What is an O.S?

O.S. is a program or system software that acts as an intermediary between user of a computer and the computer hardware.

The purpose of an O.S is to provide an environment in which a user can execute programs.

What are the goals of an O.S?

There are two primary goals.

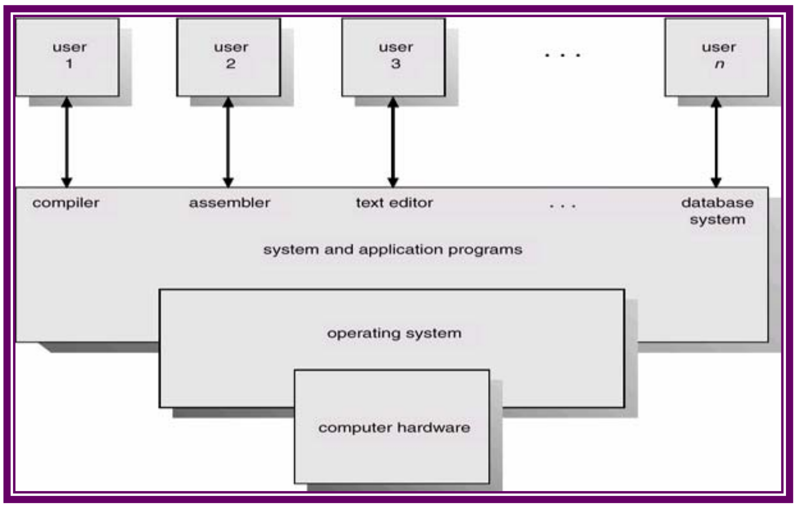
1) The primary goal of an O.S is to make the computer system convenient to use.

2) A secondary goal is to use the computer hardware in an efficient manner.

What are the main components of the computer system?

An OS is an important part of almost every computer system. A computer system can be divided into four components.

1) Hardware 2) O.S 3) Application programs 4) and the users



1) Hardware:- The H/W includes CPU, Memory and the I/O devices. The H/W provides the basic computing resources.

2)Application programs:-It contains database systems, games and business programs. These programs use resources to solve the computing problems of the users.

3)Operating System:-The O.S controls and coordinates the use of hardware among the various application programs for various users.

4) The users:- Users (people /computers) receive services from operating system.

What are the main functions of the O.S?

1) *The O.S is similar to a government*:- like a government , the O.S performs no useful function by it self. It simply provides an environment within which other programs can do useful work.

2) O.S acts as a Resource allocator:-

A computer system has many resources that may be required to solve a problem (cpu time, memory space, I/O devices, file storage etc). The O.S acts as the manager of these resources and allocates them to specific programs and users as necessary for their task. The O.S must decide which requests are allocated resources to operate the computer system efficiently and fairly.

3) *O.S acts a control program:-*

A control program controls the execution of user programs to prevent errors.

Briefly explain the various types of O.S?

1) Simple batch systems:-

Early computers were physically large machines run from a console. The common input devices were card readers and tape drives. The common output devices were line printers, tape drives and card punches. The users of such systems did not interact directly with the computer systems.

Here the user prepares a job. The job contains the program, the data and some control information about the nature of the job, and submitted it to the computer operator. The job would be in the form of the punch cards. After some time, the out put will be appeared. The out put contains the result of the program.

To speed up of the processing, jobs with similar needs were batched together and were run through the computer as a group. The programmers would give their programs to the operator. The operator would sort programs in to batches with similar requirements, and the computer became available, would run each batch. The out put from the each job would be sent back to the appropriate programmer.

A batch O.S normally reads a stream of separate jobs each with its own control cards that predefine the job does. When the job is complete, its output is usually printed.

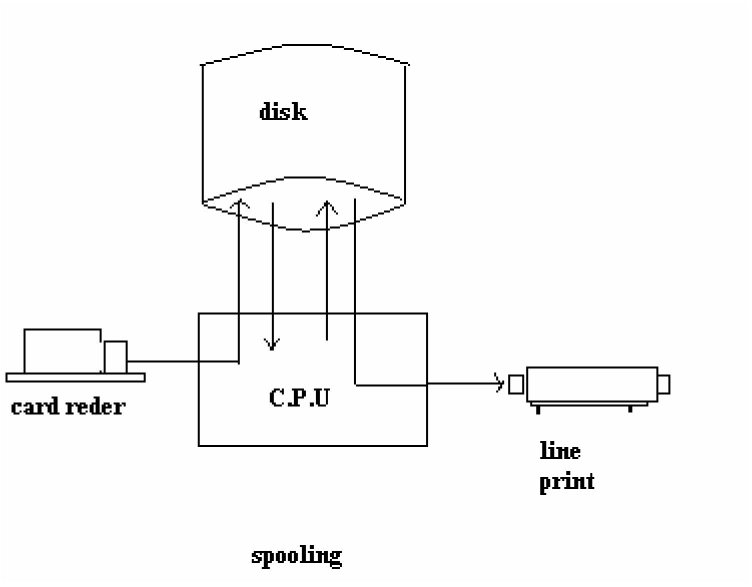
Disadvantages:-

1) The disadvantage of batch O.S is the lack of interaction between the user and the job while the job is executed.

