be used in selecting an algorithm.As we saw , criteria are often defined in terms of CPU utilization,response time, or throughput. To select an algorithm, we must first definethe relative importance of these measures. Our criteria may include severalmeasures, such as:

• Maximizing CPU utilization under the constraint that the maximumresponse time is 1 second

• Maximizing throughput such that turnaround time is (on average) linearlyproportional to total execution time

Once the selection criteria have been defined, we want to evaluate thealgorithms under consideration. We next describe the various evaluationmethods we can use.

1 Deterministic Modeling

One major class of evaluation methods is **analytic evaluation.** Analyticevaluation uses the given algorithm and the system workload to produce aformula or number that evaluates the performance of the algorithm for thatworkload.

One type of analytic evaluation is **deterministic modeling.** This methodtakes a particular predetermined workload and defines the performance of eachalgorithm for that workload.

Deterministic modeling is simple and fast. It gives us exact numbers,allowing us to compare the algorithms. However, it requires exact numbers forinput, and its answers apply only to those cases.

2 Queuing Models

On many systems, the processes that are run vary from day to day, so thereis no static set of processes (or times) to use for deterministic modeling. Whatcan be determined, however, is the distribution of CPU and I/O bursts. Thesedistributions can be measured and then approximated or simply estimated. Theresult is a mathematical formula describing the probability of a particular CPUburst. Commonly, this distribution is exponential and is described by its mean.

Similarly, we can describe the distribution of times when processes arrive inthe system (the arrival-time distribution). From these two distributions, it ispossible to compute the average throughput, utilization, waiting time, and soon for most algorithms.

Queueing analysis can be useful in comparing scheduling algorithms,but it also has limitations. At the moment, the classes of algorithms anddistributions that can be handled are fairly limited. The mathematics of complicated algorithms and distributions can be difficult to work with.

3 Simulations

To get a more accurate evaluation of scheduling algorithms, we can use**simulations.** Running simulations involves programming a model of thecomputer system. Software data structures represent the major componentsof the system. Simulations can be expensive, often requiring hours of computer time. Amore detailed simulation provides more accurate results, but it also requiresmore computer time. In addition, trace tapes can require large amounts ofstorage space. Finally, the design, coding, and debugging of the simulator canbe a major task.

4 Implementation

Even a simulation is of limited accuracy. The only completely accurate wayto evaluate a scheduling algorithm is to code it up, put it in the operatingsystem, and see how it works. This approach puts the actual algorithm in thereal system for evaluation under real operating conditions.

The major difficulty with this approach is the high cost. Another difficulty is that the environment in which the algorithm is usedwill change. The environment will change not only in the usual way, as new programs are written and the types of problems change, but also as a result of the performance of the scheduler.

SYNCHRONIZATION

The system consisting of cooperatingsequential processes or threads, all running asynchronously and possiblysharing data. We illustrated this model with the producer-consumer problem,described how a bounded buffer could be used to enable processes to sharememory.

Solution allows at most BUFFER.SIZE - 1 items in the buffer at the sametime. Suppose we want to modify the algorithm to remedy this deficiency. Onepossibility is to add an integer variable counter, initialized to 0. counter isincremented every time we add a new item to the buffer and is decrementedevery time we remove one item from the buffer. The code for the producerprocess can be modified as follows:

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

The code for the consumer process can be modified as

```
follows: while (true)
{
while (counter ==
0) ; /* do nothing */
nextConsumed = buffer [out] ,-out
= (out + 1) % BUFFER_SIZE;
counter--;
/* consume the item in nextConsumed */
}
```

Although both the producer and consumer routines are correct separately,they may not function correctly when executed concurrently. As an illustration,suppose that the value of the variable counter is currently 5 and that theproducer and consumer processes execute the statements "counter++" and"counter—" concurrently.

We can show that the value of counter may be incorrect as follows. Notethat the statement "counter++" may be implemented in machine language (ona typical machine) as

Register1=- counter
Register1 = register1 + 1
counter =*register1*

where *register1*is a local CPU register. Similarly, the statement "counter—" isimplemented as follows:

register2=counter
register2 = register2- 1
counter = *register2*

where again *register2*is a local CPU register. Even though *register1*and*register2*may be the same physical register (an accumulator, say), rememberthat the contents of this register will be saved and restored by the interrupthandler.

The concurrent execution of "counter++" and "counter—" is equivalentto a sequential execution where the lower-level statements presented previouslyare interleaved in some arbitrary order. One such interleaving is

Register1= counter *{register1 = 5}*
register 1= register1+ 1 *{register1*= 6}
register2. = counter *{register2* =5}
register2 = registeri — 1 *{register2* =4}
counter = *register1 {counter* = 6}
counter = *register2 {counter* = 4}

Notice that we have arrived at the incorrect state "counter == 4", indicatingthat four buffers are full, when, in fact, five buffers are full. If we reversed theorder of the statements, we would arrive at the incorrect state"counter —— 6".

We would arrive at this incorrect state because we allowed both processesto manipulate the variable counter concurrently. A situation like this, whereseveral processes access and manipulate the same data concurrently and theoutcome of the execution depends on the particular order in which the accesstakes place, is called a **race condition.** To guard against the race conditionabove, we need to ensure that only one process at a time can be manipulatingthe variable counter. To make such a guarantee, we require that the processes be synchronized in some way.

1. The Critical-Section Problem

Consider a system consisting of *n* processes {PQ, PI, ..., *P,,~\}.* Each processhas a segment of code, called a **critical section,** in which the process maybe changing common variables, updating a table, writing a file, and so on.The important feature of the system is that, when one process is executing inits critical section, no other process is to be allowed to execute in its criticalsection. That is, no two processes are executing in their critical sections at thesame time. The *critical-section problem* is to design a protocol that the processes can use to cooperate. Each process must request permission to enter its criticalsection. The section of code implementing this request is the **entry section.** Thecritical section may be followed by an **exit section.** The remaining code is the**remainder section.** The general structure of a typical process P, is shown inFigure 6.1. The entry section and exit section are enclosed in boxes to highlightthese important segments of code.

```
do{
\\entry section
critical section
\\exit section
\\remainder section
} while (TRUE);
```

General structure of a typical process *Pi.*

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

- **Mutual exclusion.** If process P; is executing in its critical section, then noother processes can be executing in their critical sections.
- **Progress.** If no process is executing in its critical section and someprocesses wish to enter their critical sections, then only those processesthat are not executing in their remainder sections can participate in the decision on which will enter its critical section next, and this selectioncannot be postponed indefinitely.
- **Bounded waiting.** There exists a bound, or limit, on the number of timesthat other processes are allowed to enter their critical sections after aprocess has made a request to enter its critical section and before that request is granted.

Peterson's Solution

It is a classic software-based solution to the critical-sectionproblem known as **Peterson's solution.** Peterson's solution is restricted to two processes that alternate executionbetween their critical sections and remainder sections. The processes arenumbered Po and Pi. For convenience, when presenting P,-, we use *Pj* todenote the other process; that is, j equals 1 — i.

Peterson's solution requires two data items to be shared between the twoprocesses:

```
int turn;
boolean f l a g [2] •
```

The variable turn indicates whose turn it is to enter its critical section. That is,if turn == i, then process P; is allowed to execute in its critical section. Theflag array is used to indicate if a process *is ready* to enter its critical section.For example, if f lag[i] is true, this value indicates that P; is ready to enterits critical section.

To enter the critical section, process P, first sets flag[i] to be true andthen sets turn to the value j, thereby asserting that if the other process wishesto enter the critical section, it can do so. If both processes try to enter at thesame time, turn will be set to both i and j at roughly the same time. Onlyone of these assignments will last; the other will occur but will be overwrittenimmediately. The eventual value of turn decides which of the two processesis allowed to enter its critical section first.

```
do
 {
flag[i] = TRUE;
turn = j ;
while (flag[j] turn == j )
; critical section
flag[i] = FALSE;
remainder section
} while (TRUE);
```

The structure of process *P-,* in Peterson's solution.

We now prove that this solution is correct. We need to show that:

- Mutual exclusion is preserved.
- The progress requirement is satisfied.
- The bounded-waiting requirement is met.

To prove property 1, we note that each P; enters its critical section onlyif either flag[j] == false or turn -- i. Also note that, if both processescan be executing in their critical sections at the same time, then flag [0] == flag [1] == true. These two observations imply that Po and Pi could not havesuccessfully executed their while statements at about the same time, since thevalue of turn can be either 0 or 1 but cannot be both. Hence, one of the processes—say *Pj*—must have successfully executed the while statement, whereas P,had to execute at least one additional statement ("turn == j"). However, since,at that time, f lag[j] == true, and turn == j, and this condition will persistas long as *Pj* is in its critical section, the result follows: Mutual exclusion ispreserved.

To prove properties 2 and 3, we note that a process P, can be prevented fromentering the critical section only if it is stuck in the while loop with the conditionflag [j] == true and turn == j; this loop is the only one possible. If P; is notready to enter the critical section, then flag [j] == false, and P; can enter itscritical section. If *Pj* has set flag [j] to true and is also executing in its whilestatement, then either turn == i or turn == j . If turn == i, then P, will enterthe critical section. If turn == j, then *Pj* will enter the critical section. However,once *P;* exits its critical section, it will reset f lag[j] to false, allowing P, toenter its critical section. If *Pj* resets flag [j] to true, it must also set turn to i.

Thus, since P, does not change the value of the variable turn while executingthe while statement, P,- will enter the critical section (progress) after at mostone entry by P/ (bounded waiting).

3. Synchronization Hardware

We have just described one software-based solution to the critical-sectionproblem. We explore several more solutions to thecritical-section problem using techniques ranging from hardware to softwarebasedAPIs available to application programmers. Hardware features can make any programming task easier and improve system efficiency. In this section, we present some simple hardware instructionsthat are available on many systems and show how they can be used effectivelyin solving the critical-section problem.

The critical-section problem could be solved simply in a uniprocessor environment, we could be sure that the current sequenceof instructions would be allowed to execute in order without preemption. Unfortunately, this solution is not as feasible in a multiprocessor environment.Disabling interrupts on a multiprocessor can be time consuming, as themessage is passed to all the processors. This message passing delays entry intoeach critical section, and system efficiency decreases.

```
boolean TestAndSet(boolean *target)
{ boolean rv = *target;
*target = TRUE;
return rv;
```

The definition of the TestAndSet () instruction.

```
do {
while (TestAndSetLock(&lock) )
; // do nothing
 critical  section
lock = FALSE;
 remainder  section
}while (TRUE);
```

Mutual-exclusion implementation with TestAndSet ( ) .

The TestAndSet() instruction can be defined as shown in Figure . The important characteristic is that this instruction is executed atomically.Thus, if two TestAndSet C) instructions are executed simultaneously (each ona different CPU), they will be executed sequentially in some arbitrary order. Ifthe machine supports the TestAndSet () instruction, then we can implementmutual exclusion by declaring a Boolean variable lock, initialized to false.The structure of process P, is shown in above Figure.

The SwapO instruction, in contrast to the TestAndSet0 instruction,operates on the contents of two words; it is defined as shown in Figure .Like the TestAndSet 0 instruction, it is executed atomically. If the machinesupports the SwapO instruction, then mutual exclusion can be provided asfollows. A global Boolean variable lock is declared and is initialized to false.In addition, each process has a local Boolean variable key. The structure ofprocess *P,* is shown in Figure .

```
void Swap(boolean *a, boolean *b)
{
boolean temp = *a;
*a = *b;
*b = temp;
}
```

The definition of the Swap () instruction.

```
do { ,
key = TRUE;
while (key == TRUE)
Swap (&lock, &key) ,-
 critical  section
lock = FALSE;
 remainder  section
}while (TRUE);
```

Mutual-exclusion implementation with the SwapO instruction.

4. Semaphores

The various hardware-based solutions to the critical-section problem (usingthe TestAndSetC) and SwapO instructions) arecomplicated for application programmers to use. To overcome this difficulty,we can use a synchronization tool called a semaphore.

A semaphore S is an integer variable that, apart from initialization, isaccessed only through two standard atomic operations: wait () and signal ().The waitO operation was originally termed P (from the Dutch *probercn,* "totest"); signal () was originally called V (from *verhogen,* "to increment"). Thedefinition of wait() is as follows:

```
wait(S) {
while (S <= 0)
; // no-op
S--;
}
```

The definition of signal () is as follows: signal(S)

```
{
S + + ;
}
```

All the modifications to the integer value of the semaphore in the wait ()and signal() operations must be executed indivisibly. That is, when oneprocess modifies the semaphore value, no other process can simultaneouslymodify that same semaphore value.

In addition, in the case of wait(S), thetesting of the integer value of S (S < 0), and its possible modification (S—),must also be executed without interruption. We shall see how these operationscan be implemented in Section 6.5.2; first, let us see how semaphores can beused.

Operating systems often distinguish between counting and binary semaphores.The value of a **counting semaphore** can range over an unrestricted domain.The value of a **binary semaphore** can range only between 0 and 1. On somesystems, binary semaphores are known as **mutex locks,** as they are locks thatprovide *mutual Ex*clusion.

We can use binary semaphores to deal with the critical-section problem formultiple processes. The *n* processes share a semaphore, mutex, initialized to 1.Each process P, is organized as shown in Figure.

```
do {
wait(mutex);
 critical  section
signal(mutex);
 remainder  section
}while (TRUE);
```

Mutual-exclusion implementation with semaphores.

Counting semaphores can be used to control access to a given resourceconsisting of a finite number of instances. The semaphore is initialized to thenumber of resources available. Each process that wishes to use a

resourceperforms a waitQ operation on the semaphore (thereby decrementing thecount). When a process releases a resource, it performs a signal () operation(incrementing the count). When the count for the semaphore goes to 0, allresources are being used. After that, processes that wish to use a resource willblock until the count becomes greater than 0.

The main disadvantage of the semaphore definition given here is that it requires**busy waiting.** While a process is in its critical section, any other process thattries to enter its critical section must loop continuously in the entry code. Thiscontinual looping is clearly a problem in a real multiprogramming system,where a single CPU is shared among many processes. Busy waiting wastesCPU cycles that some other process might be able to use productively. Thistype of semaphore is also called a **spinlock** because the process "spins" whilewaiting for the lock.

To overcome the need for busy waiting, we can modify the definition ofthe wait () and signal () semaphore operations. When a process executes thewait () operation and finds that the semaphore value is not positive, it mustwait. However, rather than engaging in busy waiting, the process can *block*itself. The block operation places a process into a waiting queue associatedwith the semaphore, and the state of the process is switched to the waitingstate. Then control is transferred to the CPU scheduler, which selects anotherprocess to execute.

A process that is blocked, waiting on a semaphore S, should be restartedwhen some other process executes a signal() operation. The process isrestarted by a wakeup () operation, which changes the process from the waitingstate to the ready state. The process is then placed in the ready queue.

To implement semaphores under this definition, we define a semaphore asa "C"

```
struct: typedef struct {
int value;
struct process *list;
} semaphore;
```

Each semaphore has an integer value and a list of processes l i s t . Whena process must wait on a semaphore, it is added to the list of processes. Asignal () operation removes one process from the list of waiting processesand awakens that process.

The wait () semaphore operation can now be defined

```
as wait(semaphore *S) {
S->value—;
if (S->value < 0) {
add this process to S-
>list; block();

}}
```

The signal () semaphore operation can now be defined as

```
# signal(semaphore *S) { S-
>value++;
if (S->value <= 0) {
remove a process P from S->list;
wakeup(P);
}
}
```

The block() operation suspends the process that invokes it. The wakeup(P)operation resumes the execution of a blocked process P. These two operationsare provided by the operating system as basic system calls.

If the semaphore value is negative, its magnitudeis the number of processes waiting on that semaphore. The list of waiting processes can be easily implemented by a link field ineach process control block (PCB). Each semaphore contains an integer valueand a pointer to a list of PCBs. One way to add and remove processes fromthe list in a way that ensures bounded waiting is to use a FIFO queue, wherethe semaphore contains both head and tail pointers to the queue.

Deadlocks and Starvation

The implementation of a semaphore with a waiting queue may result in asituation where two or more processes are waiting indefinitely for an eventthat can be caused only by one of the waiting processes. The

event in questionis the execution of a signal() operation. When such a state is reached, theseprocesses are said to be **deadlocked.**

To illustrate this, we consider a system consisting of two processes, P1 and P2 , each accessing two semaphores, S and Q, set to the value 1:

P1	P2
wait(S);	wait(Q);
wait(Q);	wait(S);
---------	----------
---------	----------
signal(S);	signal(Q);
signal(Q);	signal(S);

Suppose that *P1*executes wait (S) and then P2 executes wait (Q). When P1 executes wait(Q), it must wait until P2 executes signal(Q). Similarly, whenP2 executes wait(S), it must wait until P1 executes signal(S). Since thesesignal () operations cannot be executed, P1 and P2 are deadlocked.

We say that a set of processes is in a deadlock state when every process inthe set is waiting for an event that can be caused only by another process in theset.

Another problem related to deadlocks is **indefinite blocking,** or **starvation,**a situation in which processes wait indefinitely within the semaphore.Indefinite blocking may occur if we add and remove processes from the listassociated with a semaphore in LIFO (last-in, first-out) order.

5. Classic Problems of Synchronization

In this section, we present a number of synchronization problems as examples. In our solutionsto the problems, we use semaphores for synchronization.

The Bounded-Buffer Problem

The *bounded-buffer problem* is commonlyused to illustrate the power of synchronization primitives. We assume that the pool consists of *n* buffers, each capable of holdingone item.

The mutex semaphore provides mutual exclusion for accesses to thebuffer pool and is initialized to the value 1.

The empty and f u l l semaphorescount the number of empty and full buffers. The semaphore empty is initializedto the value n; the semaphore f u l l is initialized to the value 0.

The code for the producer process is shown in below Figure ; the code forthe consumer process is shown in below Figure. We can interpret this code as the producerproducing full buffers for the consumer or as the consumer producing emptybuffers for the producer.

```
do {
• produce an item in nextp
wait(empty);
wait(mutex);
• add nextp to buffer
signal(mutex);
signal(full);
}while (TRUE) ,-
```

The structure of the producer process.

```
do {
wait(full);
wait(mutex);
• remove an item from buffer to
nextc signal(mutex);
signal(empty);
• consume the item in nextc
}while (TRUE);
```

The structure of the consumer process.

The Readers-Writers Problem

A database is to be shared among several concurrent processes. Some of theseprocesses may want only to read the database, whereas others may want toupdate (that is, to read and write) the database. We distinguish between thesetwo types of processes by referring to the former as **readers** and to the latteras **writers**.

Obviously, if two readers access the shared data simultaneously, noadverse affects will result. However, if a writer and some other thread (eithera reader or a writer) access the database simultaneously, chaos may ensue.To ensure that these difficulties do not arise, we require that the writershave exclusive access to the shared database. This synchronization problem isreferred to as the *readers-writers problem.*

The readers-writers problem, requires that no readerwill be kept waiting unless a writer has already obtained permission to usethe shared object. In other words, no reader should wait for other readers to finish simply because a writer is waiting.

In the solution to the readers-writers problem, the reader processesshare the following data structures: semaphore mutex, wrt;
int readcount;

The semaphores mutex and wrt are initialized to 1; readcount is initializedto 0. The semaphore wrt is common to both reader and writer processes.

The mutex semaphore is used to ensure mutual exclusion when the variablereadcount is updated. The readcount variable keeps track of how manyprocesses are currently reading the object.

The semaphore wrt functions as amutual-exclusion semaphore for the writers. It is also used by the first reader that enters or exits the critical section. It is not used by readers whoenter or exit while other readers are in their critical sections or last.

```
do {
wait(wrt);
// writing is performed
signal (wrt) ,-
}while (TRUE);
```

The structure of a writer process.

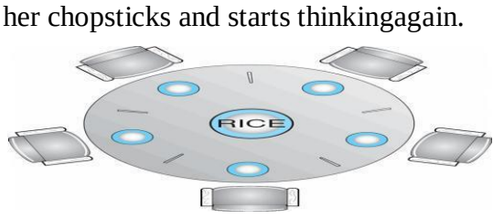
```
do {
wait(mutex);
readcount + + ;
if (readcount ==
1) wait(wrt);
signal(mutex);
// reading is performed
wait (mutex) ,-
readcount--;
if (readcount ==
0) signal(wrt);
signal(mutex);
Jwhile (TRUE);
```

The structure of a reader process.

The code for a writer process is shown in above Figure; the code for a readerprocess is shown in above Figure. Note that, if a writer is in the critical sectionand *n* readers are waiting, then one reader is queued on wrt, and n — 1 readersare queued on mutex. Also observe that, when a writer executes signal (wrt),we may resume the execution of either the waiting readers or a single waitingwriter. The selection is made by the scheduler.

The Dining-Philosophers Problem

Consider five philosophers who spend their lives thinking and eating. Thephilosophers share a circular table surrounded by five chairs, each belongingto one philosopher. In the center of the table is a bowl of rice, and the table is laidwith five single chopsticks. When a philosopher thinks, she doesnot interact with her colleagues. From time to time, a philosopher gets hungryand tries to pick up the two chopsticks that are closest to her (the chopsticksthat are between her and her left and right neighbors). A philosopher may pickup only one chopstick at a time. Obviously, she cannot pick up a chopstick thatis already in the hand of a neighbor. When a hungry philosopher has both herchopsticks at the same time, she eats without releasing her chopsticks. Whenshe is finished eating, she puts down both of her chopsticks and starts thinkingagain.



The situation of the dining philosophers.

One simple solution is to represent each chopstick with a semaphore. Aphilosopher tries to grab a chopstick by executing a wait () operation on thatsemaphore; she releases her chopsticks by executing the signal() operationon the appropriate semaphores. Thus, the shared data are

semaphore chopstick[5];

where all the elements of chopstick are initialized to 1. The structure ofphilosopheri is shown in Figure

```
do {
wait (chopstick [i] ) ,-
wait(chopstick [ (i + 1) % 5] )
; // eat
signal(chopstick [i]);
signal(chopstick [(i + 1) %
5]); / / think
}while (TRUE);
```

The structure of philosopher *i.*

Although this solution guarantees that no two neighbors are eatingsimultaneously, it nevertheless must be rejected because it could create adeadlock. Suppose that all five philosophers become hungry simultaneouslyand each grabs her left chopstick. All the elements of chopstick will now beequal to 0. When each philosopher tries to grab her right chopstick, she will bedelayed forever.

we present a solution to the dining-philosophers problem thatensures freedom from deadlocks.

5. Allow at most four philosophers to be sitting simultaneously at the table.
6. Allow a philosopher to pick up her chopsticks only if both chopsticks areavailable.
7. Use an asymmetric solution; that is, an odd philosopher picks up first herleft chopstick and then her right chopstick, whereas an even philosopherpicks up her right chopstick and then her left chopstick.