2) In this environment the CPU is often idle.

Spooling:-

In simple batch systems the CPU is often idle. To solve this problem one method was introduced i.e spooling (simultaneous peripheral operation online).



Here disk technology was introduced. Instead of the cards being read from the card reader directly in to memory, cards are read directly from the card reader on to the disk. The location of the card images are recorded in a table kept by O.S. When a job is executed, the O.S satisfies its requests for card reader input by reading from the disk.

Similarly when the job requests the printer to output a line, that line is copied in to a system buffer and is written to the disk. When the job is completed, the output is actually printed. This form of process is called spooling.

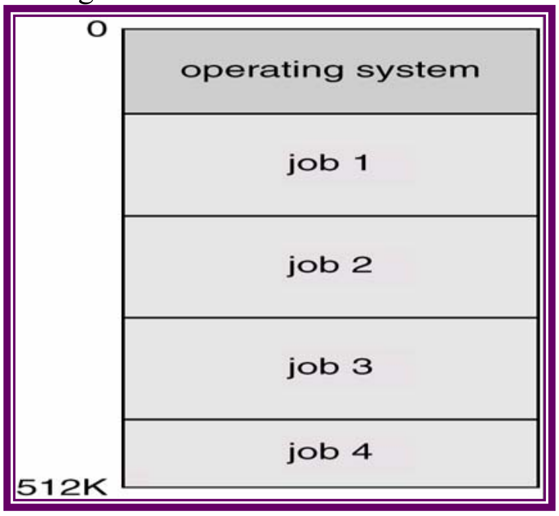
Multiprogrammed Batch Systems:-

When several jobs are on a disk, the O.S selects which job goes next for execution. The O.S selects the jobs on the basis of FCFS. This type of selection is called job *scheduling.* The most important part of job scheduling is the ability to multiprogramming. A single user can not keep either the cpu or I/O devices busy at all times. Multiprogramming increases CPU utilization by organizing jobs such that the CPU always has one to execute.

The idea is as follows. The O.S keeps several jobs in memory at a time. The O.S picks and begins to execute one of jobs in the memory. Eventually, the job may have to wait for some task, in non multi programmed system, the CPU would sit idle. In a multiprogramming system, the O.S simply switches to and executes another job. When that job needs to wait, the CPU is switches to another job, and so on. When the first job finishes waiting and gets the CPU back. As long as there is always some job to execute, the CPU will never be idle.

Disadvantages:-

In multiprogramming environment the user can not interact with the job when it is running.



Time sharing systems:-

Time sharing or *multitasking* is a logical extension of multiprogramming. Multiple jobs are executed by the CPU switching between them. But the switches occurs so frequently that the users may interact with each program while it is running.

An interactive computer system provides an on-line communication between the user and the system. The user gives instructions to the O.S or to a program directly, and immediately gets response.

A time sharing O.S uses CPU scheduling and multiprogramming to provide each user with a small portion of a time shared computer. Each user has at least one separate program in memory. A program that is loaded in to memory and is executed is called as a process. When a process executes, it typically executes for only a short time. A time shared O.S allows the many users to share the computer simultaneously. So each action or command in a time shared system tends to be short, only a little CPU time is needed for each user.

A time sharing O.S are even more complex than the multiprogramming O.S.

Personal computer systems:-

The computer systems are dedicated to the single user. Such systems are called PCs. The PCs appeared in 1970. They are microcomputers that are considerably smaller and less expensive than main frame systems. PC O.S is neither multi user nor multi tasking. How ever, the goals of this O.S have changed with time, instead of the maximizing CPU and peripheral utilization.

Parallel systems:- The system contains more than one CPU is called multiprocessor systems. These CPUs shares the computer buses, the clocks, and sometimes memory and peripheral devices. These systems are referred to as tightly coupled systems.

There are several reasons for making such systems.

1) Increasing the through put:-
By increasing the no. of processors, we hope to get more work done in a shorter period of time.

2) Multiprocessors can also save money compared to multiple single systems because processors can share peripherals.