Finally, any satisfactory solution to the dining-philosophers problem mustguard against the possibility that one of the philosophers will starve to death.A deadlock-free solution does not necessarily eliminate the possibility of starvation.

6. Monitors

Although semaphores provide a convenient and effective mechanism forprocess synchronization, using them incorrectly can result in timing errors that are difficult to detect.

To illustrate how, we review the semaphore solution to the criticalsectionproblem. All processes share a semaphore variable mutex, which is.initialized to 1. Each process must execute wait (mutex) before entering thecritical section and signal (mutex) afterward. If this sequence is not observed,two processes may be in their critical sections simultaneously.

Let us examinethe various difficulties that may result.

• Suppose that a process interchanges the order in which the wait(j andsignal () operations on the semaphore mutex are executed, resulting inthe following execution:

```
signal(mutex);
critical section
wait(mutex);
```

In this situation, several processes may be executing in their critical sectionssimultaneously, violating the rmitual-exclusion requirement.

8. Suppose that a process replaces signal (mutex) with wait (mutex). Thatis, it executes wait(mutex);

```
critical section
wait(mutex);
```

In this case, a deadlock will occur.

9.Suppose that a process omits the wait (mutex), or the signal (mutex), orboth. In this case, either mutual exclusion is violated or a deadlock willoccur.

To deal with such errors, researchers have developed high-level languageconstructs. In this section, we describe one fundamental high-level synchronizationconstruct—the monitor type.

Usage

A type, or abstract data type, encapsulates private data with public methodsto operate on that data. A monitor type presents a set of programmer-definedoperations that are provided mutual exclusion within the monitor. The syntax of a monitor is shown in Figure.

```
monitor monitor name f
{
II shared variable declarations
procedure PI ( . . . ) {
}
p r o c e d u r e P 2 ( . . . ) {…..}
::::::::::::::::::::::::::::::::::::::::
p r o c e d u r e P n ( . . . ) {….}

i n i t i a l i z a t i o n c o d e ( . . . ) {
……..
}
        }
```

Syntax of a monitor.

Thus, a procedure defined within a monitor can access onlythose variables declared locally within the monitor and its formal parameters.Similarly, the local variables of a monitor can be accessed by only the localprocedures.

The monitor construct ensures that only one process at a time can beactive within the monitor. Consequently, the monitor construct, as defined so far, is not sufficiently powerful formodeling some synchronization schemes. For this purpose, we need to defineadditional synchronization mechanisms. These mechanisms are provided bythe condition construct. A programmer who needs to write a tailor-madesynchronization scheme can define one or more variables of type *condition:*

condition x, y;

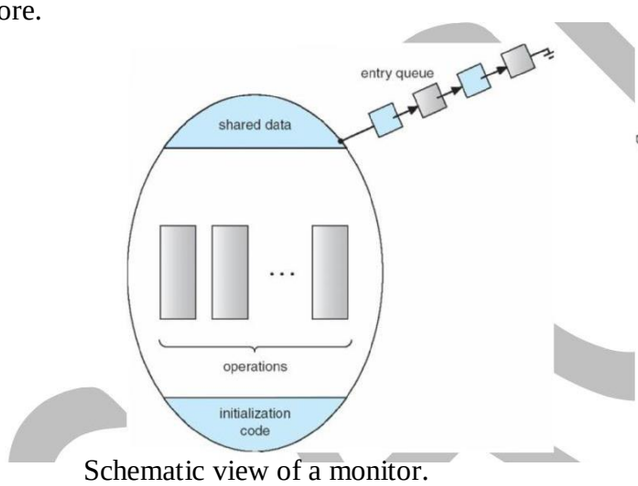
The only operations that can be invoked on a condition variable are wait ()and signal(). The

operation x.waitO ;

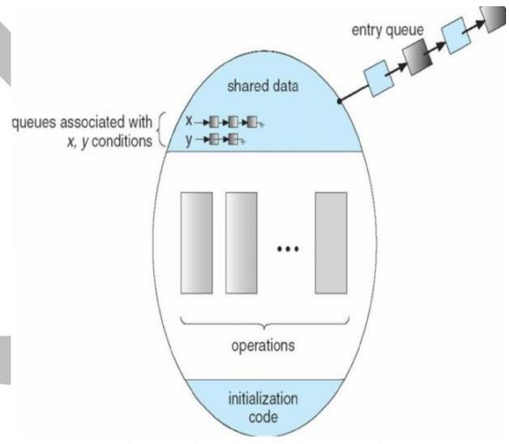
means that the process invoking this operation is suspended until anotherprocess invokes

x . s i g n a l ( ) ;

The x. signal () operation resumes exactly one suspended process. If noprocess is suspended, then the signal () operation has no effect; that is, thestate of x is the same as if the operation had never been executed. Contrast this operation with the signal () operation associated withsemaphores, which always affects the state of the semaphore.



Schematic view of a monitor.



Monitor with condition variables.

Now suppose that, when the x. **s** ignal () operation is invoked by a processP, there is a suspended process Q associated with condition x. Clearly, if thesuspended process Q is allowed to resume its execution, the signaling process Pmust wait. Otherwise, both P and Q would be active simultaneously within themonitor. Note, however, that both processes can conceptually continue withtheir execution. Two possibilities exist:

- **Signal and wait.** *P* either waits until Q leaves the monitor or waits foranother condition.
- **Signal and continue.** *Q* either waits until *P* leaves the monitor or waitsfor another condition.

Dining-Philosophers Solution Using Monitors

We now illustrate monitor concepts by presenting a deadlock-free solution tothe dining-philosophers problem. This solution imposes the restriction that aphilosopher may pick up her chopsticks only if both of them are available. Tocode this solution, we need to distinguish among three states in which we mayfind a philosopher. For this purpose, we introduce the following data structure:

enum {thinking, hungry, eating} s t a t e [5] ;

Philosopher *i* can set the variable s t a t e [i] = eating only if her twoneighbors are not eating: (s t a te [(i+4) °/» 5] != eating) and (s t a te [(i+1)% 5] != eating).

We also need to declarecondition self [5];

where philosopheri can delay herself when she is hungry but is unable toobtain the chopsticks she needs.

We are now in a position to describe our solution to the dining-philosophersproblem. The distribution of the chopsticks is controlled by the monitor dp,whose definition is shown in Figure.

dp.pickup(i);
eat
dp.putdown(i);

Each philosopher, before starting toeat, must invoke the operation pi ckup (). This may result in the suspension ofthe philosopher process. After the successful completion of the operation, thephilosopher may eat. Following this, the philosopher invokes the putdownOoperation. Thus, philosopher *i* must invoke the operations pi ckup () andputdownO in the following sequence:

```
monitor DP
  {
        enum { THINKING; HUNGRY, EATING) state [5] ;
        condition self [5];
        void pickup (int i) {
              state[i] = HUNGRY;
              test(i);
              if (state[i] != EATING) self [i].wait;
        }

     void putdown (int i) {
              state[i] = THINKING;
              // test left and right neighbors
               test((i + 4) % 5);
               test((i + 1) % 5);
      }

void test (int i) {
               if ( (state[(i + 4) % 5] != EATING)
               && (state[i] == HUNGRY) &&
               (state[(i + 1) % 5] != EATING) )
                   { state[i] = EATING ;
                     self[i].signal () ;
                   }
           }
     initialization_code() {
              for (int i = 0; i < 5; i++)
              state[i] = THINKING;
        }
}
```

A monitor solution to the dining-philosopher problem.

It is easy to show that this solution ensures that no two neighbors are eatingsimultaneously and that no deadlocks will occur. We note, however, that it ispossible for a philosopher to starve to death. We do not present a solution tothis problem but rather leave it as an exercise for you.

Implementing a Monitor Using Semaphores

We now consider a possible implementation of the monitor mechanism usingsemaphores. For each monitor, a semaphore mutex (initialized to 1) is provided.A process must execute wait (mutex) before entering the monitor and mustexecute signal (mutex) after leaving the monitor.

Since a signaling process must wait until the resumed process either leavesor waits, an additional semaphore, next, is introduced, initialized to 0, onwhich the signaling processes may suspend themselves. An integer variablenext-count is also provided to count the number of processes suspended onnext. Thus, each external procedure F is replaced by

```
wait(mutex);
body of F
if (next_count >
0) signal(next);
else
signal(mutex);
```

Mutual exclusion within a monitor is ensured.

We can now describe how condition variables are implemented. For eachcondition x, we introduce a semaphore x_sem and an integer variable x_countboth initialized to 0.

The operation x. wait () can now be implemented as

```
x_count++;
if (next_count > 0)
signal(next);
else
signal(mutex);
wait(x_sem);
x_count—;
```

The operation x. signal () can be implemented as

```
if (x_count > 0) {
next_count++;
signal(x_sem);
wait(next) ;
next_count--;
```

- **Synchronization Examples**
  - Solaris
  - Windows XP
  - Linux
  - Pthreads

Solaris Synchronization

- **Implements a variety of locks to support** multitasking, multithreading (including real-time threads), and multiprocessing
- Uses adaptive mutexes for efficiency when protecting data from short code segments
- Uses condition variables and readers-writers locks when longer sections of code need access to data
- Uses turnstiles to order the list of threads waiting to acquire either an adaptive mutex or reader-writer lock

Windows XP Synchronization

- Uses interrupt masks to protect access to global resources on uniprocessor systems
- Uses spinlocks on multiprocessor systems
- Also provides dispatcher objects which may act as either mutexes and semaphores
- Dispatcher objects may also provide events
- An event acts much like a condition variable

Linux Synchronization

- Linux:lPrior to kernel Version 2.6, disables interrupts to implement short critical sections
- Version 2.6 and later, fully preemptive
- Linux provides:
- semaphores
- spin locks

Pthreads Synchronization

- Pthreads API is OS-independent
- It provides:
- mutex locks
- condition variablesnNon-portable extensions include:
- read-write locks
- spin locks

Memory Management

- To provide a detailed description of various ways of organizing memory hardware
- To discuss various memory-management techniques, including paging and segmentation
- To provide a detailed description of the Intel Pentium, which supports both pure segmentation and segmentation with paging
- Program must be brought (from disk) into memory and placed within a process for it to be run
- Main memory and registers are only storage CPU can access directly
- Register access in one CPU clock (or less)
- Main memory can take many cycles
- **Cache** sits between main memory and CPU registers
- Protection of memory required to ensure correct operation

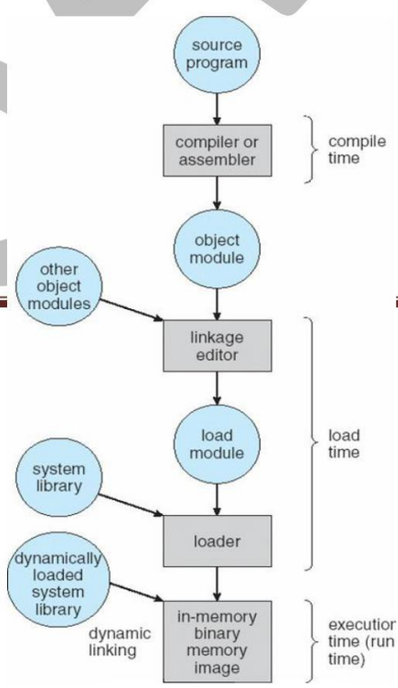
Base and Limit Registers

A pair of **base** and **limit** registers define the logical address space

Binding of Instructions and Data to Memory

- Address binding of instructions and data to memory addresses can happen at three different stages
- **Compile time**: If memory location known a priori, **absolute code** can be generated; must recompile code if starting location changes
- **Load time**: Must generate **relocatable code** if memory location is not known at compile time
- **Execution time**: Binding delayed until run time if the process can be moved during its execution from one memory segment to another. Need hardware support for address maps (e.g., base and limitregisters)

Multistep Processing of a User Program



SACET

Logical vs. Physical Address Space

- The concept of a logical address space that is bound to a separate **physical address space** is central to proper memory management
- **Logical address** – generated by the CPU; also referred to as **virtual address**
- **Physical address** – address seen by the memory unit
- Logical and physical addresses are the same in compile-time and load-time address-binding schemes; logical (virtual) and physical addresses differ in execution-time address-binding scheme

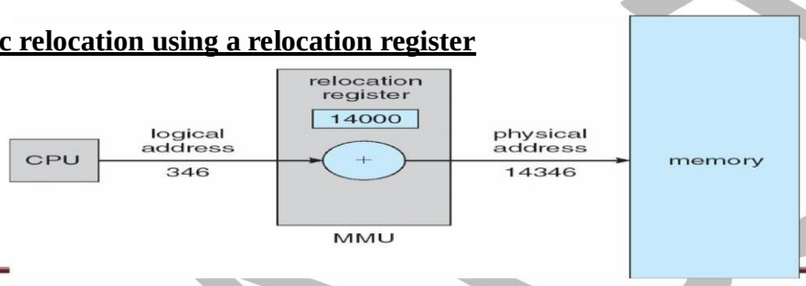
Memory-Management Unit (MMU)

Hardware device that maps virtual to physical address

In MMU scheme, the value in the relocation register is added to every address generated by a user process at the time it is sent to memory

The user program deals with *logical* addresses; it never sees the *real* physical addresses

Dynamic relocation using a relocation register



Dynamic Loading

- Routine is not loaded until it is called
- Better memory-space utilization; unused routine is never loaded
- Useful when large amounts of code are needed to handle infrequently occurring cases
- No special support from the operating system is required implemented through program design

Dynamic Linking

- Linking postponed until execution time
- **Small piece of code, *stub*, used to locate the appropriate memory-resident library routine**
- Stub replaces itself with the address of the routine, and executes the routine
- Operating system needed to check if routine is in processes' memory address
- Dynamic linking is particularly useful for libraries
- System also known as **shared libraries**

Swapping

A process can be swapped temporarily out of memory to a backing store, and then brought back into memory for continued execution

n**Backing store** –fast disk large enough to accommodate copies of all memory images for all users; must provide direct access to these memory images

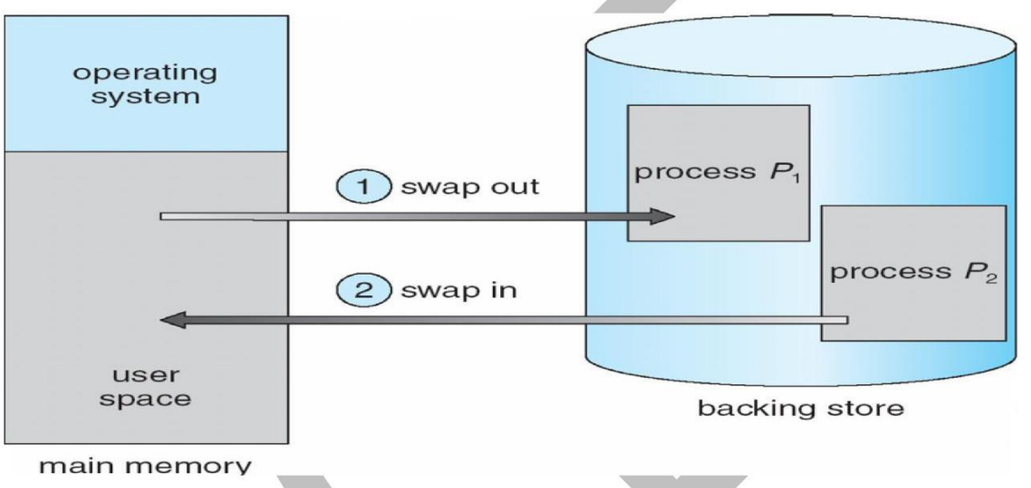
n**Roll out, roll in** –swapping variant used for priority-based scheduling algorithms; lower-priority process is swapped out so higher-priority process can be loaded and executed

nMajor part of swap time is transfer time; total transfer time is directly proportional to the amount of memory swapped

nModified versions of swapping are found on many systems (i.e., UNIX, Linux, and Windows)

System maintains a **ready queue** of ready-to-run processes which have memory images on disk

Schematic View of Swapping



Contiguous Allocation

Main memory usually into two partitions:

Resident operating system, usually held in low memory with interrupt vector

User processes then held in high memory

nRelocation registers used to protect user processes from each other, and from changing operating-system code and data

Base register contains value of smallest physical address

Limit register contains range of logical addresses – each logical address must be less than the limit register

MMU maps logical address *dynamically*

Hardware Support for Relocation and Limit Registers

- Multiple-partition allocation
- Hole – block of available memory; holes of various size are scattered throughout memory
- When a process arrives, it is allocated memory from a hole large enough to accommodate it
- Operating system maintains information about:
  a) allocated partitions b) free partitions (hole)

Dynamic Storage-Allocation Problem

O • **First -fit**: Allocate the O*first* hole that is big enough O O

process 5 process 5 process 5 process 5

- **Best-fit**: Allocate the *smallest* hole that is big enough; must search entire list, unless ordered by size Produces the smallest leftover hole
- **Worst-fit**: Allocate the *largest* hole; must also search entire list
- Produces the largest leftover hole
- First-fit and best-fit better than worst-fit in terms of speed and storage utilization

Fragmentation

- **External Fragmentation** – total memory space exists to satisfy a request, but it is not contiguous
- **Internal Fragmentation** – allocated memory may be slightly larger than requested memory; this size difference is memory internal to a partition, but not being used
- Reduce external fragmentation by **compaction**
- Shuffle memory contents to place all free memory together in one large block
- Compaction is possible *only* if relocation is dynamic, and is done at execution time.
- I/O problem

  Latch job in memory while it is involved in I/O

  Do I/O only into OS buffers

Paging

Logical address space of a process can be noncontiguous; process is allocated physical memory whenever the latter is available

Divide physical memory into fixed-sized blocks called **frames** (size is power of 2, between 512 bytes and 8,192 bytes)

Divide logical memory into blocks of same size called **pages**

nKeep track of all free frames

To run a program of size ***n*** pages, need to find *n* free frames and load

program Set up a page table to translate logical to physical addresses

Internal fragmentation

Address Translation Scheme

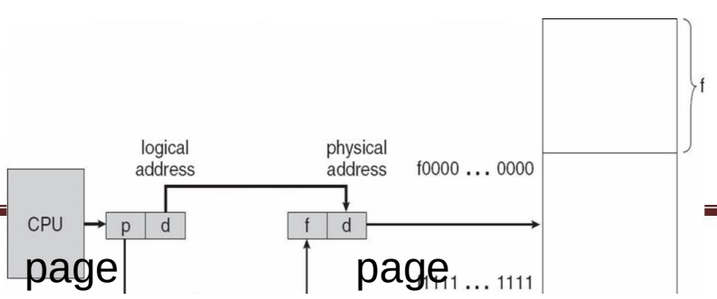
Address generated by CPU is divided into

Page number (*p*) – used as an index into a *pagetable* which contains base address of each page in physical memory

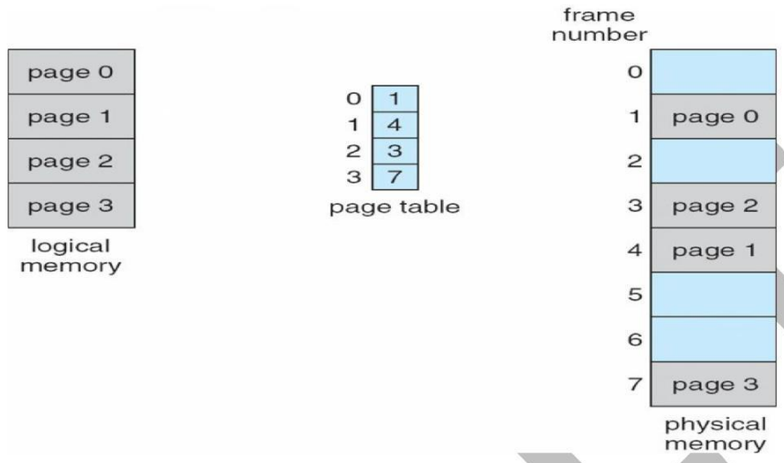
Page offset (d) – combined with base address to define the physical memory address that is sent to the memory unit

For given logical address space 2*m and page size2*n

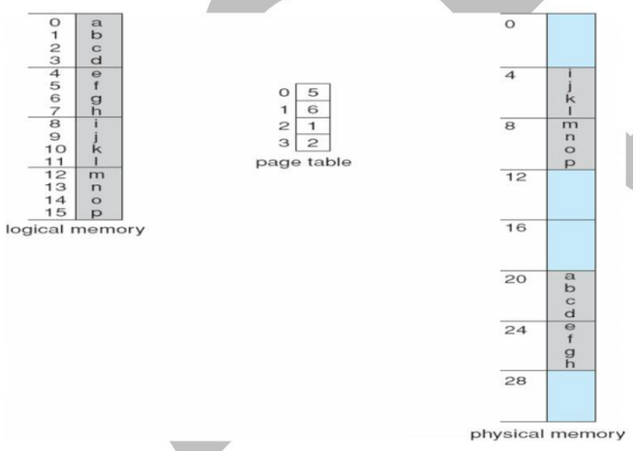
<u>Paging Hardware</u>



Paging Model of Logical and Physical Memory



Paging Example



Structure of the Page Table

In this section, we explore some of the most common techniques for structuring the page table.

1. Hierarchical Paging

Most modern computer systems support a large logical address space (2^{32} to 2^{64}). In such an environment, the page table itself becomes excessively large.

For example, consider a system with a 32-bit logical address space. If the page size in such a system is 4 KB (2^{12}), then a page table may consist of up to 1 million entries ($2^{32}/2^{12}$). Assuming that each entry consists of 4 bytes, each process may need up to 4 MB of physical address space for the page table alone. Clearly, we would not want to allocate the page table contiguously in main memory.

One simple solution to this problem is to divide the page table into smaller pieces. We can accomplish this division in several ways. One way is to use a two-level paging algorithm, in which the page table itself is also paged. Remember our example of a 32-bit machine

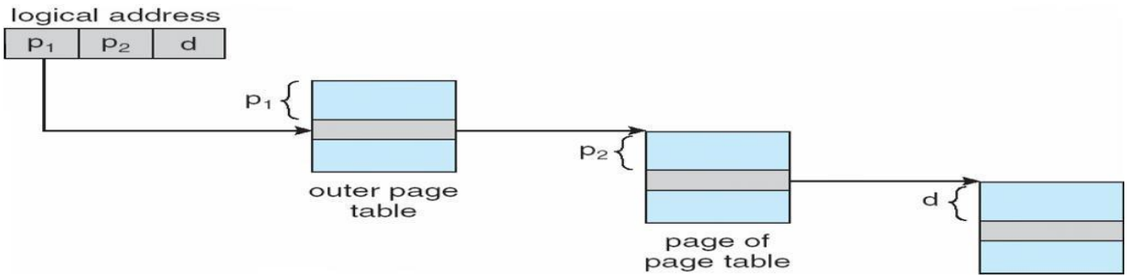
A two-level page-table scheme.

with a page size of 4 KB. A logical address is divided into a page number consisting of 20 bits and a page offset consisting of 12 bits. Because we page the page table, the page number is further divided into a 10-bit page number and a 10-bit page offset. Thus, a logical address is as follows:

page number		page offset
$P1$	p_2	d

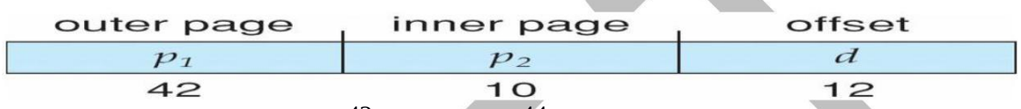
10 10 12

where p1is an index into the outer page table and *p2* is the displacement within the page of the outer page table. The address-translation method for this architecture is shown in Figure. Because address translation works from the outer page table inward, this scheme is also known as a forward-mapped page table.