3) Another reason is the increased reliability.

Ex:- If we have 10 processors and suppose one got failed, then the remaining nine processors will share the work of the failed processor. Thus the entire system runs 10% slower, rather than failing altogether.

The most common multiprocessor system now uses the *symmetric* multiprocessing model and some use *asymmetric model.*

Symmetric multiprocessing:-

In which each processor runs an identical copy of the O.S, and these copies communicate with one another as needed.

Asymmetric multiprocessing:-

In which each processor is assigned a specific task. **A** master processor controls the system. The client processors look for the master processor's instructions. The master processor schedules and allocates work to slave processors.

Distributed systems:-

A recent trend in computer systems is to distribute computation among several processors. Here the processors do not share memory or clocks. Instead of that each processor has its own local memory. The processors communicate with one another through various communication lines, such as high speed buses or telephone lines. These systems are usually referred as loosely coupled systems or distributed systems.

The processors in distributed systems may vary in size and function. These processors are referred to by a no. of different names, such as sites, nodes, and computers and so on.

Reasons to build Distributed Systems:-

1) Resource sharing:-

If no of different sites are connected to one another, then a user at one site may be able to use the resources available at another.

2) **Computation speed up**:-

A distributed system may allow us to distribute the computation among the various processors to run that computation concurrently.

3) **Reliability.-**

When one processor got failed in a distributed system, the remaining sites can potentially continue operations.

4) **Communication**:-

There are many instances in which programs need to exchange data with one another. Users may initiate file transfer or communication with one another via *Electronic Mails*
Ex:- A user can send mail to another user at the same site or at a different site.

Real time systems:-

Another type of systems is called as real time systems. The real time systems are used when there is a strict time requirement on the operation of the processor or flow of data. The real time O.S is used as control device in a dedicated application.
Ex:- Medical imaging systems.
A real time O.S has well defined, fixed time limit. Processing must be done with in the defined limit or the system will fail.
Real Time Systems are of two types.

Hard real time systems:-

It gives guarantee to complete the critical task on time. This goal requires that all system delays must be bounded.

Soft real time systems:-

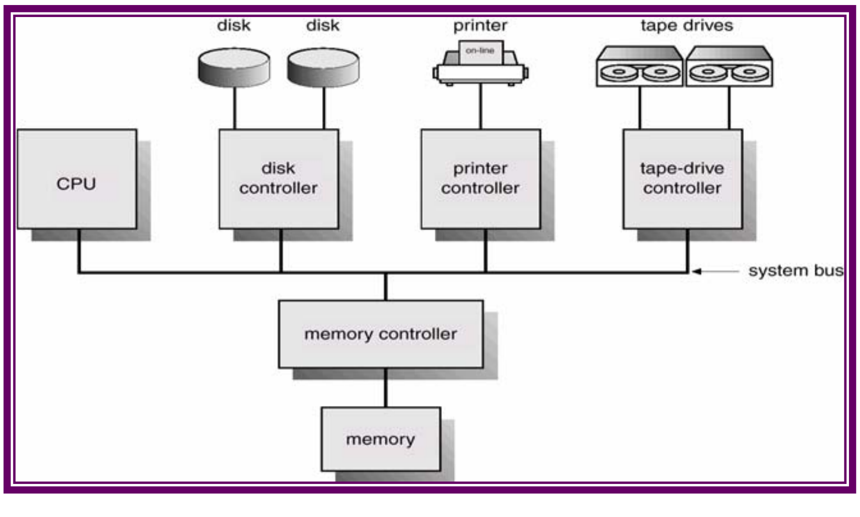
Less restrictive type of real time system is soft real time systems. Where a critical real time tasks get priority over other task, and retain that priority until it is completed.

*****

Computer System Operations:

A modern, general purpose computer system consists of a CPU and a number of device controllers that are connected through a common bus that provides access to shared memory. Each device controller is in charge of a specific type of device. The operating system then starts executing the first process, such as "init", and waits for some event to occur. The occurrence of an event is usually signaled by an interrupt from either the hardware or the software. Hardware may trigger an interrupt from either the hardware or the software. Hardware may trigger an interrupt at any time by sending a signal to the CPU, usually by way of the system bus. Software may trigger an interrupt by executing a special operation called a *System Call*.