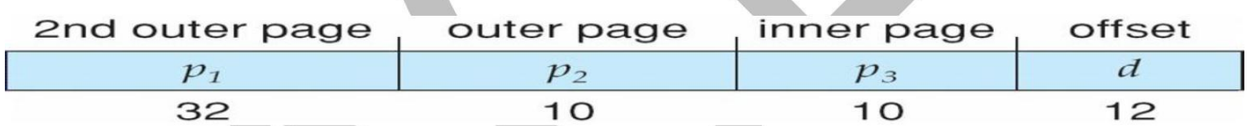
Address translation for a two-level 32-bit paging architecture.

For a system with a 64-bit logical-address space, a two-level paging scheme is no longer appropriate. To illustrate this point, let us suppose that the page size in such a system, is 4 KB (2^{12}). In this case, the page table consists of up to 2^{52} entries. If we use a two-level paging scheme, then the inner page tables can conveniently be one page long, or contain 2^{10} 4-byte entries. The addresses look like this:



The outer page table consists of 2^{42} entries, or 2^{44} bytes. The obvious way to avoid such a large table is to divide the outer page table into smaller pieces. This approach is also used on some 32-bit processors for added flexibility and efficiency.

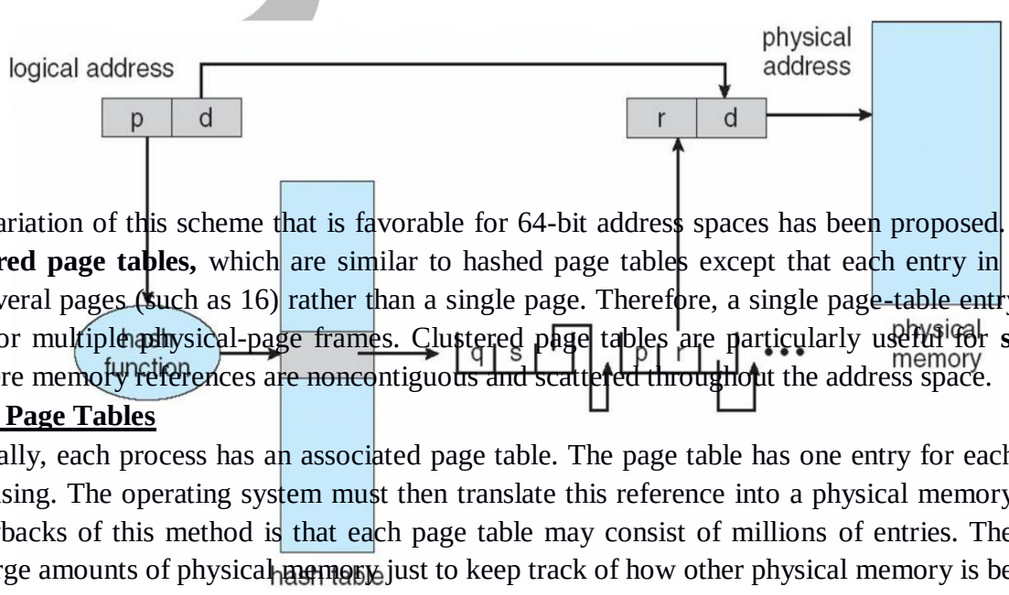
We can divide the outer page table in various ways. We can page the outer page table, giving us a three-level paging scheme. Suppose that the outer page table is made up of standard-size pages (2^{32} entries, or 2^{12} bytes); a 64-bit address space is still daunting: The outer page table is still 2^{32} bytes in size.



2. Hashed Page Tables

A common approach for handling address spaces larger than 32 bits is to use a **hashed** page table, with the hash value being the virtual page number. Each entry in the hash table contains a linked list of elements that hash to the same location (to handle collisions). Each element consists of three fields: (1) the virtual page number, (2) the value of the mapped page frame, and (3) a pointer to the next element in the linked list.

The algorithm works as follows: The virtual page number in the virtual address is hashed into the hash table. The virtual page number is compared with field 1 in the first element in the linked list. If there is a match, the corresponding page frame (field 2) is used to form the desired physical address. If there is no match, subsequent entries in the linked list are searched for a matching virtual page number. This scheme is shown in Figure.

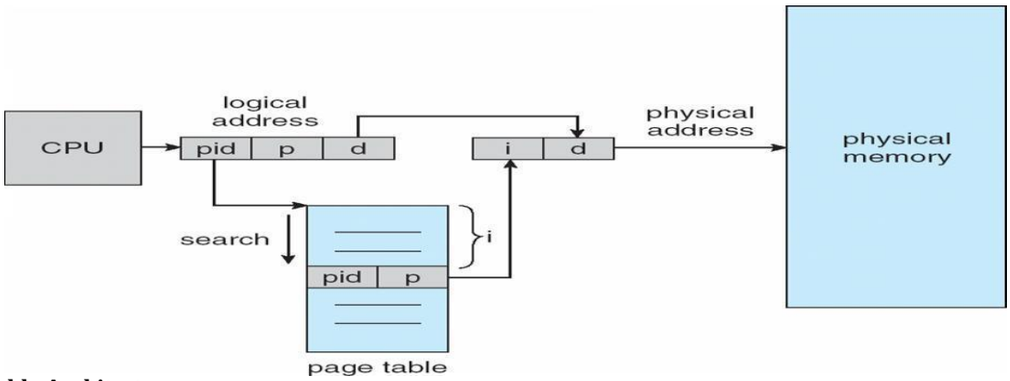


A variation of this scheme that is favorable for 64-bit address spaces has been proposed. This variation uses **clustered page tables,** which are similar to hashed page tables except that each entry in the hash table refers to several pages (such as 16) rather than a single page. Therefore, a single page-table entry can store the mappings for multiple physical-page frames. Clustered page tables are particularly useful for **sparse** address spaces, where memory references are noncontiguous and scattered throughout the address space.

3. Inverted Page Tables

Usually, each process has an associated page table. The page table has one entry for each page that the process is using. The operating system must then translate this reference into a physical memory address. One of the drawbacks of this method is that each page table may consist of millions of entries. These tables may consume large amounts of physical memory just to keep track of how other physical memory is being used.

To solve this problem, we can. use an **inverted page table.** An inverted page table has one entry for each real page (or frame) of memory. Each entry consists of the virtual address of the page stored in that real memory location, with information about the process that owns that page. Thus, only one page table is in the system, and it has only one entry for each page of physical memory. Figure shows the operation of an inverted page table.



Inverted Page Table Architecture

Each virtual address in the system consists of a triplet <process-id, page-number, offset>.

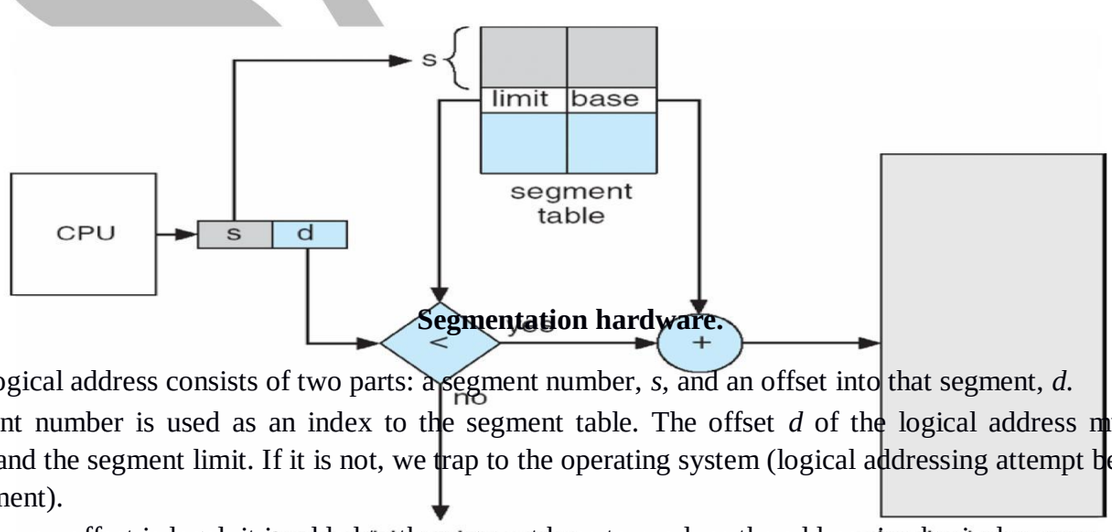
Each inverted page-table entry is a pair <process-id, page-number> where the process-id assumes the role of the address-space identifier. When a memory reference occurs, part of the virtual address, consisting of <process-id, pagenumber>, is presented to the memory subsystem. The inverted page table is then searched for a match. If a match is found—say; at entry i—then the physical address <*i*, offset> is generated. If no match is found, then an illegal address access has been attempted.

Although this scheme decreases the amount of memory needed to store each page table, it increases the amount of time needed to search the table when a page reference occurs. Because the inverted page table is sorted by physical address, but lookups occur on virtual addresses, the whole table might need to be searched for a match. This search would take far too long.

5. Segmentation

Segmentation is a memory-management scheme that supports this user view of memory. A logical address space is a collection of segments. Each segment has a name and a length. The addresses specify both the segment name and the offset within the segment. The user therefore specifies each address by two quantities: a segment name and an offset For simplicity of implementation, segments are numbered and are referred to by a segment number, rather than by a segment name. Tluis, a logical address consists of a *two tuple:* < segment-number, offset >.

Address mapping is effected by a segment table. Each entry in the segment table has a *segment base* and a *segment limit.* The segment base contains the starting physical address where the segment resides in memory, whereas the segment limit specifies the length of the segment. The use of a segment table is illustrated in Figure.



Segmentation hardware.

A logical address consists of two parts: a segment number, *s,* and an offset into that segment, *d.* The segment number is used as an index to the segment table. The offset *d* of the logical address must be between 0 and the segment limit. If it is not, we trap to the operating system (logical addressing attempt beyond, end of segment).

When an offset is legal, it is added to the segment base to produce the address in physical memory of the desired byte. The segment table is thus essentially an array of base-limit register pairs.

VIRTUAL MEMORY MANAGEMENT

1.Virtual memory

Virtual memory is a technique that allows the execution of processes that are not completely in memory. One major advantage of this scheme is that programs can be larger than physical memory. This technique frees programmers from the concerns of memory limitations.

The entire process *to* be executed must be in physical memory is reasonable, but infact the entire program is not needed. For instance, consider the following:

• Programs often have code to handle unusual error conditions. Since these errors rarely occur in practice, this code is almost never executed.

• Arrays, lists, and tables are often allocated more memory than they actually need.

• Certain options and features of a program may be used rarely.

Even in those cases where the entire program is needed, it may not all be needed at the same time. The ability to execute a program that is partially in memory would provide many benefits:

• A program would no longer be constrained by the amount of physical memory that is available.

Because each user program could take less physical memory, more programs could be run at the same time, with a corresponding increase in CPU utilization and throughput but with no increase in response time or turnaround time.

• Less I/O would be needed to load or swap each user program into memory, so each user program would run faster.

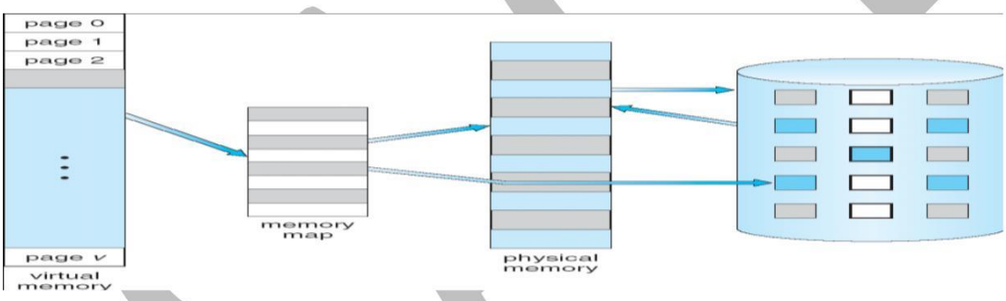
Thus, running a program that is not entirely in memory would benefit both the system and the user.

Virtual memory also allows files and memory to be shared by two or more processes Through page sharing. This leads to the following benefits:

• System libraries can be shared by several processes through mapping of the shared object into a virtual address space.

• Similarly, virtual memory enables processes to share memory. Virtual memory allows one process to create a region of memory that it can share with another process.

• Virtual memory can allow pages to be shared during process creation with the forkO system call, thus speeding up process creation.

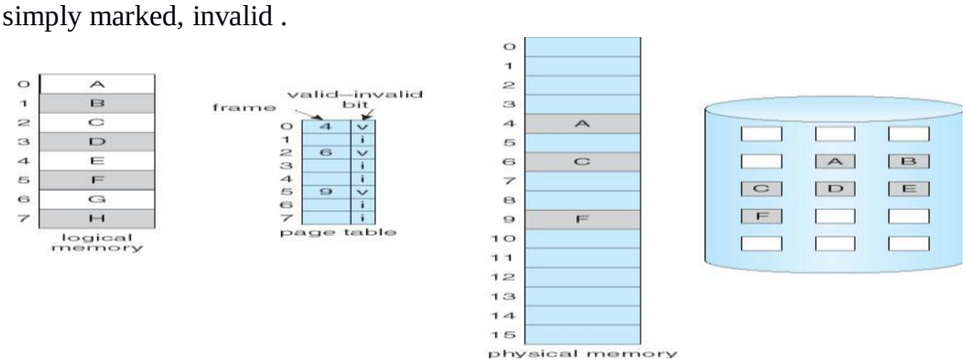


2.Demand Paging

Load the entire program in physical memory at program execution time is called pre paging. Load pages only as they are needed are known as demand paging and is commonly used in virtual memory systems. With demand-paged virtual memory, pages are only loaded when they are demanded during program execution

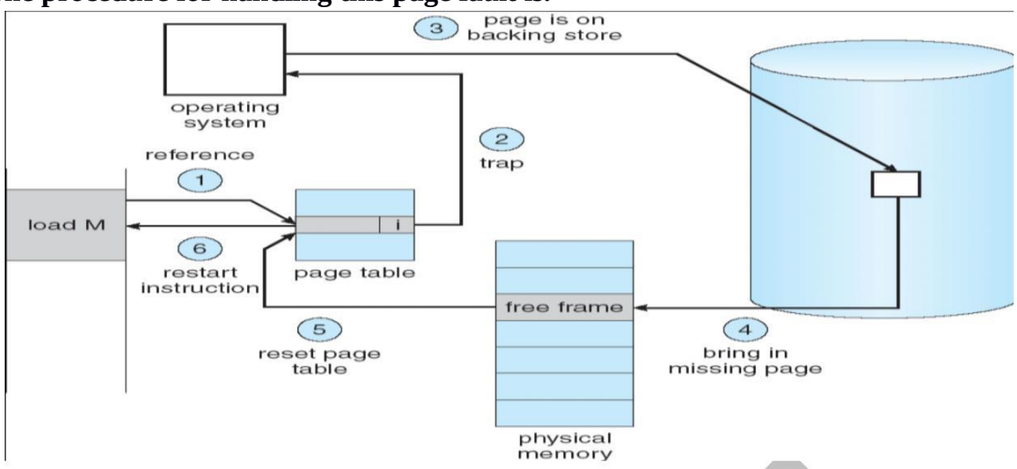
In this, we need some form of hardware support to distinguish between the pages that are in memory and the pages that are on the disk. The valid-invalid bit can be used for this purpose.

When this bit is set to "valid/" the associated page is both legal and in memory. If the bit is set to "invalid," the page either is not valid or is valid but is currently on the disk. The page-table entry for a page that is brought into memory is set as usual, but the page-table entry for a page that is not currently in memory is either simply marked, invalid .



Page table when some pages are not in main memory.

Notice that marking a page invalid will have no effect if the process never attempts to access that page. But if the process tries to access a page that was not brought into memory, it causes a page-fault trap. The procedure for handling this page fault is.



Steps in Handling Page Replacement

8. We check an internal table for this process to determine whether the reference was a valid or an invalid memory access.

9. If the reference was invalid, we terminate the process. If it was valid, but we have not yet brought in that page, we now page it in.

10. We find a free frame (by taking one from the free-frame list, for example).

11. We schedule a disk operation to read the desired page into the newly allocated frame.

12. When the disk read is complete, we modify the internal table kept with the process and the page table to indicate that the page is now in memory.

13. We restart the instruction that was interrupted by the trap.

The process can now access the page as though it had always been in memory.

In the extreme case, we can start executing a process with *no* pages in memory. When the operating system sets the instruction pointer to the first instruction of the process, which is on a disk, the page is brought into memory, the process continues to execute, faulting as necessary until every page that it needs is in memory. This scheme is pure demand paging.

If the page fault occurs on the instruction fetch, we can restart by fetching the instruction again. If a page fault occurs while we are fetching an operand, we must fetch and decode the instruction again and then fetch the operand

Performance of Demand Paging

Demand paging can significantly affect the performance of a computer system. As long as we have no page faults, the effective access time is equal to the memory access time. If, however, a page fault occurs, we must first read the relevant page from disk and then access the desired word.

Let *p* be the probability of a page fault (0 *s p* 5 1). We would expect *p* to be close to zero—that is, we would expect to have only a few page faults. The effective access time is then

Effective access time = (1 - *p)* x *ma* + *p* x page fault time.

To compute the effective access time, we must know how much time is needed to service a page fault. A page fault causes the following sequence to occur:

10. Service the page-fault interrupt.

11. Read in the page.

12. Restart the process.

The first and third tasks can be reduced, with careful coding, to several hundred instructions. These tasks may take from 1 to 100 microseconds each. The page-switch time, however, will probably be close to 8 milliseconds and a memory-access time of 200 nanoseconds, then the effective access time is

Effective access time = (1 - *p)* x (200) + *p* (8 milliseconds)

- (1 - *p)* x 200 + *p x* 8.00(1000
- 200 + 7,999,800 x *p.*

We see that the effective access time is directly proportional to the page-fault rate. The computer will be slowed down because of demand paging.

Copy-on-Write

Process creation using the fork () system call may initially bypass the need for demand paging by using a technique similar to page sharing .The fork() system call creates a child process as a duplicate of its parent. We can use a technique known as copy-on-write, which works by allowing the parent and child processes initially to share the same pages. These shared pages are marked as copy-on-write pages, meaning that if either process writes to a shared page, a copy of the shared page is created. Copy-on-write is illustrated in Figures which show the contents of the physical memory before and after process 1 modifies page C.

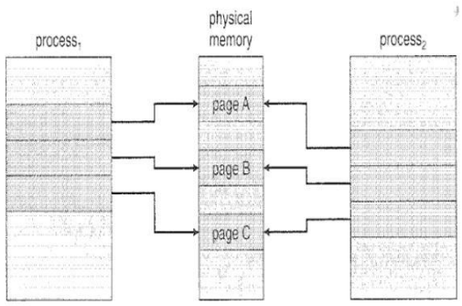


Figure 9.7 Before process 1 modifies page C.

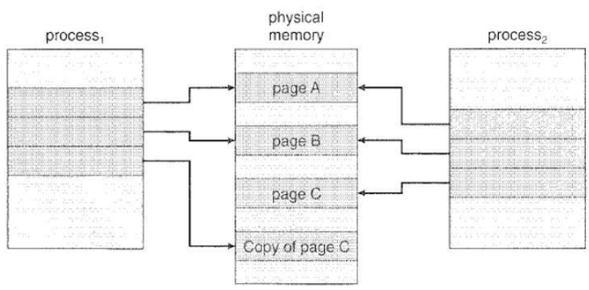


Figure 9.8 After process 1 modifies page C.

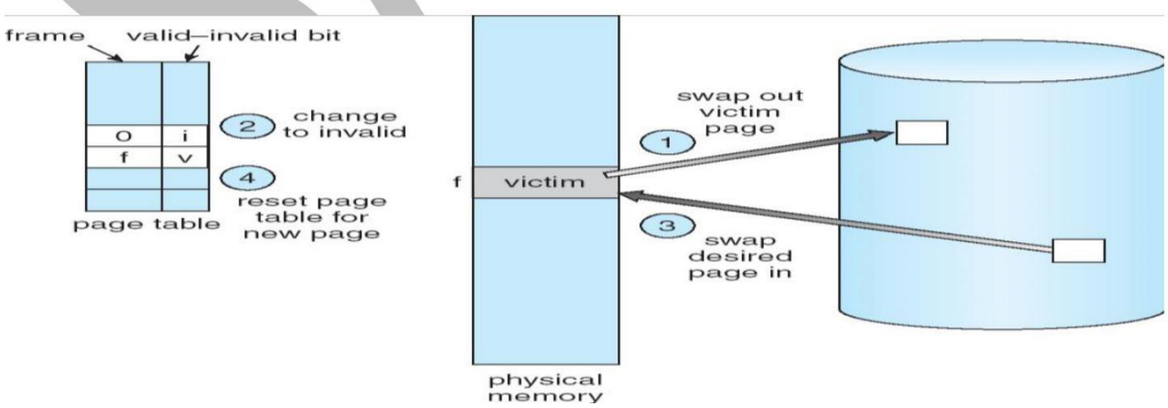
For example, assume that the child process attempts to modify a page containing portions of the stack, with the pages set to be copy-on-write. The operating system will then create a copy of this page, mapping it to the address space of the child process. The child process will then modify its copied page and not the page belonging to the parent process. Note, that only pages that can be modified need be marked as copy-onwrite. Pages that cannot be modified (pages containing executable code) can be shared by the parent and child.

Many operating systems provide a **pool** of free pages for such requests. Operating systems typically allocate these pages using a technique known as zero-fill-on-demand.

3.Page Replacement

While a user process is executing, a page fault occurs. The operating system finds that there are *no* free frames on the free-frame list; all memory is in use. Then we follow the most common solution: page replacement. Page replacement takes the following approach.