Interrupts are an important part of computer architecture. Each computer design has its own interrupt mechanism, but several functions are common. The interrupt must transfer control to the appropriate interrupt service routine. The interrupt routine is then called indirectly through the table, with no intermediate routine needed. Generally, the table of pointers is stored in low memory. Interrupts are *disabled* while an interrupt is being processed, so any incoming interrupts are delayed until the operating system is done with the current one; then, interrupts are *enabled.*



I/O Structure:

A general purpose computer system consists of a CPU and a number of device controllers that are connected through a common bus. Each device controller is in charge of a specific type of device. Depending on the controller, there may be more than one attached device. For instance, the small computer systems interface controller, found on many small to medium sized computers. The device controller is responsible for moving the data between the peripheral devices that it controls and its local buffer storage. The size of the local buffer within a device controller varies from one controller to another, depending on the particular device being controlled.

I/O Interrupts:

To start an I/O operation, the CPU loads the appropriate registers within the device controller. Once the I/O is started, two courses of action are possible. In the simplest case, the I/O is started; then, at I/O completion, control is returned to the user process. This case is known as synchronous I/O. The other possibility called asynchronous I/O then can continue while other system operations occur.

We also need to be able to keep track of many I/O requests at the same time. For this purpose, the operating system uses a table containing an entry for each I/O device: the device-status table. Each table entry indicates device's type, address, and state.

DMA:

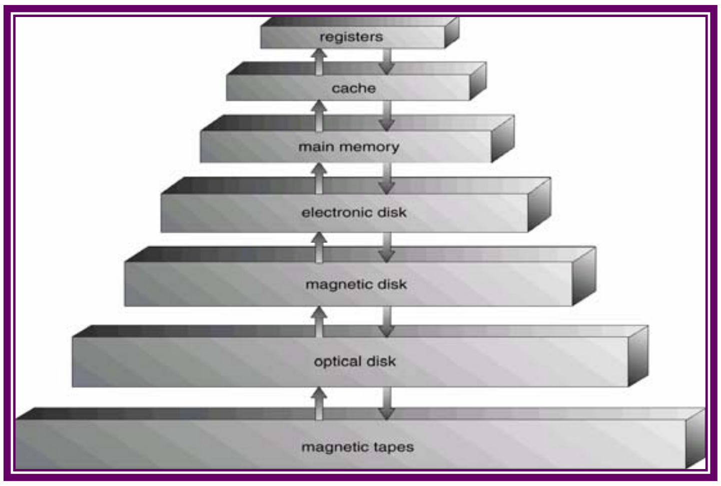
Definition: The transfer of data between a fast storage device such as magnetic disk and memory is often limited by the speed of the CPU. Removing the CPU from the path and letting the peripheral device manage the memory buses directly would improve the speed of transfer.

Direct Memory Access (DMA) is used for high-speed I/O devices. After setting up buffers, pointers, and counters for the I/O device, the device controller transfers an entire block of data directly to or from its own buffer storage to memory, with no intervention by the CPU. Only one interrupt is generated per block, rather than the one interrupt per byte generated for low – speed devices.

The basic operation of the CPU is the same. A user program, or the operating system itself, may request data transfer. The operating system finds a buffer from a pool of buffers for the transfer.

Storage device hierarchy:

The wide variety of storage system in a computer system can be organized in a hierarchy according to speed and their cost. The higher levels are expensive, but are fast. As we move down the hierarchy, the cost per bit generally decreases, whereas the access time generally increases. In addition to the speed and cost of the various storage systems, there is also the issue of storage volatility. 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. As we progress from bottom to top, then cost and speed increases but access time and capacity decreases.



*****

Caching:

Caching is an important principle of computer systems. Information is normally kept in some storage system. As it is used, it is copied into a faster storage system. As it is used, it is copied into a faster storage system the cache on a temporary basis. When we need a particular piece of information, we first check whether it is in the cache.

Since caches have limited size, cache management is an important design problem. Careful selection of the cache size and of a replacement policy can result in 80 to 99 percent of all accesses being in the cache, causing extremely high performance.

Main memory can be viewed as a fast cache for secondary memory, since data on secondary storage must be copied into main memory for use, and data must be in main memory before being moved to secondary storage for safekeeping. The file-system data may appear on several levels in the storage hierarchy.

Coherency and Consistency:

Hierarchical storage structure, the same data may appear in different levels of the storage system. For example, consider an integer A located in file B that is to be incremented by 1. Suppose that file B resides on magnetic disk. The increment operation proceeds by first issuing an I/O operation to copy the disk block on which A resides to main memory. This operation is followed by a possible copying of A to the cache, and by copying of A to an internal register.

The situation becomes more complicated in a multiprocessor environment where, in addition to maintaining internal registers, the CPU also contains a local cache. In such an environment, a copy of A may exist simultaneously in several caches. Since the various CPUs can all executer concurrently, we must make sure that an update to the value of A resides. This problem is called "*cache coherency*".

A distributed environment, the situation becomes even more complex. In such an environment, several copies of the same file can be kept on different computers that are distributed in space.

Hardware Protection:

Early computer systems were single-user programmer-operated systems. When the programmers operated the computer from the console, they had complete control over the system. As operating systems developed, however this control was given to the operating system. Starting with the resident monitor, the operating system began performing many of the functions, especially I/O, for which the programmer had been responsible previously.

This sharing created both improved utilization and increased problems. When the system was run without sharing, an error in a program could cause problems for only the one program that was running. With sharing, many processes could be adversely affected by a bug in one program.

Even more subtle errors can occur in a multiprogramming system, when one erroneous program might modify the program or data of another program, or even the resident monitor itself. MS-DOS and the Macintosh os both allow this kind of error.

Dual – Mode Operation:

The approach taken is to provide hardware support to allow us to differentiate among various modes of executions. At the very least, we need two separate modes of operation: user mode and monitor mode. A bit, called the monitor (0) or user (1). With the mode bit, we are able to distinguish between an execution that is done on behalf of the operating system, and one that is done on behalf of the user. As we shall see, this architectural enhancement is useful for many other aspects of system operation.

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

I/O Protection:

A user program may disrupt the normal operation of the system by issuing illegal I/O instructions, by accessing memory locations within the operating system itself, or by refusing to relinquish the CPU. We can use various mechanisms to ensure that such disruptions cannot take place in the system.

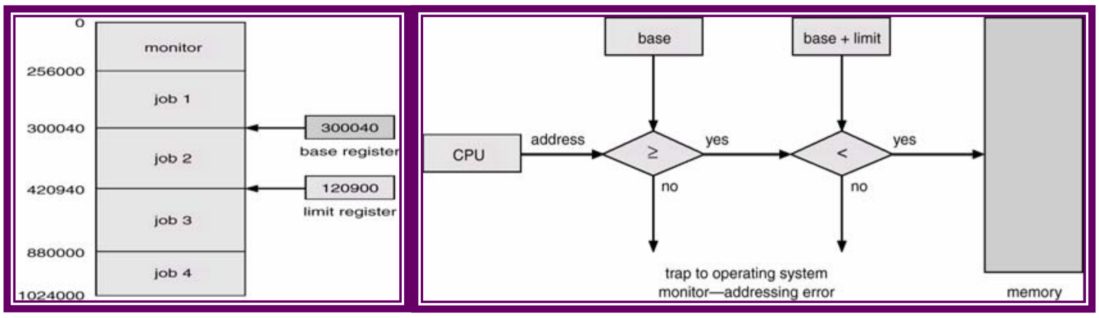
Consider the computer executing in user mode. It will switch to monitor mode whenever an interrupt or trap occurs, jumping to the address determined from the interrupt vector. Suppose that a user program, as part of its execution stores a new address in the interrupt vector. This new address could overwrite the previous address with an address with an address in the user program.

Memory Protection:

To ensure correct operation, we must protect the interrupt vector from modification by the user program. We must also protect the interrupt service routines in the operating systems from modification. Otherwise, a user program might overwrite instructions in the interrupt service routine with jumps to the user program, thus gaining control from the interrupt service routine that was executing in monitor mode. Even if the user did not gain unauthorized control of the computer, modifying the interrupt service routines would probably disrupt the proper operation of the computer system and of its spooling and buffering.

Users (programs) memory space is needed to separate by determining the range of legal addresses that the program may access, and to protect the memory outside that space. We can provide this protection by using two registers, usually a base and a limit. The base register holds the smallest legal physical memory address; the limit register contains the size of the range. For example, if the base register holds 300040 and limit register is 120900, then the program can legally access all addresses from 300040 through 420940 inclusive.

The base and limit registers can be loaded by only the operating system, which uses a special privileged instruction. Since privileged instructions can be executed in only monitor mode, and since only the operating system executes in monitory mode, only the operating system can load the vase and limit registers. This scheme allows the monitor to change the value of the registers, but prevents user programs from changing the registers contents.



CPU Protection:

The third piece of the protection puzzle is ensuring that the operating system maintains control. We must prevent a user program from getting shuck in infinite loop, 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.

Another use of the timer is to compute the current time. A timer interrupt signals the passage of some period, allowing the operating system to compute the current time in reference to some initial time. If we have interrupts every 1 second, and we have had 1427 interrupts since we were told that it was 1:00 PM, and then we can compute that the current time is 1:23:42 PM. Some computers determine the current time in this manner, but the calculations must be done carefully for the time to be kept accurately, since the interrupt-processing time tends to cause the software clock to slow down. Most computers have a separate hardware time-of-day clock that is independent of the operating system.