- Find the location of the desired page on the disk.
- Find a free frame:
  - If there is a free frame, use it.
  - If there is no free frame, use a page-replacement algorithm to select victim frame.
  - Write the victim frame to the disk; change the page and frame tables accordingly.
- Read the desired page into the newly freed frame; change the page and frame tables.
- **4. Restart the user process.**



Page Replacement

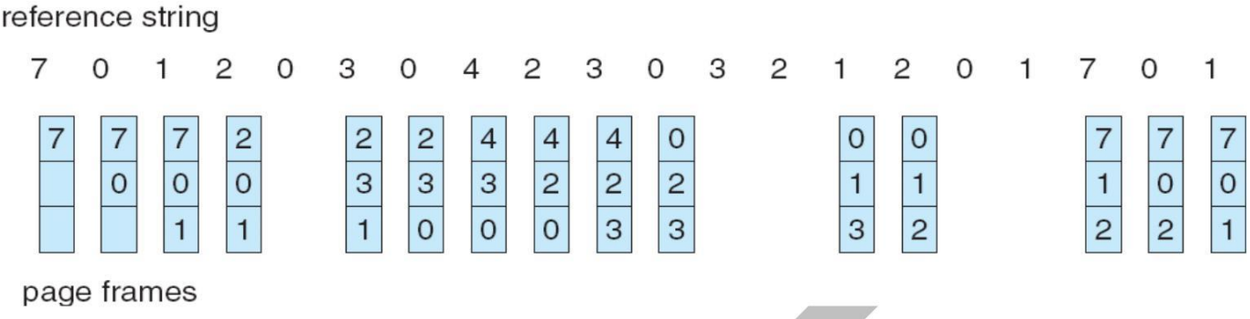
4.Page Replacement Algorithms

If we have multiple processes in memory, we must decide how many frames to allocate to each process. Further, when page replacement is required, we must select the frames that are to be replaced. There are many different page-replacement algorithms. We want the one with the lowest page-fault rate. We evaluate an algorithm by running it on a particular string of memory references and computing the number of page faults. The string of memory references is called a **reference string.** the number of frames available increases, the

number of page faults decreases.We illustrate several page-replacement algorithms. We use the reference string
7, 0,1, 2, 0, 3, 0, 4, 2, 3, 0, 3, 2,1, 2, 0, 1, 7, 0,1
for a memory with three frames.

FIFO Page Replacement

It is the simplest page-replacement algorithm. A FIFO replacement algorithm associates with each page the time when that page was brought into memory. When a page must be replaced, the oldest page is chosen. We can. Create a FIFO queue to hold all pages in memory. We replace the page at the head of the queue. When a page is brought into memory, we insert it at the tail of the queue.



For our example reference string, our three frames are initially empty. The first three references (7,0,1) cause page faults and are brought into these empty frames. The next reference (2) replaces page 7, because page 7 was brought in first. Since 0 is the next reference and 0 is already in memory, we have no fault for this reference. The first reference to 3 results in replacement of page 0, since it is now first in line. Because of this replacement, the next reference, to 0, will fault. Page 1 is then replaced by page 0. This process continues as shown in Figure. Every time a fault occurs, we show which pages are in our three frames. There are 15 faults altogether.

The FIFO page-replacement algorithm is easy to understand and program. But if we select for replacement a page that is in active use, everything still works correctly. After we replace an active page with a new one, a fault occurs almost immediately to retrieve the active page. Some other page will need to be replaced to bring the active page back into memory. Thus, a bad replacement choice increases the page-fault rate and slows process execution. To illustrate the problems, we consider the following reference string:

1,2,3,4,1,2,5,1,2,3,4,5

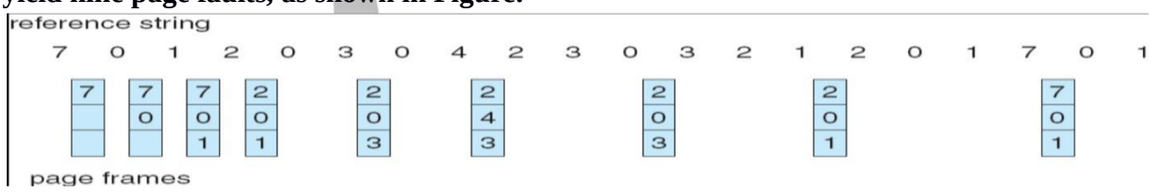
Notice that the number of faults for four frames (ten) is *greater* than the number of faults for three frames (nine)! This most unexpected result is known as Belady's **anomaly:** For some page-replacement algorithms, the page-fault rate may *increase* as the number of allocated frames increases.

Optimal Page Replacement

An optimal page-replacement algorithm has the lowest page-fault rate of all algorithms and will never suffer from Belady's anomaly. Such an algorithm does exist and has been called OPT or MIN. It is simply this:

Replace the page that will not be used for the longest period of time.

Use of this page-replacement algorithm guarantees the lowest possible page fault rate for a fixed number of frames. For example, on our sample reference string, the optimal page-replacement algorithm would yield nine page faults, as shown in Figure.



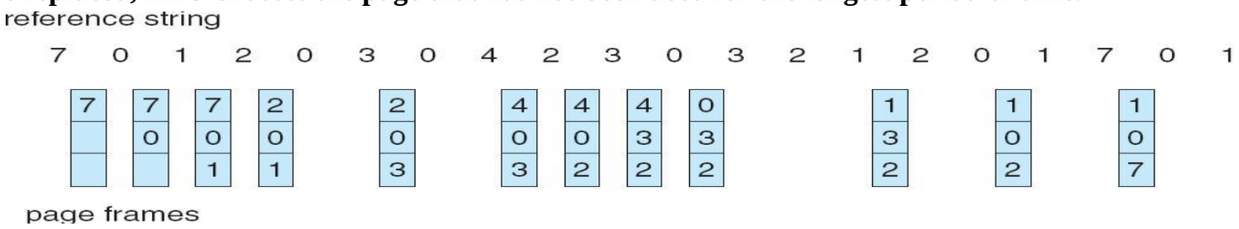
The first three references cause faults that fill the three empty frames. The reference to page 2 replaces page 7, because 7 will not be used until reference 18, whereas page 0 will be used at 5, and page 1 at 14. The reference to page 3 replaces page 1, as page 1 will be the last of the three pages in memory to be referenced again. With only nine page faults, optimal replacement is much better than a FIFO algorithm, which resulted in fifteen faults.

Unfortunately, the optimal page-replacement algorithm is difficult to implement, because it requires future knowledge of the reference string.

LRU Page Replacement

This approach is the least-recently-used (LRU) algorithm. In this, it uses the recent past as an approximation of the near future, then we can replace the page that *has not been used* for the longest

period of time. LRU replacement associates with each page the time of that page's last use. When a page must be replaced, LRU chooses the page that has not been used for the longest period of time.



The result of applying LRU replacement to our example reference string is shown in Figure. The LRU algorithm produces 12 faults. Notice that the first 5 faults are the same as those for optimal replacement. When the reference to page 4 occurs, however, LRU replacement sees that, of the three frames in memory, page 2 was used least recently. Thus, the LRU algorithm replaces page 2, not knowing that page 2 is about to be used. When it then faults for page *2,* the LRU algorithm replaces page 3, since it is now the least recently used of the three pages in memory. Despite these problems, LRU replacement with 12 faults is much better than FIFO replacement with 15.

The major problem is *how* to implement LRU replacement. An LRU page-replacement algorithm may require substantial hardware assistance. The problem is to determine an order for the frames defined by the time of last use. Two implementations are feasible:

• **Counters**. In this, we associate with each page-table entry a time-of-use field and add to the CPU a logical clock or counter. The clock is incremented for every memory reference. Whenever a reference to a page is made, the contents of the clock register are copied to the time-of-use field in the page-table entry for that page. In this way, we always have
the "time" of the last reference to each page. We replace the page with the smallest time value.

• **Stack.** Another approach to implementing LRU replacement is to keep a stack of page numbers. Whenever a page is referenced, it is removed from the stack and put on the top. In this way, the most recently used page is always at the top of the stack and the least recently used page is always at the bottom.

Like optimal replacement, LRU replacement does not suffer from Belady's anomaly.

Additional-Reference-Bits Algorithm

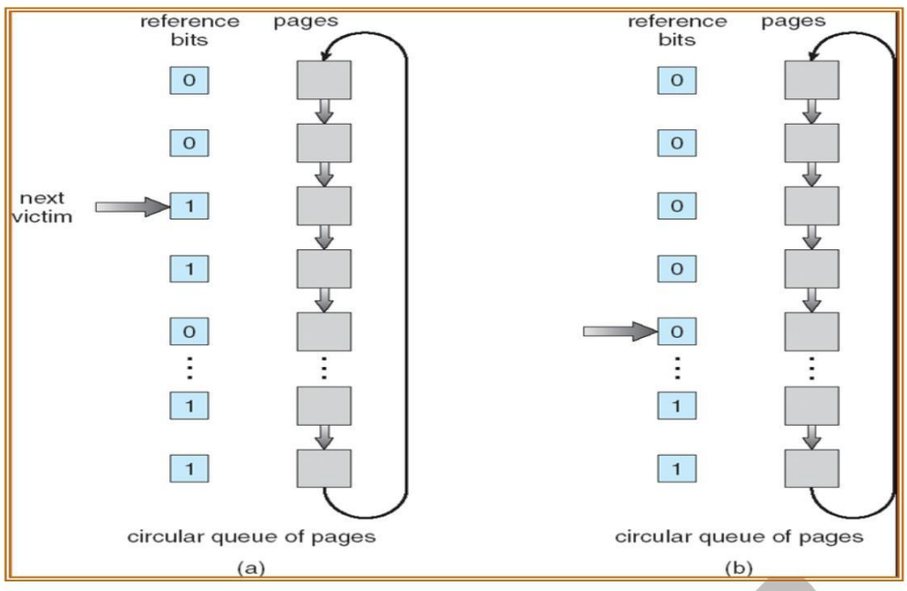
We can gain additional ordering information by recording the reference bits at regular intervals. We can keep an 8-bit byte for each page in a table in memory. At regular intervals (say, every 100 milliseconds), a timer interrupt transfers
control to the operating system. The operating system shifts the reference bit for each page into the high-order bit of its 8-bit byte. These 8-bit shift registers contain the history of page use for the last eight time periods. If the shift register contains 00000000, for example, then the page has not been used for eight time periods; a page that is used at least once in each period has a shift register value of 11111111. A page with a history register value of 11000100 has been used more
recently than one with a value of 01110111.

The number of bits of history can be varied. In the extreme case, the number can be reduced to zero, leaving
only the reference bit itself. This algorithm is called the second-chance page **replacement algorithm.**

Second-Chance Algorithm

The basic algorithm of second-chance replacement is a FIFO replacement algorithm. When a page has been selected, however, we inspect its reference bit. If the value is 0, we proceed to replace this page; but if the reference bit is set to 1, we give the page a second chance and move on to select the next FIFO page. When a page gets a second chance, its reference bit is cleared, and its arrival time is reset to the current time. Thus, a page that is given a second chance will not be replaced until all other pages have been replaced

One way to implement the second-chance algorithm is as a circular queue. A pointer indicates which page is to be replaced next. When a frame is needed, the pointer advances until it finds *a* page with a 0 reference bit. Once a victim page is found, the page is replaced, and the new page is inserted in the circular queue in that position.



Second-Chance (clock) Page-Replacement Algorithm

Enhanced Second-Chance Algorithm

We can enhance the second-chance algorithm by considering the reference bit and the modify bit as an ordered pair. With these two bits, we have the following four possible classes:

- (0, 0) neither recently used nor modified—best page to replace
- (0, 1) not recently used but modified—not quite as good, because the page will need to be written out before replacement
- (1., 0) recently used but clean—probably will be used again soon
- (1,1) recently used and modified—probably will be used again soon, and the page will be need to be written out to disk before it can be replaced

Each page is in one of these four classes. When page replacement is called for, we use the same scheme as in the clock algorithm; but instead of examining whether the page to which we are pointing has the reference bit set to 1, we examine the class to which that page belongs. We replace the first page encountered in the lowest nonempty class. Notice that we may have to scan the circular queue several times before we find a page to be replaced.

Counting-Based Page Replacement

There are many other algorithms that can be used for page replacement. For example, we can keep a counter of the number of references that have been made to each page and develop the following two schemes.

- The **least frequently used (LFU) page-replacement algorithm** requires that the page with the smallest count be replaced. The reason for this selection is that an actively used page should have a large reference count.
- The **most frequently used** (MFU) page-replacement algorithm is based on the argument that the page with the smallest count was probably just brought in and has yet to be used.

As you might expect, neither MFU nor LFU replacement is common. The implementation of these algorithms is expensive.

- **Allocation of Frames**

We discuss the issue of allocate the fixed amount of free memory among the various processes. The minimum number of frames per process is defined by the architecture, the maximum number is defined by the amount of available physical memory. There are two allocation algorithms used.

Allocation Algorithms

The easiest way to split *in* frames among *n* processes is to give everyone an equal share, *m/n* frames. For instance, if there are 93 frames and five processes, each process will get 18 frames. The leftover three frames can be used as a free-frame buffer pool. This scheme is called equal allocation.

An alternative is to recognize that various processes will need differing amounts of memory. We can use **proportional** allocation, in which we allocate available memory to each process according to its size. Let the size of the virtual memory for process *pi* be *si* and define

$S=\sum si$

Then, if the total number of available frames is *m,* we allocate *ai* frames to process pi,-, where *ai* is approximately

ai = Si/S x *m.*

Of course, we must adjust each *ai* to be an integer that is greater than the minimum number of frames required by the instruction set, with a sum not exceeding *m.*

In this way, both processes share the available frames according to their "needs," rather than equally.
In both equal and proportional allocation, of course, the allocation may vary according to the multiprogramming level. If the multiprogramming level is increased, each process will lose some frames to provide the memory needed for the new process. Conversely, if the multiprogramming level decreases, the frames that were allocated to the departed process can be spread over the remaining processes.

In either equal or proportional allocation, a high-priority process is treated the same as a low-priority process. By its definition, however, we may want to give the high-priority process more memory to speed its execution, to the detriment of low-priority processes. One solution is to use a proportional allocation scheme wherein the ratio of frames depends not on the relative sizes of processes but rather on the priorities of processes or on a combination of size and priority.

Global versus Local Allocation

Another important factor in the way frames are allocated to the various processes is page replacement. With multiple processes competing for frames, we can classify page-replacement algorithms into two broad categories: **global replacement** and **local replacement.**

Global replacement allows a process to select a replacement frame from the set of all frames, even if that frame is currently allocated to some other process; that is, one process can take a frame from another.
Local replacement requires that each process select from only its own set of allocated frames.

A process can select a replacement from among its own frames or the frames of any lower-priority process. This approach allows a high-priority process to increase its frame allocation at the expense of a low-priority process.

With a local replacement strategy, the number of frames allocated to a process does not change. With global replacement, a process may happen to select only frames allocated to other processes, thus increasing the number of frames allocated to it.

One problem with a global replacement algorithm is that a process cannot control its own page-fault rate. The set of pages in memory for a process depends not only on the paging behavior of that process but also on the paging behavior of other processes. Therefore, the same process may perform quite differently.

Under local replacement, the set of pages in memory for a process is affected by the paging behavior of Only that process. Local replacement might hinder a process, however, by not making available to it other, less used pages of memory.

Thus, global replacement generally results in greater system throughput and is therefore the more common method.

6. Thrashing

If the number of frames allocated to a low-priority process falls below the minimum number required by the computer architecture, we must suspend that process's execution. We should then page out its remaining pages, freeing all its allocated frames. This provision introduces a swap-in, swap-out level of intermediate CPU scheduling.

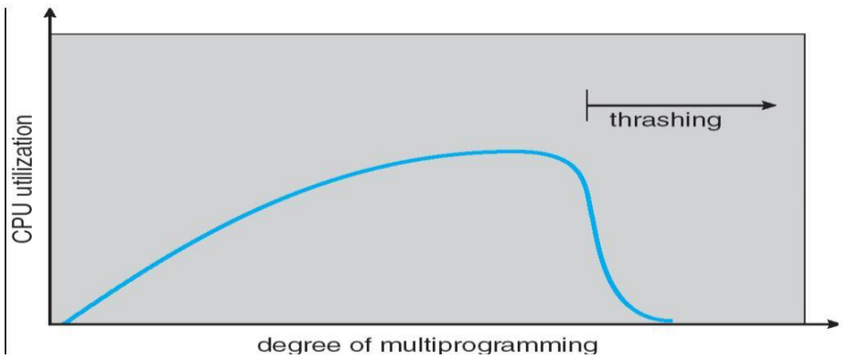
If the process does not have the number of frames it needs to support pages in active use, it will quickly page-fault. At this point, it must replace some page. However, since all its pages are in active use, it must replace a page that will be needed again right away. Consequently, it quickly faults again, and again, and again, replacing pages that it must bring back in immediately.

This high paging activity is called **thrashing.** A process is thrashing if it is spending more time paging than executing.

Cause of Thrashing

Thrashing results in severe performance problems. If CPU utilization is too low, we increase the degree of multiprogramming by introducing a new process to the system. A global page-replacement algorithm is used; it replaces pages without regard to the process to which they belong. Now suppose that a process enters a new phase in its execution and needs more frames. It starts faulting and taking frames away from other processes. These processes need those pages, however, and so they also fault, taking frames from other processes. These faulting processes must use the paging device to swap pages in and out. As they queue up for the paging device,the ready queue empties. As processes wait for the paging device, CPU utilization decreases.

SACET



Demand Paging and Thrashing

This phenomenon is illustrated in Figure in which CPU utilization is plotted against the degree of multiprogramming. As the degree of multiprogramming increases, CPU utilization also increases, until a maximum is reached. If the degree of multiprogramming is increased even further, thrashing sets in, and CPU utilization drops sharply. At this point, to increase CPU utilization and stop thrashing, we must *decrease* the degree of multiprogramming.

We can limit the effects of thrashing by using a **local replacement algorithm** (or **priority replacement algorithm).** With local replacement, if one process starts thrashing, it cannot steal frames from another process and cause the latter to thrash as well.

To prevent thrashing, we must provide a process with as many frames as it needs.

File-System Interface

1 File Concept

A file is a named collection of related information that is recorded on secondary storage.
Files represent programs and data.

File Attributes

A file's attributes vary from one operating system to another but typically consist of these:

• **Name**. The symbolic file name is the only information kept in humanreadable form,.

• **Identifier**. This is usually a number, identifies the file within the file system; it is the non-human-readable name for the file.

• **Type**. This information is needed for systems that support different types of files.

• **Location**. This information is a pointer to a device and to the location of the file on that device.

• **Size**. The current size of the file (in bytes, words, or blocks) and possibly the maximum allowed size are included in this attribute.

• **Protection**. Access-control information determines who can do reading, writing, executing, and so on.

• **Time, date, and user identification**. This information may be kept for creation, last modification, and last use. These data can be useful for protection, security, and usage monitoring.

The information about all files is kept in the directory structure. Typically, a directory entry consists of the file's name and its unique identifier. The identifier in turn locates the other file attributes. It may take more than a kilobyte to record this information for each. file. In a system with many files, the size of the directory itself may be megabytes.

File Operations

A file is an **abstract data type.** We need to consider the operations that can be performed on files. The operating system can provide system calls to create, write, read, reposition, delete, and truncate files.

• Creating **a file.** Two steps are necessary to create a file. First, space in the file system must be found for the file. Second, an entry for the new file must be made in the directory.

• **Writing a file.** To write a file, we make a system call specifying both the name of the file and the information to be written to the file. Given the name of the file, the system searches the directory to find the file's location. The system must keep a *write* pointer to the location in the file where the next write is to take place.

• **Reading a file.** To read from a file, we use a system call that specifies the name of the file and where (in memory) the next block of the file should be put. Again, the directory is searched for the associated entry, and the system needs to keep a *read* pointer to the location in the file where the next read is to take place.

• Repositioning **within** a **file.** The directory is searched for the appropriate entry, and the current-file-position pointer is repositioned to a given value. This file operation is also known as a file *seek.*

• **Deleting a file.** To delete a file, we search the directory for the named file. Having found the associated directory entry, we release all file space, so that it can be reused bv other files, and erase the directory entry.

• **Truncating a file**. The user may want to erase the contents of a file but keep its attributes.

This function allows all attributes to remain unchanged—except for file length—be reset to zero and its file space released.

Most of the file operations mentioned involve searching the directory for the entry associated with the named file. To avoid this constant searching, many systems require that an open( ) system call be made before a file is first used actively. The operating system keeps a small table, called the **open-file table,** containing information about all open files. When a file operation is requested, the file is specified via an index into this table, so no searching is required.

When the file is no longer being actively used, it is *closed* by the process, and the operating system removes its entry from the open-file table, create and delete are system calls that work with closed rather than open files.

Fiie Types

A common technique for implementing file types is to include the type as part of the file name. The name is split into two parts—a name and an *extension,* usually separated by a period character. In this way, the user and the operating system can tell from the name alone what the type of a file is.

file type	usual extension	function
executable	exe, com, bin or none	ready-to-run machine-language program
object	obj, o	compiled, machine language, not linked
source code	c, cc, java, pas, asm, a	source code in various languages
batch	bat, sh	commands to the command interpreter
text	txt, doc	textual data, documents
word processor	wp, tex, rtf, doc	various word-processor formats
library	lib, a, so, dll	libraries of routines for programmers
print or view	ps, pdf, jpg	ASCII or binary file in a format for printing or viewing
archive	arc, zip, tar	related files grouped into one file, sometimes com-pressed, for archiving or storage
multimedia	mpeg, mov, rm, mp3, avi	binary file containing audio or A/V information

Internal File Structure

Disk systems have a well-defined block size determined by the size of a sector. All disk I/O is performed in units of one block (physical record), and all blocks are the same size. It is unlikely that the physical record size will exactly match the length of the desired logical record. Logical records may even vary in length. Packing a number of logical records into physical blocks is a common solution to this problem.

For example, the UNIX operating system defines all files to be simply streams of bytes. In this case, the logical record size is 1 byte. The file system automatically packs and unpacks bytes into physical disk blocks—say, 512 bytes per block—as necessary.

The logical record size, physical block size, and packing technique determine how many logical records are in each physical block. The packing can be done either by the user's application program or by the operating system.

Because disk space is always allocated in blocks, some portion of the last block of each file is generally wasted. If each block were 512 bytes, for example, then a file of 1,949 bytes would be allocated four blocks (2,048 bytes); the last 99 bytes would be wasted. The waste incurred is internal fragmentation. All file systems suffer from internal fragmentation; the larger the block size, the greater the internal fragmentation.

2. Access Methods

Files store information. The information in the file can be accessed in several ways.

Sequential Access

The simplest access method is sequential access. Information in the file is processed in order, one record after the other. This mode of access is used editors and compilers.

A read operation—*read next*—reads the next portion of the file and automatically advances a file pointer. Similarly, the write operation—*write next*—appends to the end of the file and advances to the end of the newly written material. Such a file can be reset to the beginning

Direct Access

Another method is **direct access** (or **relative** access). A file is made up of fixed length **logical records. It** allow programs to read and write records rapidly in no particular order.