*****

Operating System Components

Even though, not all systems have the same structure many modern operating systems share the same goal of supporting the following types of system components.

1. Process Management

The operating system manages many kinds of activities ranging from user programs to system programs like printer spooler, name servers, file server etc. Each of these activities is encapsulated in a process. A process includes the complete execution context (code, data, PC, registers, OS resources in use etc.).It is important to note that a process is not a program. A process is only ONE instant of a program in execution. There are many processes can be running the same program. The five major activities of an operating system in regard to process management are

- *Creation and deletion of user and system processes.*
- *Suspension and resumption of processes.*
- *A mechanism for process synchronization.*
- *A mechanism for process communication.*
- *A mechanism for deadlock handling.*

2. Main-Memory Management

Primary-Memory or Main-Memory is a large array of words or bytes. Each word or byte has its own address. Main-memory provides storage that can be access directly by the CPU. That is to say for a program to be executed, it must in the main memory. The major activities of an operating in regard to memory-management are:

- Keep track of which part of memory are currently being used and by whom.
- Decide which process is loaded into memory when memory space becomes available.
- Allocate and deal locates memory space as needed.

3. File Management

A file is a collected of related information defined by its creator. Computer can store files on the disk (secondary storage), which provide long term storage. Some examples of storage media are magnetic tape, magnetic disk and optical disk. Each of these media has its own properties like speed, capacity, and data transfer rate and access methods. A file system normally organized into directories to ease their use. These directories may contain files and other directions. the five main major activities of an operating system in regard to file management are

- *The creation and deletion of files.*
- *The creation and deletion of directories.*
- *The support of primitives for manipulating files and directories.*
- *The mapping of files onto secondary storage.*
- *The back up of files on stable storage media (several copies of same data).*

4. I/O System Management

I/O subsystem hides the peculiarities of specific hardware devices from the user. Only the device driver knows the peculiarities of the specific device to which it is assigned.

- A Memory management component including buffering, caching and spooling.
- A general device-driver interface.
- Drivers for septic H/W devices.

5. Secondary-Storage Management

Generally speaking, systems have several levels of storage, including primary storage, secondary storage and cache storage. Instructions and data must be placed in primary storage or cache to be referenced by a running program. Because main memory is too small to accommodate all data and programs, and its data are lost when power is lost, the computer system must provide secondary storage to back up main memory. Secondary storage consists of tapes, disks, and other media designed to hold information that will eventually be accessed in primary storage (primary, secondary, cache) is ordinarily divided into bytes or words consisting of a fixed number of bytes. Each location in storage has an address; the set of all addresses available to a program is called an address space.

The three major activities of an operating system in regard to secondary storage management are:

- Managing the free space available on the secondary-storage device.
- Allocation of storage space when new files have to be written.
- Scheduling the requests for memory access.

6. Networking

A distributed system is a collection of processors that do not share memory, peripheral devices, or a clock. The processors communicate with one another through communication lines called network. The communication-network design must consider routing and connection strategies, and the problems of contention and security.

7. Protection System

If computer systems has multiple users and allows the concurrent execution of multiple processes, then the various processes must be protected from one another's activities. Protection refers to mechanism for controlling the access of programs, processes, or users to the resources defined by computer systems.

8. Command Interpreter System

A command interpreter is an interface of the operating system to the user. The user gives commands which are executed by operating system (usually by turning them into system calls). The main function of a command interpreter is to get and execute the next user specified command. Command-Interpreter is usually not a part of the kernel, since multiple command interpreters (shell, in UNIX terminology) may be supported by an operating system, and they do not really need to run in kernel mode.

*****

Explain Operating System Services

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

1. Program execution:

The system must be able to load a program into memory and to run int. The program must be able to end its execution, either normally or abnormally.

2. I/O operations:

A running program may require I/O. This I/O may involve a file or an I/O device. For specific devices, special functions may be desired. For efficiency and protection, users usually cannot control I/O devices directly. Therefore the operating system must provide some means to do I/O.

3. File – System manipulation:

The file system is of particular interest. It should be obvious that programs need to read and write files. They also need to create and delete files by name.

4. Communications:

There are many circumstances in which one process needs to exchange information with another process. Communications may be implemented via shared memory or the technique of Message passing in which packets of information are moved between processes by the O.S.

5. Error detection:

The operating system constantly needs to be aware of possible errors. Errors may occur in the CPU and memory hardware (such as memory error or power failure).For each type of error the OS should take the appropriate action to ensure correct and consistent computing.

6. Resource allocation:

Multiple users or multiple jobs running at the same time, resource must be allocated to each of them. Many different types of resources are managed by the operating system. One such routine locates an unused tape drive and marks an internal table to record the drive's new user. Another routine is used to clear that table. These routines may also be used to allocate plotters, modems, and other peripheral devices.

7. Accounting:

We want to keep track of which users use how much and what kinds of computer resources. This record keeping may be for accounting or simply for accumulating usage statistics. Usage statistics may be a valuable tool for researchers who wish to reconfigure the system to improve computing services.

8. Protection:

The owners of information stored in a multi-user computer system may want to control its use. Protection involves ensuring that all access to system resources is controlled. Security of the system from outsiders is also important.

*****

Purpose of System Calls:

When a computer is turned on, the program that gets executed first is called the operating system. It controls all activities of the computer. This includes who logs in, how disks are used, how much memory is used, how the CPU is used, and how you talk with other computers.

The way that programs talk to the operating system is via ``system calls." A system call looks like a procedure call (see below), but it's different -- it is a request to the operating system to perform some activity.

System calls are expensive. While a procedure call can usually be performed in a few machine instructions, a system call requires the computer to save its state, let the operating system take control of the CPU, have the operating system perform some function, have the operating system save its state, and then have the operating system give control of the CPU back to you.

System calls can be roughly grouped into five major categories:
Process Control, File Manipulation, Device Manipulation, Information Maintenance, and Communications.

- Process control
    - end, abort
    - load, execute
    - create process, terminate process
    - get process attributes, set process attributes
    - wait for a time
    - wait event, signal event
    - allocate and free memory
- file manipulation
    - create file, delete file
    - open ,close
    - read, write, reposition
    - get file attributes, set file attribute
- device manipulation
    - request device, release device
    - read, write, reposition
    - get device attributes, set device attributes
    - logically attach or detach devices
- information maintenance
    - get time or date, set time or date
    - get system data, set system data
    - get process, file, or device attributes
    - set process, file, or device attributes
- communications
    - create, delete communication connection
    - send, receive messages
    - transfer status information
    - attach or detach remote devices

1. Process Control:

Running program needs to be able to halt its execution either normally (**end)** or abnormally (**abort)**.

A process or job executing one program may want to **load** and **execute** other program. This feature allows the command interpreter to execute a program as directed.

If control returns to the existing program when the new program **terminates** we must save the memory image of the existing program. When a new job or process to be programmed, System calls like **create process** or submit job may be used.

If we create a new job or process, or perhaps even a set of jobs or processes, we should be able to control its execution of a job process including the job's priority, its maximum allowed execution time and so on (**get process attributes** and **set process attributes**).

After creating a new job or process, we may need to wait for them to finish their execution. We may want to wait for a certain amount of time (**wait time**), more probably, we may want to wait for a specific event to occur (**wait event**). The jobs or processes should then signal when that event has occurred (**signal event**).

2. File manipulation:

The file system is needed to **create** and **delete** files. Once the file is created, we need to **open** it and use. We may also read, write, or reposition. Finally we need to **close** the file. To know the properties of a file, we need **get attribute** and **set attribute** system calls.

3. Device manipulation:

A program, as it is running we must first request the device, to ensure exclusive use of it. After we are finished work with the device, we must release it. Once the device has been requested, we can **read, write** and **reposition** the device just as we can with ordinary files.

4. Information Maintenance:

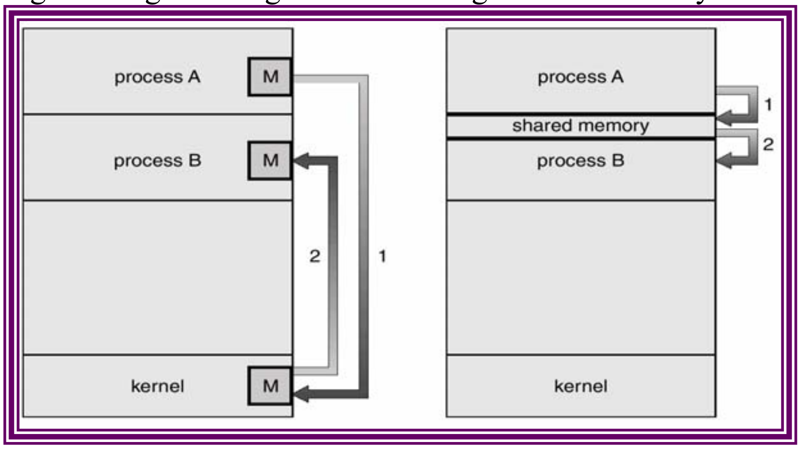
Many system calls exit simply for the purpose of transferring information between the user program and the O.S. For example most systems have a system call to return the current **time** and **date.** Other system calls may return information about the system, such as the number of current users, the version number of the operating system.