For the direct-access method, the file operations must be modified to include the block number as a parameter. Thus, we have *read n,* where *n* is the block number, rather than *read next,* and *write n* rather than *write next. A*nd to add an operation *position file to n,* where *n* is the block number.

The block number provided by the user to the operating system is normally a **relative block number.** A relative block number is an index relative to the beginning of the file.

Not all operating systems support both sequential and direct access for files. Some systems allow only sequential file access; others allow only direct access. Some systems require that a file be defined as sequential or direct when it is created; such a file can be accessed only in a manner consistent with its declaration.

Other Access(Indexed) Methods

Other access methods can be built on top of a direct-access method. These methods generally involve the construction of an index for the file. The index, contains pointers to the various blocks. To find a record in the file, we first search the index and then use the pointer to access the file directly and to find the desired record.

For example, IBM's indexed sequential-access method (ISAM) uses a small master index that points to disk blocks of a secondary index. The secondary index blocks point to the actual file blocks.

3.Directory Structure

The file systems store millions of files. To manage all these files, we need directories to organize them. In this section, we explore the topic of directory structure. First, we explain some basic features of storage structure.

Storage Structure

Each volume contains information about the files in the system. This information is kept in entries in a **device directory.** The device directory records information—such as name, location, size, and type—for all files on that volume.

Directory Overview

The directory can be viewed as a symbol table of entries. We want to be able to insert entries, to delete entries, to search for a named entry, and to list all the entries in the directory.

• Create a file.New files need to be created and added to the directory.

• Delete a file. When a file is no longer needed, we want to be able to remove it from the directory.

• List a directory. We need to be able to list the files in a directory and the contents of the directory entry for each file in the list.

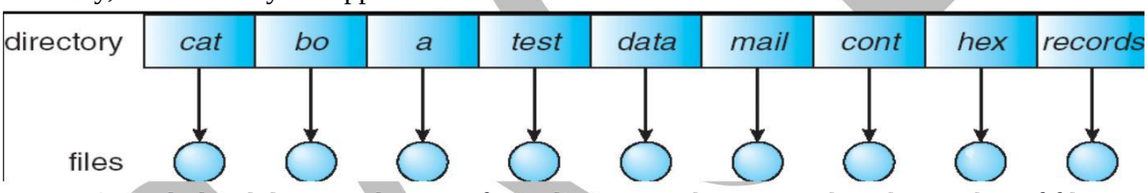
• Rename a file. we must be able to change the name when the contents or use of the file changes.

• Traverse the file system. We may wish to access every directory and every file within a directory structure.

In the following sections, we describe the most common schemes for defining the logical structure of a directory.

Single-Level Directory

The simplest directory structure is the single-level directory. All files are contained in the same directory, which is easy to support and understand



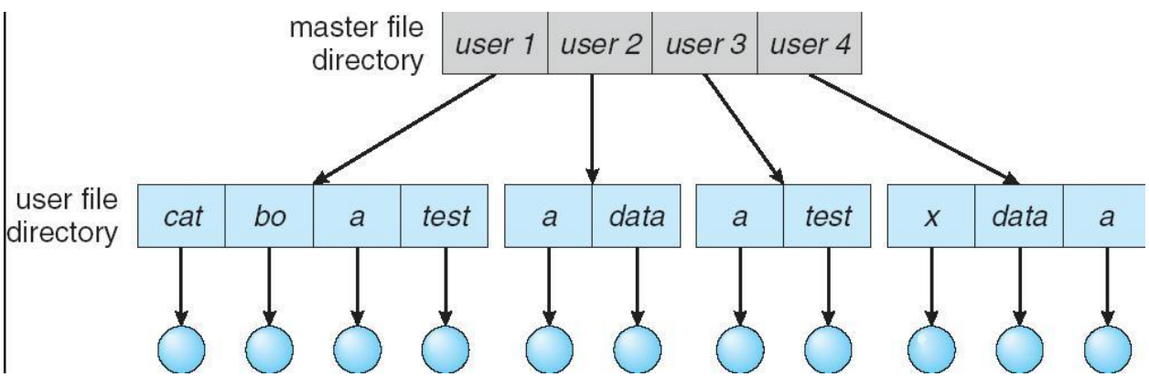
A single-level directory has significant limitations, however, when the number of files increases or when the system has more than one user. Since all files are in the same directory, they must have unique names.

Even a single user on a single-level directory may find it difficult to remember the names of all the files as the number of files increases.

Two-Level Directory

As we have seen, a single-level directory often leads to confusion of file names among different users. The standard solution is to create a *separate* directory for each user.

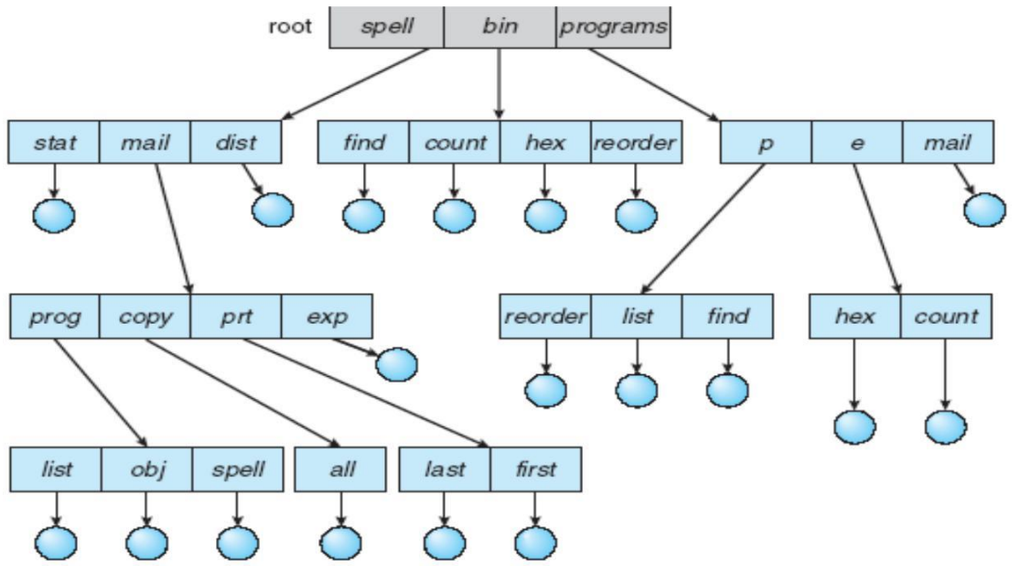
In the two-level directory structure, each user has his own user file directory (UFD). The UFDs have similar structures, but each lists only the files of a single user. When a user job starts or a user logs in, the system's master file directory (MFD) is searched. The MFD is indexed by user name or account number, and each entry points to the UFD for that user.When a user refers to a particular file, only his own UFD is searched. Thus, different users may have files with the same name, as long as all the file names within each UFD are unique.



Although the two-level directory structure solves the name-collision problem, it still has disadvantages. This structure effectively isolates one user from another. Isolation do not allow local user files to be accessed by other users.

Tree-Structured Directories

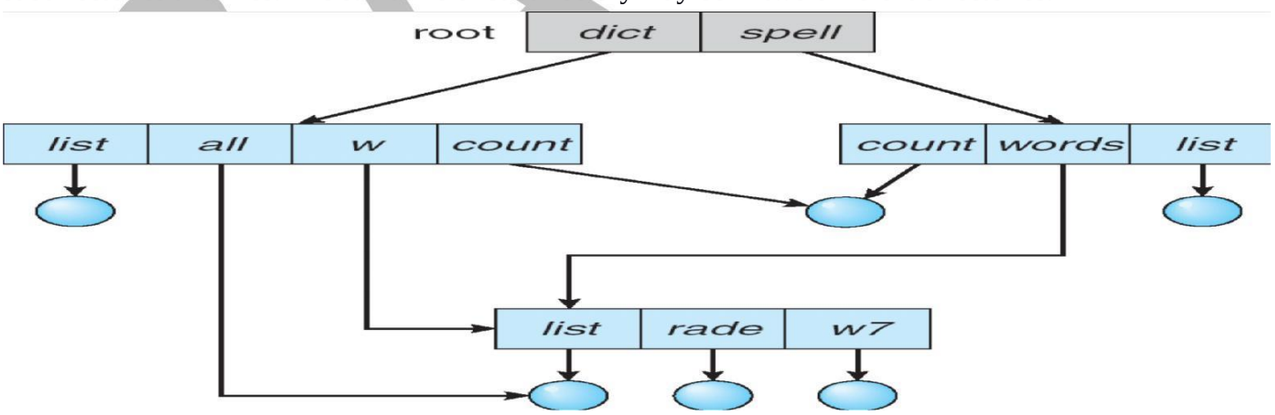
It is generalization of two-level directory, to extend the directory structure to a tree of arbitrary height. This generalization allows users to create their own subdirectories and to organize their files accordingly. A tree is the most common directory structure. A directory (or subdirectory) contains a set of files or subdirectories. A directory is simply another file, but it is treated in a special way. All directories have the same internal format. One bit in each directory entry defines the entry as a file (0) or as a subdirectory (1). Special system calls are used to create and delete directories.



With a tree-structured directory system, users can be allowed to access, the files of other users by specifying its path names.

Acyclic-Graph Directories

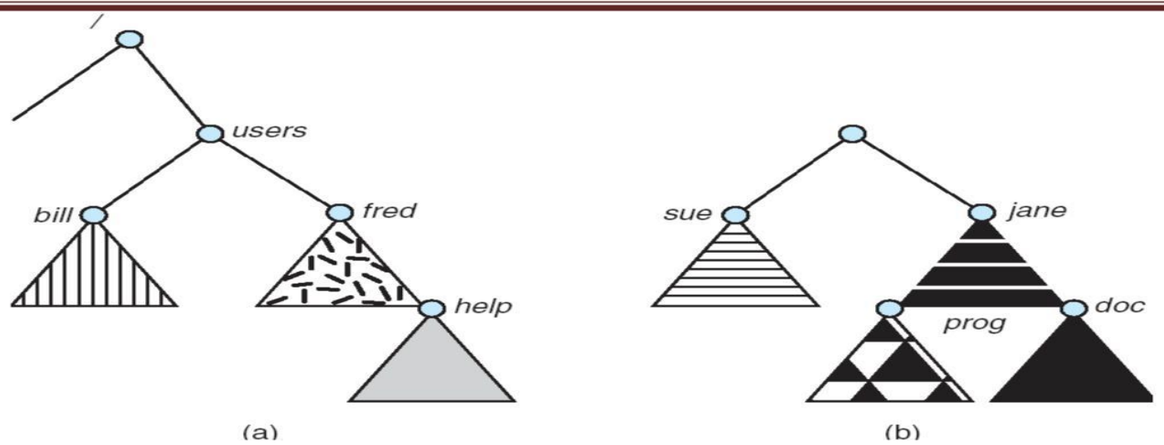
Consider two programmers who are working on a joint project. Both want the subdirectory to be *shared. A* shared directory *or* file will exist in the file system in two (or more) places at once. A tree structure prohibits the sharing of files or directories. An **acyclic graph** —that is, a graph with no cycles—allows directories to share subdirectories and files. The *same* file or subdirectory may be in two different directories.



The acyclic graph is a natural generalization of the tree-structured directory scheme. With a shared file, only *one* actual file exists, so any changes made by one person are immediately visible to the other. Sharing is particularly important for subdirectories; a new file created by one person will automatically appear in all the shared subdirectories.

4.File-System Mounting

A file system must be *mounted* before it can be available to processes on the system. The mount procedure is straightforward. The operating system is given the name of the device and the **mount point**—the location within the file structure where the file system is to be attached. Next, the operating system verifies that the device contains a valid file system. Finally, the operating system notes in its directory structure that a file system is mounted at the specified mount point.



To illustrate file mounting, consider the file system depicted in Figure. Here the triangles represent subtrees of directories that are of interest. Figure (a) shows an existing file system, while Figure (b) shows an unmounted volume residing on */device'/dsk.* At this point, only the files on the existing file system can be accessed.

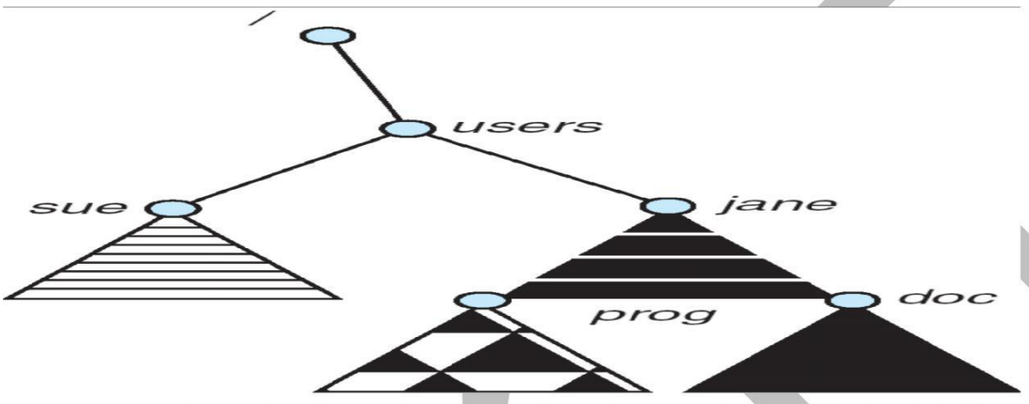


Figure shows the effects of mounting the volume residing on */device/disk* over */users.* If the volume is unmounted, the file system is restored to the previous situation depicted in Figure.

A system may allow the same file system to be mounted repeatedly, at different mount points; or it may only allow one mount per file system.

5. File Sharing

We begin by discussing general issues that arise when multiple users share files.

Multiple Users

To implement sharing among multiple users, the system must maintain more file and directory attributes than are needed on a single-user system. Most systems have use the concepts of file *owner* (or *user)* and *group.* The owner is the user who can change attributes and grant access over the file. The group attribute defines a subset of users who can share access to the file.

The owner and group IDs of a given file (or directory) are stored with the other file attributes. When a user requests an operation on a file, the user ID can be compared with the owner attribute to determine if the requesting user is the owner of the file. Likewise, the group IDs can be compared. The result indicates which permissions are applicable. The system then applies those permissions to the requested operation and allows or denies it.

Remote File Systems

Remote file systems allow a computer to mount one or more file systems from one or more remote machines. In this case, the machine containing the files is the *server,* and the machine seeking access to the files is the *client.* A server can serve multiple clients, and a client can use multiple servers, depending on the implementation details of a given client-server facility.

The server usually specifies the available files on a volume or directory level. Client identification is by using secure authentication of the client via encrypted keys.

Consistency Semantics

Consistency semantics represent an important criterion for file sharing. These semantics specify how multiple users of a system are to access a shared file simultaneously. These semantics are typically implemented as code with the file system.

UNIX Semantics

The UNIX file system uses the following consistency semantics:

• Writes to an open file by a user are visible immediately to other users that have this file open.

• One mode of sharing allows users to share the pointer of current location into the file. Thus, the advancing of the pointer by one user affects all sharing users. Here, a file has a single image that interleaves all accesses, Regardless of their origin.

Session Semantics

The Andrew file system (AFS) uses the following consistency semantics:

• Writes to an open file by a user are not visible immediately to other users that have the same file open.

• Once a file is closed, the changes made to it are visible only in sessions starting later. Already open instances of the file do not reflect these changes.

According to these semantics, a file may be associated temporarily with several (possibly different) images at the same time. Consequently, multiple users are allowed to perform both read and write accesses concurrently on their images of the file, without delay. Almost no constraints are enforced on scheduling accesses.

Immutable-Shared-Files Semantics

A unique approach is that of **immutable shared files.** Once a file is declared as *shared* by its creator, it cannot be modified. An immutable file has two key properties: Its name may not be reused, and its contents may not be altered. Thus, the name of an immutable file signifies that the contents of the file are fixed. The implementation of these semantics in a distributed system is simple.

6. Protection

When information is stored in a computer system, we want to keep it safe from physical damage *(reliability)* and improper access *(protection).*

Reliability is generally provided by duplicate copies of files. Protection can be provided in many ways in a multiuser system.

Types of Access

Protection provide by the types of file access that can be made. Access is permitted or denied depending on type of access requested. Several different types of operations may be controlled:

• Read. Read from the file.

• Write. Write or rewrite the file.

• Execute. Load the file into memory and execute it.

• Append. Write new information at the end of the file.

• Delete. Delete the file and tree its space for possible reuse.

• List. List the name and attributes of the file.

Access Control

The most common approach to the protection problem is to make access dependent on the identity of the user. Different users may need different types of access to a. file or directory. The most general scheme to associate with each file and directory an access-control list (ACL) specifying user names and the types of access allowed for each user. When a user requests access to a particular file, the operating system checks the access list associated with that file. If that user is listed for the requested access, the access is allowed. Otherwise, a protection violation occurs, and the user job is denied access to the file.

The main problem with access lists length. If we want to allow everyone to read a file, we must list all users with read access. This problems can be resolved by use of a condensed version of the access list.

To condense the length of the access-control list, many systems recognize three classifications of users in connection with each file:

• Owner. The user who created the file is the owner.

• Group. A set of users who are sharing the file and need similar access is a group, or work group.

• Universe. All other users in the system constitute the universe.

The most common recent approach is to combine access-control lists with the more general (and easier to implement) owner, group, and universe access control scheme just described.

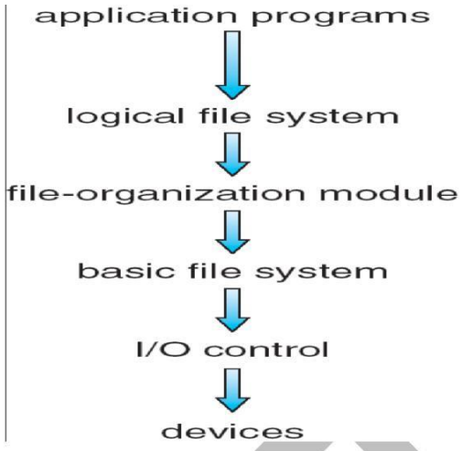
Other Protection Approaches

Another approach to the protection problem is to associate a password with each file. If the passwords are chosen randomly and changed often, this scheme may be effective in limiting access to a file. The use of passwords has a few disadvantages, however. First, the number of passwords that a user needs to remember may become large, making the scheme impractical. Second, if only one password is used for all the files, then once it is discovered, all files are accessible; protection is on an all-or-none basis. Some systems allow a user to associate a password with a subdirectory, rather than with an individual file, to deal with this problem. The IBMVM/CMS operating system allows three passwords for a minidisk—one each for read, write, and multi write access.

FILE SYSTEM IMPLEMENTATION

1. File-System Structure

File are stored on disk. To provide efficient access to the disk, the operating system imposes file systems to allow the data to be stored, and retrieved easily. The file system is generally composed of many different levels. The structure shown in Figure. Each level in the design uses the features of lower levels to create new features for use by higher levels.



Layered file system.

The lowest level, the *I/O control*, consists of **device drivers** to transfer information between the main memory and the disk system. A device driver can be thought of as a translator. Its input consists of high-level commands such as "retrieve block 123." Its output consists of lowlevel, hardware-specific instructions that are used by the hardware controller,

The **basic file system** needs only to issue generic commands to the appropriate device driver to read and write physical blocks on the disk. Each physical block is identified by its numeric disk address (for example, drive 1, cylinder 73, track 2, and sector 10).

The file-organization module knows about files and their logical blocks, as well as physical blocks. By knowing the type of file allocation used and the location of the file, the file-organization module can translate logical block addresses to physical block addresses.

Finally, the logical file system manages metadata information. Metadata includes all of the file-system structure except the actual *data* (or contents of the files). It maintains file structure via file-control blocks. A **file-control block** (FCB) contains information about the file, including ownership, permissions,
and location of the file contents.

2. File-System implementation

In this section, we discuss the structures and operations used to implement file system operations.
Several on-disk and in-memory structures are used to implement a file system.

On disk, structures are

• **A boot control block** (per volume) can contain information needed by the system to boot an operating system from that volume. If the disk does not contain an operating system, this block can be empty. It is called as boot block;

• A **volume control block** (per volume) contains volume (or partition) details, such as the number of blocks in the partition, size of the blocks, free block count and free-block pointers, and free FCB count and FCB pointers. this is called as **superblock.**

• A **directory structure** per file system is used to organize the files. This includes file names and associated **inode** numbers.

• A **per-file FCB** contains many details about the file, including file permissions, ownership, size, and location of the data blocks.

The in-memory structures may include the ones described below:

• An in-memory **mount table** contains information about each mounted volume.

• An in-memory **directory-structure cache** holds the directory information of recently accessed directories.

• The **system-wide open-file table** contains a copy of the FCB of each open file, as well as other information.

• The **per-process open-file table** contains a pointer to the appropriate entry in the system-wide open-file table, as well as other information.

To create a new file, an application program calls the logical file system. The logical file system knows the format of the directory structures. To create a new file, it allocates a new FCB. The system then reads the appropriate directory into memory, updates it with the new file name and FCB, and writes it back to the disk.

3. Directory implementation

The selection of directory-allocation and directory-management algorithms significantly affects the efficiency, performance, and reliability of the file system. In this section, we discuss various algorithms.

3.1 Linear List

The simplest method of implementing a directory is to use a linear list of file names with pointers to the data blocks. This method is simple to program but time-consuming to execute. To create a new file., we must first search the directory to be sure that no existing file has the same name. Then, we add a new entry at the end of the directory. To delete a file, we search the directory for the named file, then release the space allocated to it. The real disadvantage of a linear list of directory entries is that finding a file requires a linear search.

3.2 Hash Table

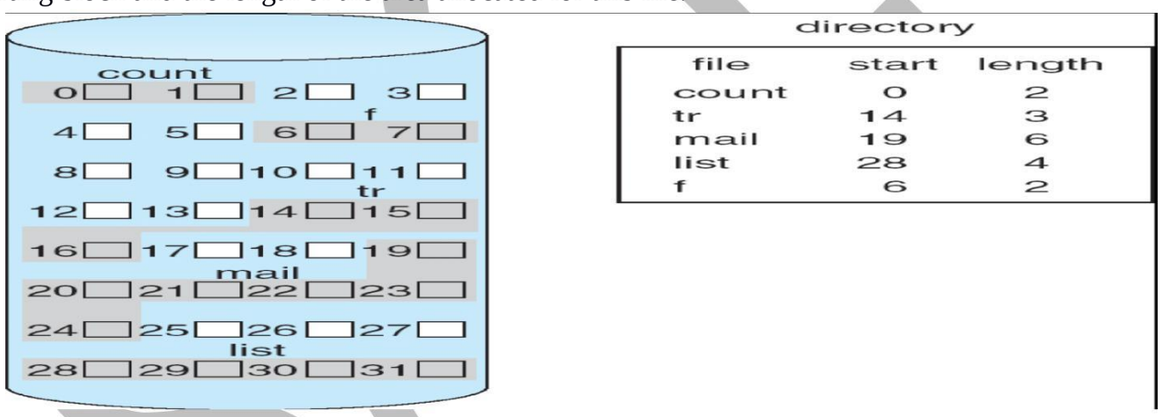
Another data structure used for a directory is a **hash table.** With this method, a hash data structure is used. The hash table takes a value computed from the file name and returns a pointer to the file name in the linear list. Therefore, it can greatly decrease the directory search time. Insertion and deletion are also fairly straightforward, although some provision must be made for **collisions**—situations in which two file names hash to the same location. a chained-overflow hash table can be used. Each hash entry can be a linked list instead of an individual value, and we can resolve collisions by adding the new entry to the linked list.

4. Allocation Methods

In this section we mainly discuss about how to allocate space to the files. Three major methods of allocating disk space are in use: contiguous, linked, and indexed.

4.1 Contiguous Allocation

Contiguous allocation requires that each file occupy a set of contiguous blocks on the disk. Disk addresses define a linear ordering on the disk. With this ordering, accessing block *b* + 1 after block *b* normally requires no head movement. When head movement is needed, the head need only move from one track to the next. Thus, the number of disk seeks required for accessing contiguously allocated files is minimal. The IBM VM/CMS operating system uses contiguous allocation. The directory entry for each file indicates the address of the starting block and the length of the area allocated for this file.



Contagious Allocation of Disk Space

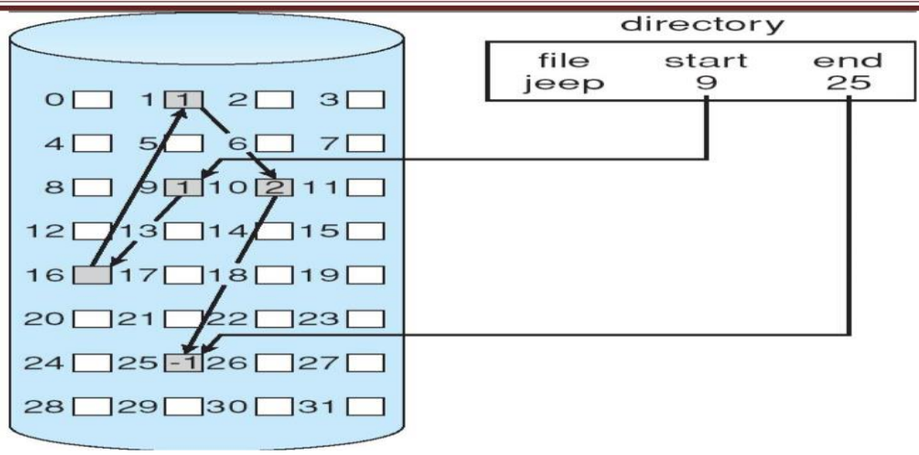
Accessing a file that has been allocated contiguously is easy. For sequential access, the file system remembers the disk address of the last block referenced and, reads the next block. For direct access to block i of a file that starts at block *b*, we can immediately access block *b* + *i*. Thus, both sequential and direct access can be supported by contiguous allocation.

Contiguous allocation has some problems. One difficulty is finding space for a new file. Second suffer from the problem of **external fragmentation.** Another problem is determining how much space is needed for a file, When the file is created.

To minimize these drawbacks, some operating systems use a modified contiguous-allocation scheme. Here, a contiguous chunk of space is allocated initially; and then, if that amount proves not to be large enough, another chunk of contiguous space, known as an **extent,** is added. The location of a file's blocks is then recorded as a location and a block count, plus a link to the first block of the next extent.

4.2 Linked Allocation

Linked allocation solves all problems of contiguous allocation. With linked allocation, each file is a linked list of disk blocks; the disk blocks may be scattered anywhere on the disk. The directory contains a pointer to the first and last blocks of the file. There is no external fragmentation with linked allocation, and any free block on the free-space list can be used to satisfy a request. The size of a file need not be declared when that file is created.



Linked Allocation

The major problem is that it can be used effectively only for sequential-access files. To find the i^{th} block of a file, we must start at the beginning of that file and follow the pointers until we get to the i^{th} block .

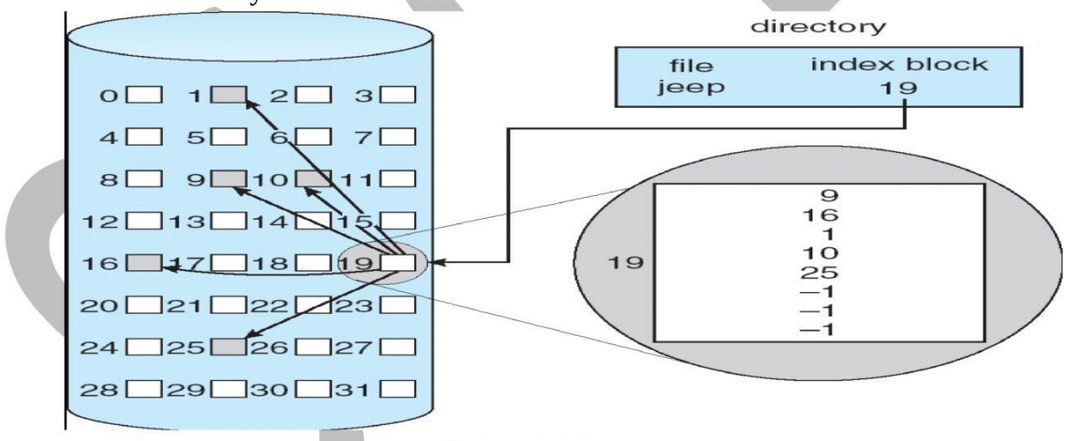
Another disadvantage is the space required for the pointers. If a pointer requires 4 bytes out of a 512-byte block, then 0.78 percent of the disk is being used for pointers, rather than for information.

The usual solution to this problem is to collect blocks into multiples, called clusters, and to allocate clusters rather than blocks. This approach is increases internal fragmentation, because more space is wasted when a cluster is partially full than when a block is partially full.

Another problem of linked allocation is reliability, because files are linked together by pointers scattered all over the disk, and pointers are lost or damaged.

4.3 Indexed Allocation

Linked allocation solves the external-fragmentation and size-declaration problems of contiguous allocation. However, linked allocation cannot support efficient direct access, since the pointers to the blocks are scattered with the blocks themselves all over the disk and must be retrieved in order. **Indexed allocation** solves this problem by bringing all the pointers together into one location: the **index block.** Each file has its own index block, which is an array of disk-block addresses. The I^{th} entry in the index block points to the I^{th} block of the file. The directory contains the address of the index block.



Indexed Allocation

Indexed allocation supports direct access, without suffering from external fragmentation. The pointer overhead of the index block is generally greater than the pointer overhead of linked allocation. If the index block is too small, it will not be able to hold enough pointers for a large file. Mechanisms for this purpose include the following:

* **Linked scheme**. An index block is normally one disk block. Thus, it can be read and written directly by itself. To allow for large files, we can link together several index blocks.

• **Multilevel index**. A variant of the linked representation is to use a first level index block to point to a set of second-level index blocks, which in turn point to the file blocks. To access a block, the operating system uses the first-level index to find a second-level index block and then uses that block to find the desired data block.

• **Combined scheme.** It keeps the first, 15 pointers of the index block in the file's inode. The first 12 of these pointers point to **direct blocks;** that is, they contain addresses of blocks that contain data of the file. The next three pointers point to **indirect blocks.** The first points to a **single indirect block,** which is an index block containing not data but the addresses of blocks that contain data. The second points to a **double indirect block,** which contains the address of a block that contains the addresses of blocks that contain pointers to the actual data blocks. The last pointer contains the address of a **triple indirect block.**

5. Free-Space Management

Since disk space is limited, we need to reuse the space from deleted files for new Files. So, To keep track of free disk space, the system maintains a **free-space list.** The free-space list records *all free* disk blocks.To create a file, we search the free-space list for the required amount of space and allocate that space to the new file. This space is then removed from the free-space list. When a file is deleted, its disk space is added to the free-space list. The free-space list, be implemented as we discussed below

5.1 Bit Vector

In this free-space list is implemented as a bit **map** or bit vector. Each block is represented by 1 bit. If the block is free, the bit is 1; if the block is allocated, the bit is 0. For example, consider a disk where blocks 2, 3, 4, 5, 8, 9, 10, 11, 12, 13, 17,18, 25,26, and 27 are free and the rest of the blocks are allocated. The free-space bit map would be 001111001111110001100000011100000 ...

The main advantage of this approach is its relative simplicity and its efficiency in finding the first free block or *n* consecutive free blocks on the disk

Unfortunately, bit vectors are inefficient because the entire vector is kept in main memory, is not possible for larger disks.

5.2 Linked List

Another approach to free-space management is to link together all the free disk blocks, keeping a pointer to the first free block in memory. This first block contains a pointer to the next free disk block, and so on. However; this scheme is not efficient; to traverse the list, we must read each block, which requires substantial I/O time.

5.3 Grouping

A modification of the free-list approach is to store the addresses of *n* free blocks in the first free block. The first n—1 of these blocks are actually free. The last block contains the addresses of other n free blocks, and so on. The addresses of a large number of free blocks can now be found quickly.

5.4 Counting

We can keep the address of the first free block and the number *n* of free contiguous blocks that follow the first block. Each entry in the free-space list then consists of a disk address and a count. Although each entry requires more space than, but the overall list will be shorter, as long as the count is generally greater than 1.

Mass-Storage Structure

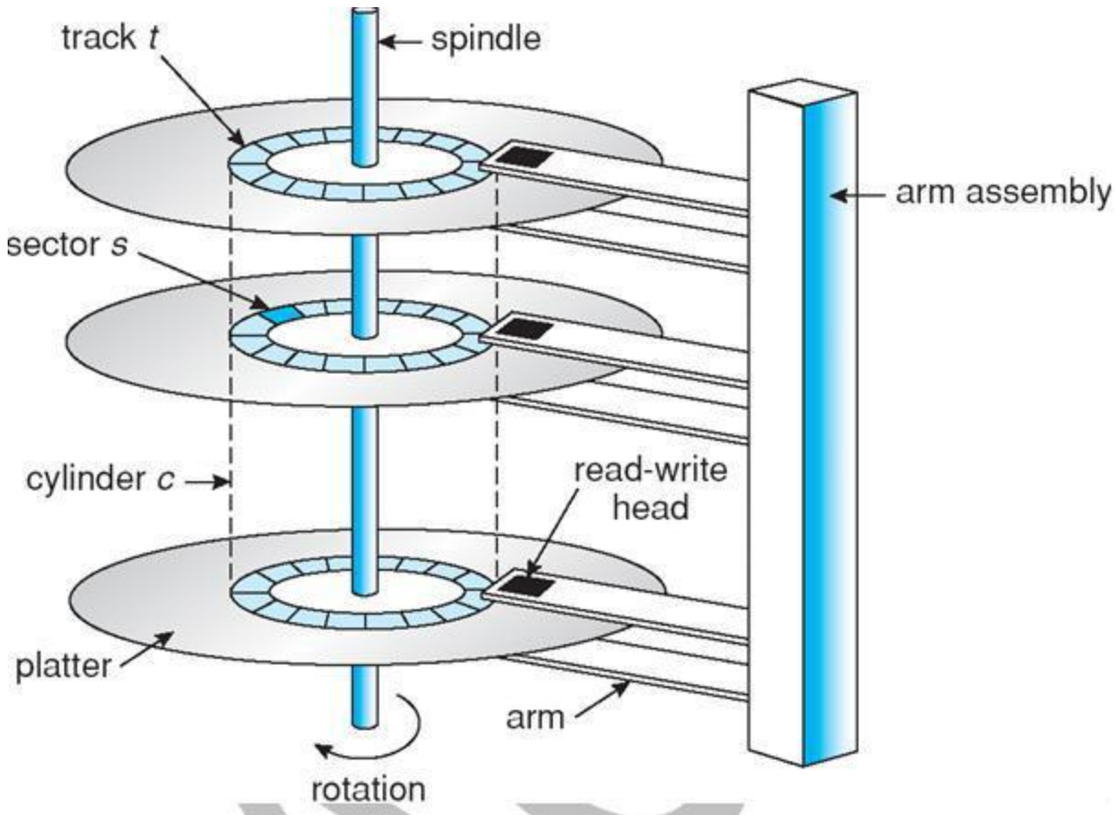
1. Overview of Mass-Storage Structure

In this section we present a general overview of the physical structure of secondary and tertiary storage devices.

Magnetic Disks

Magnetic disks provide the bulk of secondary storage for modern computer systems. Each disk

platter has a flat circular shape, like a CD. Common platter diameters range from 1.8 to 5.25 inches. The two surfaces of a platter are covered with a magnetic material. We store information by recording it magnetically on the platters.



Moving-head Disk Mechanism

A read-write head "flies" just above each surface of every platter. The heads are attached to a **disk arm** that moves all the heads as a unit. The surface of a platter is logically divided into circular **tracks,** which are subdivided into **sectors.** The set of tracks that are at one arm position makes up a **cylinder.**

When the disk is in use, a drive motor spins it at high speed. Most drives rotate 60 to 200 times per second. Disk speed has two parts. The **transfer rate** is the rate at which data flow between the drive and the computer. The **positioning time,** sometimes called the **random-access time,** consists of the time to move the disk arm to the desired cylinder, called the **seek time,** and the time for the desired sector to rotate to the disk head, called the **rotational latency.** Typical disks can transfer several megabytes of data per second, and they have seek times and rotational latencies of several milliseconds.

Because the disk head flies on an extremely thin cushion of air , there is a danger that the head will make contact with the disk surface. Although the disk platters are coated with a thin protective layer, sometimes the head will damage the magnetic surface. This accident is called a **head crash.** A head crash normally cannot be repaired; the entire disk must be replaced.

A disk can be **removable,** allowing different disks to be mounted as needed. Removable magnetic disks generally consist of one platter, held in a plastic case to prevent damage while not in the disk drive. **Floppy disks** are inexpensive removable magnetic disks that have a soft plastic case containing a flexible platter.

A disk drive is attached to a computer by a set of wires called an I/O **bus.** Several kinds of buses are available, including **enhanced integrated drive electronics (EIDE), advanced technology attachment (ATA), serial ATA** (SATA), **universal serial bus (USB), fiber channel (FC),** and **SCSI** buses.

The data transfers on a bus are carried out by special electronic processors called **controllers.** The **host controller** is the controller at the computer end of the bus. A **disk controller** is built into each disk drive. To perform a disk I/O operation, the computer places a command into the host controller. The host-

controller then sends the command via messages to the disk controller, and the disk controller operates the disk-drive hardware to carry out the command.

Magnetic Tapes

Magnetic tape was used as an early secondary-storage medium. Although it is relatively permanent and can hold large quantities of data, its access time is slow compared with that of main memory and magnetic disk. In addition, random access to magnetic tape is about a thousand times slower than random access to magnetic disk, so tapes are not very useful for secondary storage.

Tapes are used mainly for backup, for storage of infrequently used information, and as a medium for transferring information from one system to another. A tape is kept in a spool and is wound or rewound past a read-write head.

Moving to the correct spot on a tape can take minutes, but once positioned, tape drives can write data at speeds comparable to disk drives. Tape capacities vary greatly, they store from 20 GB to 200 GB.

2. Disk Structure

Modern disk drives are addressed as large one-dimensional arrays of **logical blocks,** where the logical block is the smallest unit of transfer. The size of a logical block is usually 512 bytes. The one-dimensional array of logical blocks is mapped onto the sectors of the disk sequentially. Sector 0 is the first sector of the first track on the outermost cylinder. The mapping proceeds in order through that track, then through the rest of the tracks in that cylinder, and then through the rest of the cylinders from outermost to innermost.

By using this mapping, we can convert a logical block number into an old-style disk address that consists of a cylinder number, a track number within that cylinder, and a sector number within that track. In practice, it is difficult to perform, this translation, because the number of sectors per track is not a constant on some drives.

On media that use **constant linear velocity** (CLV), the density of bits per track is uniform. The farther a track is from the center of the disk, the greater its length, so the more sectors it can hold. As we move from outer zones to inner zones, the number of sectors per track decreases. Tracks in the outermost zone typically hold 40 percent more sectors than do tracks in the innermost zone. The drive increases its rotation speed as the head moves from the outer to the inner tracks to keep the same rate of data moving under the head. This method is used in CD-ROM and DVD-ROM drives.

Alternatively, the disk rotation speed can stay constant, and the density of bits decreases from inner tracks to outer tracks to keep the data rate constant. This method is used in hard disks and is known as **constant** angular **velocity (CAV).**

3. Disk Attachment

Computers access disk storage in two ways. One way is via I/O ports (or host-attached storage); this is common on small systems. The other way is via a remote host in a distributed file system; this is referred to as network-attached storage.

Host-Attached Storage

Host-attached storage is storage accessed through local I/O ports. These ports use several technologies.

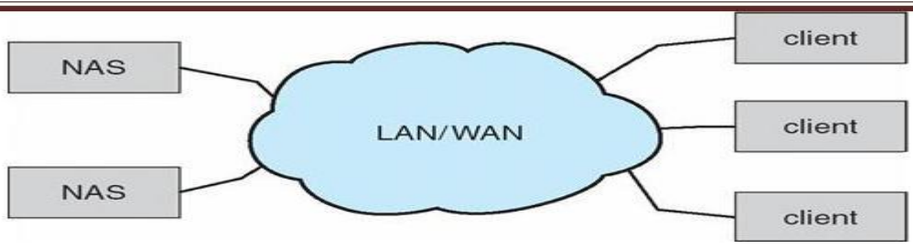
The typical desktop PC uses an I/O bus architecture called IDE or ATA. This architecture supports a maximum of two drives per I/O bus. A newer, similar protocol that has simplified cabling is SATA. High-end workstations and servers generally use more sophisticated I/O architectures, such as SCSI and fiber channel (FC).

SCSI is a bus architecture. Its physical medium is usually a ribbon cable. The SCSI protocol supports a maximum of 16 devices on the bus. Generally, the devices include one controller card in the host (the **SCSI initiator)** and up to 15 storage devices (the **SCSI targets).**

FC is a high-speed serial architecture that can operate over optical fiber or over a four-conductor copper cable. It is the basis of **storage-area networks** (SANs),

Network-Attached Storage

A network-attached storage (NAS) device is a special-purpose storage system that is accessed remotely over a data network. Clients access network-attached storage via a remote-procedure-call interface. The remote procedure calls (RPCs) are carried via TCP or UDP over an IP network—-usually the same local-area network (LAN) that carries all data traffic to the clients.



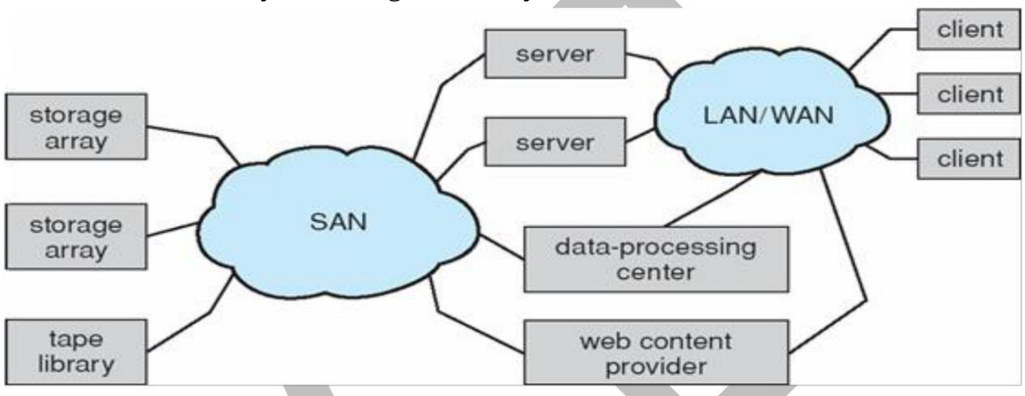
Network-Attached Storage

Network-attached storage provides a convenient way for all the computers on a LAN to share a pool of storage. However, it tends to be less efficient and have lower performance than some direct-attached storage options.

ISCSI is the latest network-attached storage protocol. In essence, it uses the IP network protocol to carry the SCSI protocol. Thus, networks rather than SCSI cables can be used as the interconnects between hosts and their storage. As a result, hosts can treat their storage as if it were directly attached, but the storage can be distant from the host.

Storage-Area Network

One drawback of network-attached storage systems is that the storage I/O operations consume bandwidth on the data network, thereby increasing the latency of network communication. It overcome by SAN.



Storage Area Network

A storage-area network (SAN) is a private network (using storage protocols rather than networking protocols) connecting servers and storage units, as shown in Figure. The power of a SAN lies in its flexibility. Multiple hosts and multiple storage arrays can attach to the same SAN. A SAN switch allows or prohibits access between the hosts and the storage.

An emerging alternative is a special-purpose bus architecture named InfiniBand, which provides hardware and software support for high-speed interconnection networks for servers and storage units.

4. Disk Scheduling

To use the hardware efficiently, the disk drives have fast access time and large disk bandwidth. The access time has two major components. The **seek time** is the time for the disk arm to move the heads to the cylinder containing the desired sector. The **rotational latency** is the additional time for the disk to rotate the desired sector to the disk head. The disk **bandwidth** is the total number of bytes transferred, divided by the total time between the first request for service and the completion of the last transfer. We can improve both the access time and the bandwidth by scheduling the servicing of disk I/O requests in a good order.

Whenever a process needs I/O to or from the disk, it issues a system call to the operating system. The request specifies several pieces of information:

3. Whether this operation is input or output
4. What the disk address for the transfer is
5. What the memory address for the transfer is
6. What the number of sectors to be transferred is

If the desired disk drive and controller are available, the request can be serviced immediately. If the drive or controller is busy, any new requests for service will be placed in the queue of pending requests for that drive.

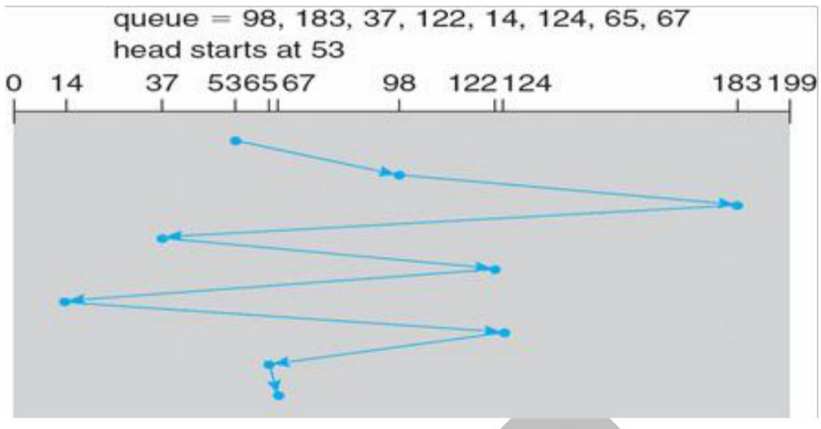
For a multiprogramming system with many processes, the disk queue may often have several pending requests. Thus, when one request is completed,, the operating system chooses which pending request to service next. Several disk-scheduling algorithms can be used, and we discuss them next.

FCFS Scheduling

The simplest form of disk scheduling is the first-come, first-served (FCFS) algorithm. This algorithm is intrinsically fair, but it generally does not provide the fastest service. Consider, for example, a disk queue with requests for I/O to blocks on cylinders

98, 183, 37,122, 14, 124, 65, 67,

in that order. If the disk head is initially at cylinder 53, it will first move from 53 to 98, then to 183, 37, 122, 14, 124/65, and finally to 67, for a total head movement of 640 cylinders.

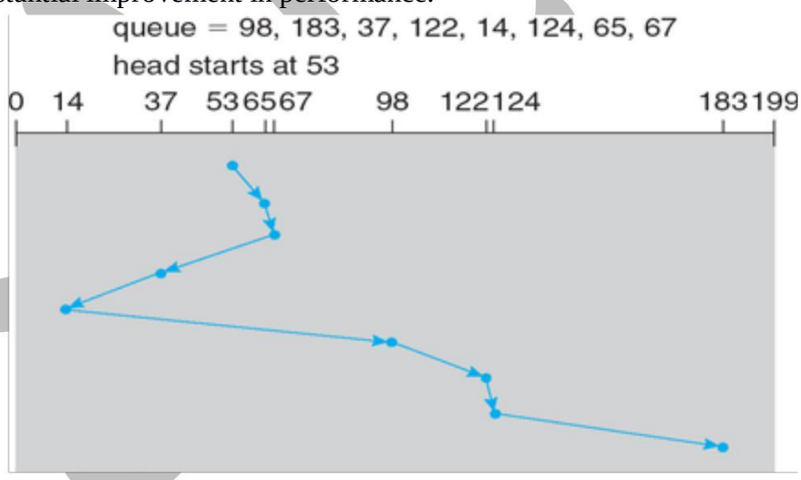


The wild swing from 122 to 14 and then back to 124 illustrates the problem with this schedule. If the requests for cylinders 37 and 14 could be serviced together, before or after the requests at 122 and 124, the total head movement could be decreased substantially, and performance could be thereby improved.

SSTF Scheduling

The shortest-seek-time-first (SSTF) algorithm selects the request with the minimum seek time from the current head position. Since seek time increases with the number of cylinders traversed by the head, SSTF chooses the pending request closest to the current head position.

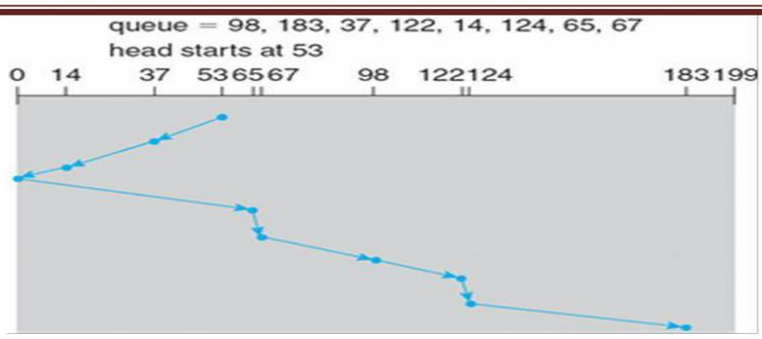
For our example request queue, the closest request to the initial head position (53) is at cylinder 65. Once we are at cylinder 65, the next closest request is at cylinder 67. From there, the request at cylinder 37 is closer than the one at 98, so 37 is served next. Continuing, we service the request at cylinder 14, then 98,122, 124, and finally 183. This scheduling method results in a total head movement of only 236 cylinders. This algorithm gives a substantial improvement in performance.



SSTF scheduling is essentially a form of shortest-job-first (SJF) scheduling; and like SJF scheduling, it may cause starvation of some requests. Remember that requests may arrive at any time. Suppose that we have two requests in the queue, for cylinders 14 and 186, and while servicing the request from 14, a new request near 14 arrives. This new request will be serviced next, making the request at 186 wait. While this request is being serviced, another request close to 14 could arrive. In theory, a continual stream of requests near one another could arrive, causing the request for cylinder 186 to wait indefinitely.

SCAN Scheduling

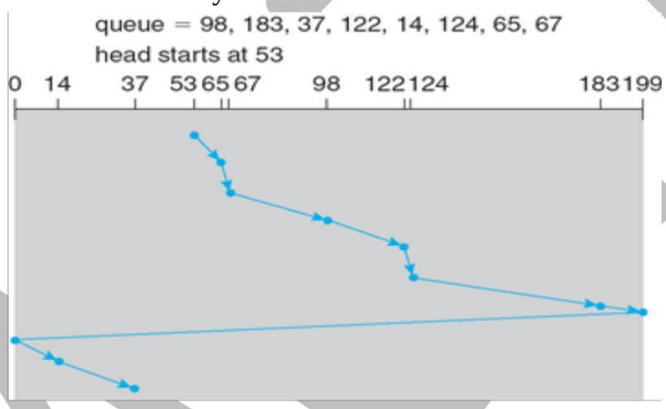
In the SCAN algorithm, the disk arm starts at one end of the disk and moves toward the other end, servicing requests as it reaches each cylinder, until it gets to the other end of the disk. At the other end, the direction of head movement is reversed, and servicing continues. The head continuously scans back and forth across the disk. The SCAN algorithm is sometimes called the elevator algorithm, since the disk arm behaves just like an elevator in a building, first servicing all the requests going up and then reversing to service requests the other way.



Let's return to our example to illustrate. Before applying SCAN to schedule the requests on cylinders 98,183, 37,122,14, 124, 65, and 67, we need to know the direction of head movement in addition to the head's current position (53). If the disk arm is moving toward 0, the head will service 37 and then 14. At cylinder 0, the arm will reverse and will move toward the other end of the disk, servicing the requests at 65, 67, 98, 122, 124, and 183 (Figure 12.6). If a request arrives in the queue just in front of the head, it will be serviced almost immediately; a request arriving just behind the head will have to wait until the arm moves to the end of the disk, reverses direction, and comes back.

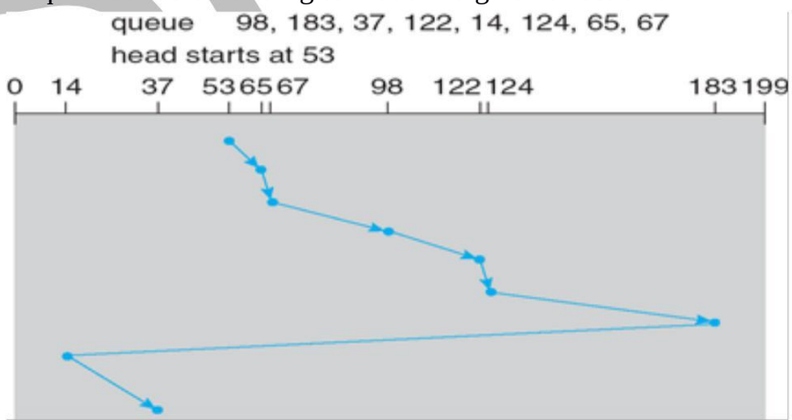
C-SCAN Scheduling

Circular SCAN (C-SCAN) **scheduling** is a variant of SCAN designed to provide a more uniform wait time. Like SCAN, C-SCAN moves the head from one end of the disk to the other, servicing requests along the way. When the head reaches the other end, however, it immediately returns to the beginning of the disk, without servicing any requests on the return trip . The C-SCAN scheduling algorithm essentially treats the cylinders as a circular list that wraps around from the final cylinder to the first one.



LOOK Scheduling

As we described, both SCAN and C-SCAK move the disk arm across the full width of the disk. In practice, neither algorithm is often implemented this way. More commonly, the arm goes only as far as the final request in each direction. Then, it reverses direction immediately, without going all the way to the end of the disk. Versions of SCAN and C-SCAN that follow this pattern are called **LOOK** and **C-LOOK scheduling,** because they *look* for a request before continuing to move in a given direction .



Selection of a Disk-Scheduling Algorithm

SSTF is common and has a natural appeal because it increases performance over FCFS. SCAM and C-SCAN perform better for systems that place a heavy load on the disk, because they are less likely to cause a starvation problem. With any scheduling algorithm, however, performance depends heavily on the number and types of requests. For instance, suppose that the queue usually has just one outstanding request. Then, allscheduling algorithms behave the same, because they have only one choice for where to move the disk head: They all behave like FCFS scheduling.

Requests for disk service can be greatly influenced by the file-allocation method. A program reading a contiguously allocated file will generate several requests that are close together on the disk, resulting in limited head movement. A linked or indexed file, in contrast, may include blocks that are widely scattered on the disk, resulting in greater head movement.

6. Swap-Space Management

Swap-space management is another low-level task of the operating system. Virtual memory uses disk space as an extension of main memory. Since disk access is much slower than memory access, using swap space significantly decreases system performance. The main goal for the design, and implementation of swap space is to provide the best throughput for the virtual memory system. In this section, we discuss how swap space is used, where swap space is located on disk, and how swap space is managed.

Swap-Space Use

Swap space is used in various ways by different operating systems, depending on the memory-management algorithms in use. For instance, systems that implement swapping may use swap space to hold an entire process image, including the code and data segments. Paging systems may simply store pages that have been pushed out of main memory.

The amount of swap space needed on a system can therefore vary depending on the amount of physical memory, the amount of virtual memory it is backing, and the way in which the virtual memory is used. It can range from a few megabytes of disk space to gigabytes.Note that it may be safer to overestimate than to underestimate the amount of swap space required, because if a system runs out of swap space it may be forced to abort processes or may crash entirely. Overestimation wastes disk space that could otherwise be used for files, but it does no other harm.

Solaris, for example, suggests setting swap space equal to the amount of physical memory. Historically, Linux suggests setting swap space to double the amount of physical memory.

Swap-Space Location

A swap space can reside in one of two places: It can be carved out of the normal file system, or it can be in a separate disk partition. If the swap space is simply a large file within the file system, normal file-system routines can be used to create it, name it, and allocate its space. This approach, though easy to implement, is inefficient. Navigating the directory structure and the disk-allocation data structures takes time and (potentially) extra disk accesses.

Alternatively, swap space can be created in a separate raw partition, as no file system or directory structure is placed in this space. Rather, a separate swap-space storage manager is used to allocate and deallocate the blocks from the raw partition. This manager uses algorithms optimized for speed rather than for storage efficiency, because swap space is accessed much more frequently than file systems.

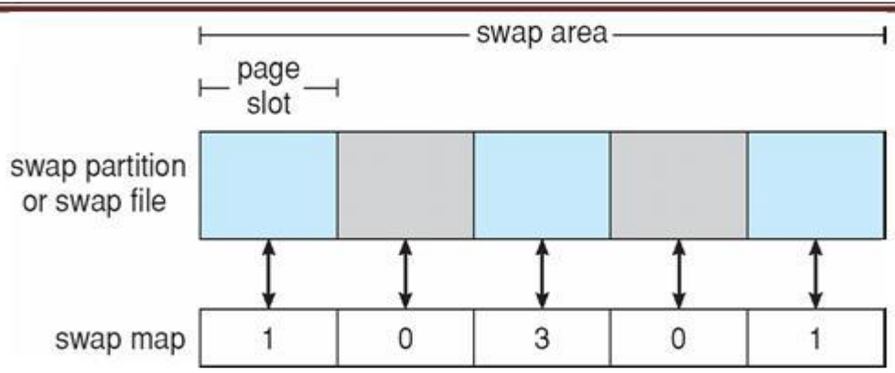
Some operating systems are flexible and can swap both in raw partitions and in file-system space. Linux is an example: The trade-off is between the convenience of allocation and management in the file system and the performance of swapping in raw partitions.

Swap-Space Management: An Example

We can illustrate how swap space is used by following the evolution of swapping and paging in various UNIX systems. The traditional UNIX kernel started with an implementation of swapping that copied entire processes between contiguous disk regions and memory. UNIX later evolved to a combination of swapping and paging as paging hardware became available.

In Solaris 1 (SunOS), the designers changed standard UNIX to improve efficiency and reflect technological changes. When a process executes, text-segment pages containing code are brought in from the file system, accessed in main memory, and thrown away if selected for page out. It is more efficient to reread a page from the file system than to write it to swap space and then reread it from there.

More changes were made in later versions of Solaris. The biggest change is that Solaris now allocates swap space only when a page is forced out of physical memory, rather than when the virtual memory page is first created. This scheme gives better performance on modern computers, which have more physical memory than older systems and tend to page less.



The data structures for swapping on Linux systems

Linux is similar to Solaris in that swap space is only used for regions of memory shared by several processes. Linux allows one or more swap areas to be established. A swap area may be in either a swap file on a regular file system or a raw swap partition. Each swap area consists of a series of 4-KB page slots, which are used to hold swapped pages. Associated with each swap area is a swap map—an array of integer counters, each corresponding to a page slot in the swap area. If the value of a counter is 0, the corresponding page slot is available. Values greater than 0 indicate that the page slot is occupied by a swapped page. The value of the counter indicates the number of mappings to the swapped page; for example, a value of 3 indicates that the swapped page is mapped to three different processes.

7. Disk Management

The operating system is responsible for several other aspects of disk management. Here we discuss disk initialization, booting from disk, and bad-block recovery.

Disk Formatting

A new magnetic disk is a platter of a magnetic recording material. Before a disk can store data, it must be divided into sectors that the disk controller can read and write. This process is called low-level formatting, or physical formatting. Low-level formatting fills the disk with a special data structure for each sector. The data structure for a sector typically consists of a header, a data area (usually 512 bytes in size), and a trailer. The header and trailer contain information used by the disk controller, such as a sector number and an error-correcting code (ECC). When the controller writes a sector of data during normal I/O, the ECC is updated with a value calculated from all the bytes in the data area. When the sector is read, the ECC is recalculated and is compared with the stored value. If the stored and calculated numbers are different, this mismatch indicates that the data area of the sector has become corrupted and that the disk sector may be bad Most hard disks are low-level-formatted at the factory as a part of the manufacturing process. This formatting enables the manufacturer to test the disk and to initialize the mapping from logical block numbers to defect-free sectors on the disk. For many hard disks, when the disk controller is instructed to low-level-format the disk, it can also be told how many bytes of data space to leave between the header and trailer of all sectors. It is usually possible to choose among a few sizes, such as 256, 512, and 1,024 bytes. Formatting a disk with a larger sector size means that fewer sectors can fit on each track; but it also means that fewer headers and trailers are written on each track and more space is available for user data.

To use a disk to hold files, the operating system still needs to record its own data structures on the disk. It does so in two steps. The first step is to **partition** the disk into one or more groups of cylinders. The operating system can treat each partition as though it were a separate disk. For instance, one partition can hold a copy of the operating system's executable code, while another holds user files. After partitioning, the second step is **logical formatting**. In this step, the operating system stores the initial file-system data structures onto the disk. These data structures may include maps of free and allocated space (a FAT or modes) and an initial empty directory.

Boot Block

When a computer is powered up or rebooted—it must have an initial program to run. This initial *bootstrap* program tends to be simple. It initializes all aspects of the system, from CPU registers to device controllers and the contents of main memory, and then starts the operating system. To do its job, the bootstrap program finds the operatingsystem kernel on disk, loads that kernel into memory, and jumps to an initial address to begin the operating-system execution.

For most computers, the bootstrap is stored in read-only memory (ROM). This location is convenient, because ROM is at a fixed location that the processor can start executing when powered up or reset. And, since ROM is read only, it cannot be infected by a computer virus. The problem is that changing this bootstrap code requires changing the ROM, hardware chips.For this reason, most systems **store a tiny bootstrap loader**

program in the boot ROM whose only job is to bring in a full bootstrap program from disk. The full bootstrap program can be changed easily: A new version is simply written onto the disk. The full bootstrap program is stored in "the boot blocks" at a fixed location on the disk. A disk that has a boot partition is called a boot disk or system disk.

Bad Blocks

Because disks have moving parts and small tolerances, they are prone to failure. Sometimes the failure is complete; in this case, the disk needs to be replaced and its contents restored from backup media to the new disk. More frequently, one or more sectors become defective. Most disks even come from the factory with bad blocks. Depending on the disk and controller in use, these blocks are handled in a variety of ways.

On simple disks, such as some disks with IDE controllers, bad blocks are handled manually. For instance, the MS-DOS format command performs logical formatting and, as a part of the process, scans the disk to find bad blocks. If format finds a bad block, it writes a special value into the corresponding FAT entry to tell the allocation routines not to use that block. If blocks go bad during normal operation, a special program (such as chkdsk) must be run manually to search for the bad blocks and to lock them away as before. Data that resided on the bad blocks usually are lost.

More sophisticated disks, such as the SCSI disks used in high-end PCs and most workstations and servers, are smarter about bad-block recovery. The controller maintains a list of bad blocks on the disk. The list is initialized during the low-level formatting at the factory and is updated over the life of the disk.

Low-level formatting also sets aside spare sectors not visible to the operating system. The controller can be told to replace each bad sector logically with one of the spare sectors. This scheme is known as **sector sparing** or **forwarding.**

A typical bad-sector transaction might be as follows:

4. The operating system tries to read logical block 87.
5. The controller calculates the ECC and finds that the sector is bad. It reports this finding to the operating system.
6. The next time the system is rebooted, a special, command is run to tell the
7. SCSI controller to replace the bad sector with a spare.
8. After that, whenever the system requests logical block 87, the request is translated into the replacement sector's address by the controller.

Such a redirection by the controller could invalidate any optimization by the operating system's disk-scheduling algorithm! For this reason, most disks are formatted to provide a few spare sectors in each, cylinder and a spare cylinder as well. When a bad block is remapped, the controller uses a spare sector from the same cylinder, if possible.

As an alternative to sector sparing, some controllers can be instructed to replace a bad block by sector slipping. Here is an example: Suppose that logical block 17 becomes defective and the first available spare follows sector 202. Then, sector slipping remaps all the sectors from 17 to 202, moving them all down one spot. That is, sector 202 is copied into the spare, then sector 201 into 202, and then 200 into 201, and so on, until sector 18 is copied into sector 19. Slipping the sectors in this way frees up the space of sector 18, so sector 17 can be mapped to it.

The replacement of a bad block generally is not totally automatic because the data in the bad block are usually lost.

DEADLOCKS

1.System Model

A system consists of a finite number of resources to be distributed among a number of competing processes. A process must request a resource before using it and must release the resource after using it. Under the normal mode of operation, a process may utilize a resource in only the following sequence:

• **Request.** If the request cannot be granted immediately (for example, if the resource is being used by another process), then the requesting process must wait until it can acquire the resource.

• **Use**, The process can operate on the resource.

3**. Release**. The process releases the resource.

A set of processes is in a deadlock state when every process in the set is waiting for an event that can be caused only by another process in the set. The events with which we are mainly concerned here are resource acquisition and release.

To illustrate a deadlock state, consider a system with one printer and one DVD drive. Suppose that process Pi is holding the DVD and process *Pj* is holding the printer. If Pi requests the printer and Pj requests the DVD drive, a deadlock occurs.

A programmer who is developing multithreaded applications must pay attention to this problem.

2. Deadlock Characterization

Before we discuss the various methods for dealing with the deadlock problem, we look more closely at features that characterize deadlocks.

Necessary Conditions

A deadlock situation can arise if the following four conditions hold simultaneously in a system:

• **Mutual exclusion.** At least one resource must be held in a non- sharable mode; that is, only one process at a time can use the resource. If another process requests that resource, the requesting process must be delayed until the resource has been released.

• **Hold and wait.** A process must be holding at least one resource and waiting to acquire additional resources that are currently being held by other processes.

• **No preemption.** Resources cannot be preempted.; that is, a resource can be released only voluntarily by the process holding it, after that process has completed its task.

• **Circular wait**. A set *{P0, Pi, ..., Pn }* of waiting processes must exist such that *P-0* is waiting for a resource held by *P1, P1* is waiting for a resource held by P2, •••, Pn-1 is waiting for a resource held by *Pn,* and *Pn* is waiting for a resource held by P0.

We emphasize that all four conditions must hold for a deadlock to occur.

Resource-Allocation Graph

Deadlocks can be described more precisely in terms of a directed graph called a **system resource-allocation** graph. This graph consists of a set of vertices *V* and a set of edges *E*. The set of vertices *V* is partitioned into two different types of nodes:

P - {Pi, *Pi,,.., Pn,},* the set consisting of all the active processes in the system, and

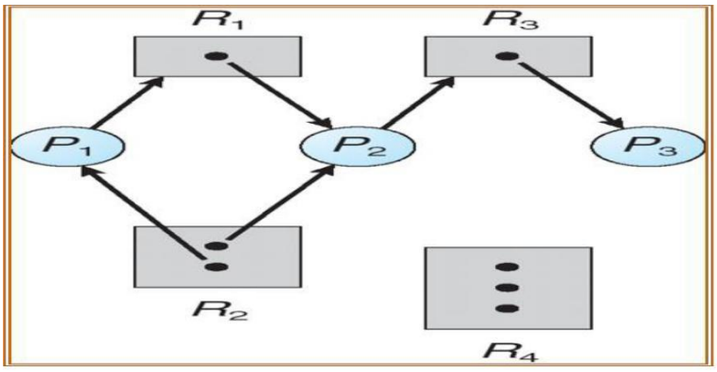
R = {R[, R?, •••/ *Rm},* the set consisting of all resource types in the system.

A directed edge from process *Pi* to resource type *Rj* is denoted by P; -> R ,•; it signifies that process Pi, has requested an instance of resource type Rj, and is currently waiting for that resource. A directed edge P, —> *Rj* is called a **request edge;**

A directed edge from resource type *Rj* to process *Pi* is denoted by *Rj* -»• Pi it signifies that an instance of resource type *Rj* has been allocated to process Pi. a directed edge *Rj* -* P; is called an **assignment edge.**

Pictorially, we represent each process P, as a circle and each resource type *Ri* as a rectangle. Since resource type *Rj* may have more than one instance, we represent each such instance as a dot within the rectangle. Note that a request edge points to only the rectangle *R;,* whereas an assignment edge must also designate one of the dots in the rectangle.

The resource-allocation graph shown in Figure depicts the following situation.



• The sets P, *R,* and

£: *o P={P1,P2,P3}*

o R= {R1, *R2,R3,* R4}

o £ = {p1 ->*R1, P2* -> R3/ R,2->*p1, R2* -> P2/ *R2* _> p.,, *R3* ->P3 }

* Resource instances:

o One instance of resource type R1 *o*

Two instances of resource type R2 '"

One instance of resource type *R3*

r>Three instances of resource type *R4*

• Process states:

o Process *P1* is holding an instance of resource type R2 and is waiting for an instance of resource type R1.

o Process P2 is holding an instance of *R1* and an instance of R2 and is waiting for an instance of *R3.*

o Process P3 is holding an instance of R3.

Given the definition of a resource-allocation graph, it can be shown that, if the graph contains no cycles, then no process in the system is deadlocked. If the graph does contain a cycle, then a deadlock may exist.

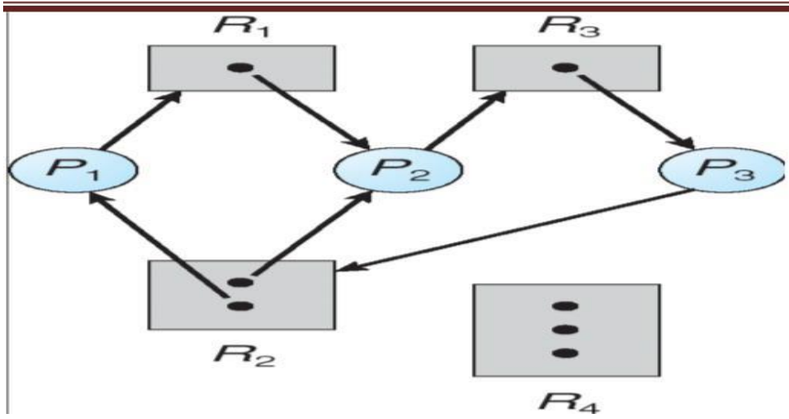
If each resource type has exactly one instance, then a cycle implies that a deadlock has occurred.

a cycle in the graph is both a necessary and a sufficient condition for the existence of deadlock.

If each resource type has several instances, then a cycle does not necessarily imply that a deadlock has occurred. In this case, a cycle in the graph is a necessary but not a sufficient condition for the existence of deadlock.

To illustrate this concept, we return to the resource-allocation graph depicted in Figure. Suppose that process P3 requests an instance of resource type *R2.* Since no resource instanceis currently available, a request edge P3 —>• R2 is added to the graph. At this point, two minimal cycles exist in the svstem:

SACET



Resource-allocation graph with a deadlock.

P1 ---+ *R* 1 ---+ P2 ---+ R3 ---+ *P3* ---+ R2 ---+ P1

P2 ---+ R3 ---+ *P3*---+ R2 ---+ P2

Processes *P1, Pz,* and *P3* are deadlocked. Process *Pz* is waiting for the resource *R3,* which is held by process *P3.* Process *P3* is waiting for either process *P1* or process *Pz* to release resource R2. In addition, process *P1* is waiting for process *P2* to release resource R1.

Methods for Handling Deadlocks

Generally speaking, we can deal with the deadlock problem in one of three ways:

7. We can use a protocol to prevent or avoid deadlocks, ensuring that the system will *never* enter a deadlock state.
8. We can allow the system to enter a deadlock state, detect it, and recover.
9. We can ignore the problem altogether and pretend that deadlocks never occur in the system.

The third solution is the one used by most operating systems, including LINUX and Windows.

3. Deadlock Prevention

As we noted ,for a deadlock to occur, each of the four necessary conditions must hold. By ensuring that at least one of these conditions cannot hold, we can *prevent* the occurrence of a deadlock.

Mutual Exclusion

The mutual-exclusion condition must hold for nonsharable resources. A process never needs to wait for a sharable resource.So, we cannot prevent deadlocks by denying the mutual-exclusion condition, because some resources are intrinsically nonsharable,

Hold and Wait

To ensure that the hold-and-wait condition never occurs in the system, we must guarantee that, whenever a process requests a resource, it does not hold any other resources.

One protocol that can be used requires each process to request and be allocated all its resources before it begins execution.

. An alternative protocol allows a process may request some resources and use them. Before it can request any additional resources, however, it must release all the resources that it is currently allocated.

To illustrate the difference between these two protocols, we consider a process that copies data from a DVD drive to a file on disk, sorts the file, and then prints the results to a printer. If all resources must be requested at the beginning of the process, then the process must initially request the DVD drive, disk file, and printer. It will hold the printer for its entire execution, even though it needs the printer only at the end.

The second method allows the process to request initially only the DVD drive and disk file. It copies from the DVD drive to the disk and then releases both the DVD drive and the disk file. The process must then again request the

disk file and the printer. After copying the disk file to the printer, it releases these two resources and terminates.

Both these protocols have two main disadvantages. First, resource utilization may be low, since resources may be allocated but unused for a long period. Second, starvation is possible. A process that needs several popular resources may have to wait indefinitely, because at least one of the resources that it needs is always allocated to some other process.

No Preemption

To ensure that this condition does not hold, we can use the following protocol. If a process is holding some resources and requests another resource that cannot be immediately allocated to it (that is, the process must wait), then all resources currently being held are preempted.

Circular Wait

The fourth and final condition for deadlocks is the circular-wait condition. One way to ensure that this condition never holds is to impose a total ordering of all resource types and to require that each process requests resources in an increasing order of enumeration.

To illustrate, we let *R = {R1, Ri, ..., Rm}* be the set of resource types. We assign to each resource type a unique integer number, which, allows us to compare two resources and to determine whether one precedes another in our ordering.

We can now consider the following protocol to prevent deadlocks: Each process can request resources only in an increasing order of enumeration. That is, a process can initially request any number of instances of a resource type— say, Rj. After that, the process can request instances of resource type R; if and only if F(R;) > F(R,). If several instances of the same resource type are needed, a *single* request for all of them must be issued. Alternatively, we can require that, whenever a process requests an instance of resource type R,, it has released any resources *R.* such that *F{Rj) >F(Rj).*

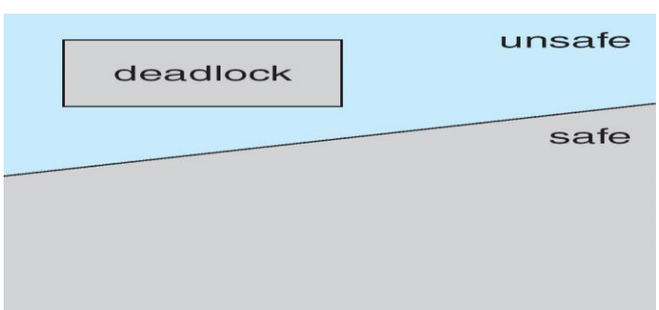
If these two protocols are used, then the circular-wait condition cannot hold.

4. Deadlock Avoidance

Possible side effects of preventing deadlocks are low device utilization and reduced system throughput. An alternative method for avoiding deadlocks is to require additional information about how resources are to be requested. With this knowledge of the complete sequence of requests and releases for each process, the system can decide for each request whether or not the process should wait in order to avoid a possible future deadlock. To making this decision the system consider the resources currently available, the resources currently allocated to each process, and the future requests and releases of each process.

Safe State

A state is *safe* if the system can allocate resources to each process (up to its maximum) in some order and still avoid a deadlock. More formally, a system is in a safe state only if there exists a safe sequence. A sequence of processes *<P1, P2, ..., Pn>*is a safe sequence for the current allocation state if, for each *Pi,* the resource requests that *Pi* can still make can be satisfied by the currently available resources plus the resources held by all *Pj,* with j < i. If no such sequence exists, then the system state is said to be *unsafe.*



Safe, unsafe, and deadlock state spaces.

A safe state is not a deadlocked state. Conversely, a deadlocked state is an unsafe state. Not all unsafe states are deadlocks. An unsafe state *may* lead to a deadlock.

To illustrate, we consider a system with 12 magnetic tape drives and three processes: P0/ *P1,* and P2. Process P0 requires 10 tape drives, process P1 may need as many as 4 tape drives, and process P2 may need up to 9 tape drives.Suppose that, at time *to,* process P0 is holding 5 tape drives, process *P1* is holding *2* tape drives, and process *P2* is holding 2 tape drives.

	Maximum Needs	Current Needs
P_0	10	5
P_1	4	2
P_2	9	2

At time To, the system is in a safe state. The sequence < P1, *P0, P2*>satisfies the safety condition. Process P1 can immediately be allocated all its tape drives and then return them (the system will then have 5 available tape drives); then process P0 can get all its tape drives and return them (the system will then have 10 available tape drives); and finally process P2 can get all its tape drives and return them

A system can go from a safe state to an unsafe state. Suppose that, at time T1, process *P2* requests and is allocated one more tape drive. The system is no longer in a safe state. At this point, only process P1, can be allocated all its tape drives. When it returns them, the system will have only 4 available tape drives.

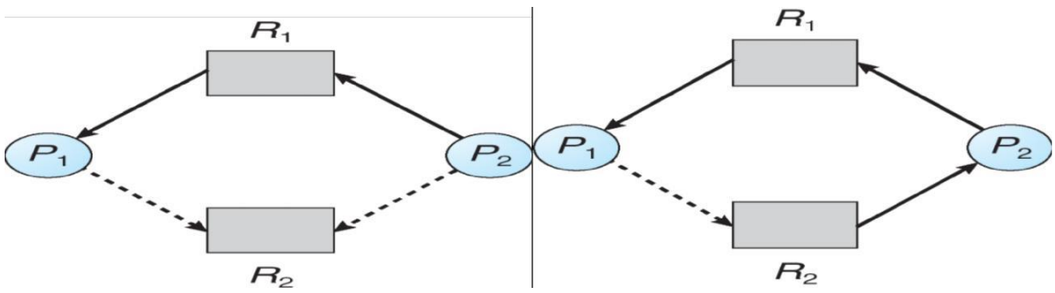
Resource-Allocation-Graph Algorithm

If we have a resource-allocation system with only one instance of each resource type, resource-allocation graph can be used for deadlock avoidance. In addition to the request and assignment edges, we introduce a new type of edge, called a claim edge.

A claim edge Pi —>*Rj* indicates that process *Pi* may request resource *Rj* at some time in the future. This edge resembles a request edge in direction but is represented in the graph by a dashed line. When process Pi requests resource *Rj,* the claim edge Pi —>*Rj* is converted to a request edge. Similarly, when a resource *Rj* is released by *Pi,* the assignment edge *Rj* -» P,- is reconverted to a claim edge *P;* —> *Rj.* Suppose that process P, requests resource *Rj.* The request can be granted only if converting the request edge P, —» *Rj* to an assignment edge *Rj* —>*Pi* does not result in the formation of a cycle in the resource-allocation graph.

If no cycle exists, then the allocation of the resource will leave the system in a safe state. If a cycle is found, then the allocation will put the system inan unsafe state. Therefore, process *P:* will have to wait for its requests to be satisfied.

To illustrate this algorithm, we consider the resource-allocation graph. Suppose that *P1,P2* requests *R2.* Although *R2* is currently free, we cannot allocate it to P2, since this action will create a cycle in the graph. A cycle indicates that the system is in an unsafe state. If P1 requests *R2,*and *P2* requests *R1,* then a deadlock will occur.



Resource-allocation graph for deadlock avoidance. An unsafe state in a resource-allocation graph.

Banker's Algorithm

The resource-allocation-graph algorithm is not applicable to a resource allocation system with multiple instances of each resource type. The *banker's algorithm* algorithm is applicable to such a system. The name was chosen because the algorithm could be used in a banking system to ensure that the bank never allocated its available cash in such a way that it could no longer satisfy the needs of all its customers.

When, a new process enters the system, it must declare the maximum number of instances of each resource type that it may need. This number may not exceed the total number of resources in the system. Then a the system must determine whether the allocation of these resources will leave the system in a safe state. If it will, the resources are allocated; otherwise, the process must wait until some other process releases enough resources.

Several data structures must be maintained to implement the banker's algorithm. These data structures encode the state of the resource-allocation system. Let *n* be the number of processes in the system and *m* be the number of resource types. We need the following data structures:

• **Available**. A vector of length *m* indicates the number of available resources of each type. If *Availab!c[f]* equals *k,* there are *k* instances of resource type *Ri* available.

• **Max**. An *n* x *m* matrix defines the maximum demand of each process. If Max[i][j] equals *k,* then process *Pi* may request at most *k* instances of resource type Rj.

9. **Allocation**. An *n* x mmatrix defines the number of resources of each type currently allocated to each process. If *Allocation[i][j]* equals *k,* then process *Pi* is currently allocated *k* instances of resource type Rj.

10. **Need**. An *n* x *m* matrix indicates the remaining resource need of each process. If *Need[i][j]* equals *k,* then process Pi may need *k* more instances of resource type *Rj* to complete its task. Note that *Need*[i][/] equals *Max[i][j]- Allocntion[i][j].*

To simplify the presentation of the banker's algorithm, we next use some notation. Let X and Y be vectors of length *n.* We say that X < Y if and only if *X[i]* < Y[i] for all i = 1, 2, ..., *n.* For example, if x= (1,7,3,2) and Y =(0,3,2,1), then Y < X. We can treat each row in the matrices *Allocation* and *Need* as vectors and refer to them as *Allocationi* and *Needi.* The vector *Allocationi,* specifies the resources currently allocated to process Pi; the vector *Needi* specifies the additional resources that process Pi may still request to complete its task.

Safety Algorithm

We can use this algorithm for finding out whether or not a system is in a safe state. This algorithm can be described, as follows:

1. Let *Work* and *Finish* be vectors of length *m*and *n,* respectively. Initialize

Work = A v a i l a b l e a n d *Fiiush[i]= false* for i=0 , 1 , ..., *n - l* .

2. Find an i such that both

 Finish[i] ==false

 Neei <= Work

If no such i exists, go to step

4. 3. *Work = Work +*

Allocation, Finish[i] = true

Go to step 2.

4. If *Finish[i]== true* for all. i, then the system is in a safe state.

This algorithm may require an order of *m* x n^2operations to determine whether a state is safe.

Resource-Request Algorithm

This algorithm determines if requests can be safely granted or not. Let *Requesti* be the request vector for process Pi.When a request for resources is made by process *Pi,* the following actions are taken:

 If *Requesti* <= *Needi,* go to step 2. Otherwise, raise an error condition, since the process has exceeded its maximum claim.

 If *Requesti* <= *Available,* go to step 3. Otherwise, *Pi* must wait, since the resources are not available.

 Have the system pretend to have allocated the requested resources to process Pi by modifying the state as follows: *Available = Available - Requesti;*

Allocationi, = Allocationi + Requesti;

Needi = Necdi - Requesti;

If the resulting resource-allocation state is safe, the transaction is completed, and process Pi is allocated its resources. However, if the new state is unsafe, then *Pi* must wait for *Request;*, and the old resource-allocation state is restored.

An Illustrative Example

Finally, to illustrate the use of the banker's algorithm, consider a system with five processes *P0* through P4 and three resource types *A, B,* and C. Resource type A has 10 instances, resource type *B* has 5 instances, and resource type *C* has 7 instances. Suppose that, at time To, the following snapshot of the system has been taken:

	Allocation	*Max*	*Available*
	A B C	*A B C*	*A B C*
P_0	0 1 0	7 5 3	3 3 2
P_1	2 0 0	3 2 2	
P_2	3 0 2	9 0 2	
P_3	2 1 1	2 2 2	
P_4	0 0 2	4 3 3	

The content of the matrix *Need* is defined to be *Max - Allocation* and is as follows:

	Need
	A B C
P_0	7 4 3
P_1	1 2 2
P_2	6 0 0
P_3	0 1 1
P_4	4 3 1

We claim that the system is currently in a safe state. Indeed, the sequence <*P1,* P3, *P4, P2, P0*>satisfies the safety criteria. Suppose now that process *P1* requests one additional instance of resource type *A* and two instances of resource type C, so *Request1* = (1,0,2). To decide whether this request can be immediately granted, we first check that *Request* < *Available*—that is, that (1 ,0,2) < (3,3,2), which is true. We then pretend that this request has been fulfilled, and we arrive at the following new state:

SACET

	Allocation	Need	Available
	A B C	A B C	A B C
P_0	0 1 0	7 4 3	2 3 0
P_1	3 0 2	0 2 0	
P_2	3 0 1	6 0 0	
P_3	2 1 1	0 1 1	
P_4	0 0 2	4 3 1	

We must determine whether this new system state is safe. To do so, we execute our safety algorithm and find that the sequence <*P1, P3, P4, P0, P2*>satisfies the safety requirement. Hence, we can immediately grant the request of process *P1.*

5. Deadlock Detection

If a system does not employ either a deadlock-prevention or a deadlock avoidance algorithm, then a deadlock situation may occur. In this environment, the system must provide:

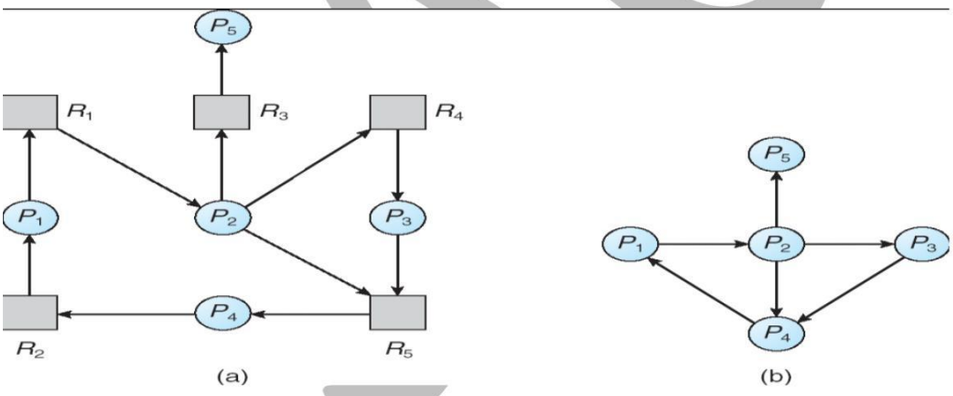
- An algorithm that examines the state of the system to determine whether a deadlock has occurred
- An algorithm to recover from the deadlock

Single Instance of Each Resource Type

If all resources have only a single instance, then we can define a deadlockdetection algorithm that uses a variant of the resource-allocation graph, called a *wait-for* graph. We obtain this graph from the resource-allocation graph by

removing the resource nodes and collapsing the appropriate edges.

More precisely, an edge from Pi to Pj in a wait-for graph implies that process Pi is waiting for process Pj to release a resource that Pi needs. An edge *Pi* -» Pj exists in a wait-for graph if and only if the corresponding resource allocation graph contains two edges Pi —>• *Rqj* and Rq, -» Pj for some resource



(a) Resource-allocation graph, (b) Corresponding wait-for graph.

A deadlock exists in the system if and only if the wait-for graph contains a cycle. An algorithm to detect a cycle in a graph requires an order of *n1* operations, where *n* is the number of vertices in the graph.

Several Instances of a Resource Type

The deadlock detection algorithm is applicable to multiple instances of each resource type. The algorithm employs several time-varying data structures that are similar to those used in the banker's algorithm .

. Available. A vector of length *m* indicates the number of available resources of each type.

- **Allocation.** An *n* x *m* matrix defines the number of resources of each type currently allocated to each process.
- **Request**. An *n* x *in* matrix indicates the current request of each process. If *Request[i][j]* equals *k,* then process Pi is requesting *k* more instances of resource type *Rj.*

Detection Algorithm

 Let *Work* and *Finish* be vectors of length *in* and *n,* respectively. Initialize *Work - Available.* For *i* = 0 , 1 , . . . , *n-1,* if *Allocation,* ^ 0, then *Finish[i] - false;* otherwise, *Finisli[i] = true.*

 Find an index *i* such that both

 Finish[i] -=false

 Requesti < Work

If no such / exists, go to step

4. 3. *Work - Work +*

Allocation! Finish[i] = true

Go to step 2.

4. If *Finish[i] == false,* for some /', 0 < / <*n,* then the system is in a deadlocked state. Moreover, if *Finish[i] == false,* then process *P;* is deadlocked.

This algorithm requires an order of *mx* n^2 operations to detect whether the system is in a deadlocked state.

To illustrate this algorithm, we consider a system with five processes *P0* through P4 and three resource types *A, B,* and C. Resource type *A* has seven instances, resource type B has two instances, and resource type C has six instances. Suppose that, at time To, we have the following resource-allocation state:

	Allocation	*Request*	*Available*
	A B C	*A B C*	*A B C*
P0	*0 1 0*	*0 0 0*	*0 0 0*
P1	*2 0 0*	*2 0 2*	
P2	*3 0 3*	*0 0 0*	
P3	*2 1 1*	*1 0 0*	
P4	*0 0 2*	*0 0 2*	

We claim that the system is not in a deadlocked state. Indeed, if we execute our algorithm, we will find that the sequence <Pn, *Pi, Pi, P*\, *PA*>results in *Finish[i]* -- *true* for all *i.*

Suppose now that process *Pj* makes one additional request for an instance of type C. The *Request* matrix is modified as follows:

	Request
	A B C
P0	0 0 0
P1	2 0 1
P2	0 0 1
P3	1 0 0
P4	0 0 2

We claim that the system is now deadlocked. Although we can reclaim the resources held by process Po, the number of available resources is not sufficient to fulfill the requests of the other processes. Thus, a deadlock exists, consisting of processes *Pi,* Pi, P3, and P4.

Recovery From Deadlock

When a detection algorithm determines that a deadlock exists, several alternatives are available. One possibility is to inform the operator that a deadlock has occurred and to let the operator deal with the deadlock manually. Another possibility is to let the system *recover* from the deadlock automatically. There are two options for breaking a deadlock. One is simply to abort one or more processes. The other is to preempt some resources from one or more of the deadlocked processes.

Process Termination

To eliminate deadlocks by aborting a process, we use one of two methods.

» **Abort all deadlocked processes.** This method clearly will break the deadlock cycle by abort all the dead locked processes, but it increases burden. The system need to restart all the aborted processes.

• **Abort one process at a time until the deadlock cycle is eliminated.** This method incurs considerable overhead, since, after each process is aborted, a deadlock-detection algorithm must be invoked to determine whether any processes are still deadlocked.

We must determine which deadlocked process (or processes) should be terminated. we should abort those processes whose termination will incur the given factors.

 What the priority of the process is

 How long the process has computed and how much longer the process will compute before completing its designated task

 How many and what type of resources the process has used .

 How many more resources the process needs in order to complete

 How many processes will need to be terminated?

Resource Preemption

In this, we successively preempt some resources from deadlocked processes and give these resources to other processes until the deadlock cycle is broken. If preemption is required to deal with deadlocks, then three issues need to be addressed:

• **Selecting a victim**. Which resources and which processes are to be preempted? As in process termination, we must determine the order of preemption to minimize cost.

• **Rollback**. If we preempt a resource from a process, We must roll back the process to some safe state and restart it from that state.

• **Starvation**. In a system where victim selection is based primarily on cost factors, it may happen that the same process is always picked as a victim. As a result, this process never completes its designated task, a starvation situation that must be dealt with in any practical system. Clearly, we must ensure that a process can be picked as a victim only a (small) finite number of times. The most common solution is to include the number of rollbacks in the cost factor.