Generally, there are also calls to reset the process information (get process attributes and set process attributes).

5. Communication:

There are two common models of communication. Firstly, In the **message-passing** model, information is exchanged through an interprocess-communication facility provided by the operating system. Before communication can take place, a connection must be opened. The name of the other communicator must be known be it another process on the same CPU, or a process on another computer connected by a communications network. Each computer in a network has a host name by which it is commonly known. Similarly, each process has a process name, which translated into an equivalent identifier by which the operating system can refer to it. The **get hostid,** and get **processid** system calls do this translation. These identifiers are then passed to the general-purpose open and close calls provided by the file system, or to specific **open connection** and **close connection** system calls, depending on the system's model. Secondly, In **shared memory** model, process can share a common address space on mutual agreement to communicate each other.

Fig: Message Passing Fig: Shared Memory



*****

System Programs:

Another aspect of a modern system is the collection of system programs. System programs provide a more convenient environment for program development and execution. Some of them are simply user interfaces to system calls, whereas others are considerably more complex. They can be divided into several categories

- **File manipulation:** These programs create, delete, copy, rename, print, dump, list, and generally manipulate files and directories.
- **Status information:** Some programs simply ask the system for the date, time, amount of available memory or disk space, number of users, or similar status information. That information is then formatted, and is printed to the terminal or other output device or file.
- **File modification:** Several text editors may be available to create and modify the content of files stored on disk or tape.
- **Programming-language support:** Compilers, assemblers, and interpreters for common programming languages (such as FORTRAN, COBOL, Pascal, BASIC, C, and LISP).
- **Program loading and execution:** Once a program is assembled or compiled, it must be loaded into memory to be executed. The system may provide absolute loaders, re locatable loaders, linkage editors, and overlay loaders.
- **Communications:** These programs provided the mechanism for creating virtual connection among processes, users, and different computer systems. They allow users to send messages to each other's screens, to send larger messages as electronic mail, or to transfer files from one machine to another, and even to use other computers remotely as though these machines were local (known as remote login).
- **Application programs:** Most operating systems are supplied with programs that are useful to solve common problems, or to perform common operations.

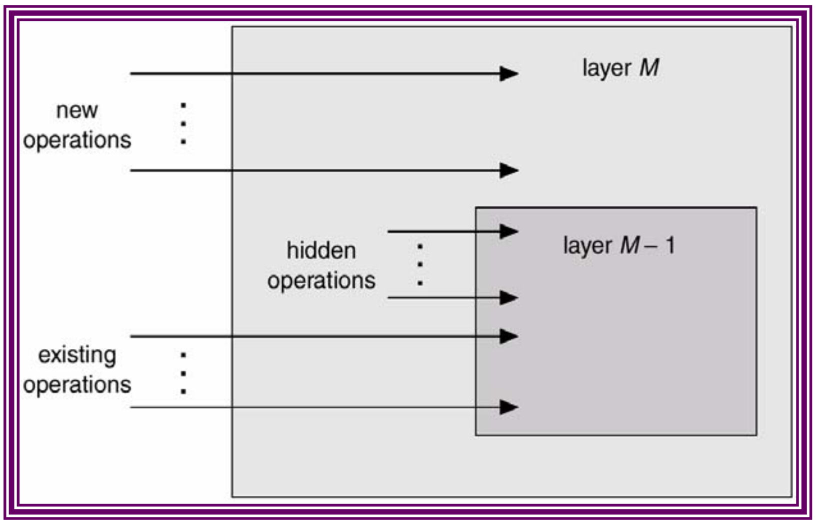
******

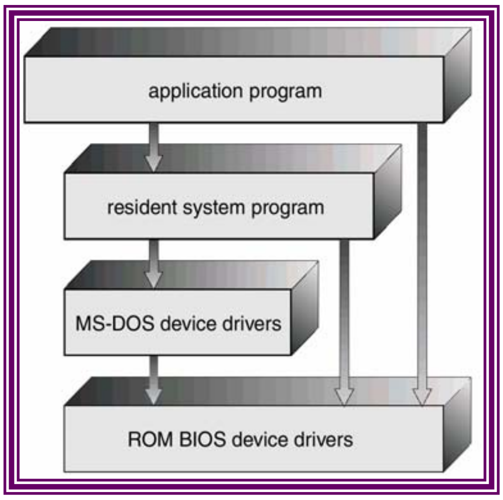
System Structure:

There are numerous commercial systems that do not have a well-defined structure. Frequently, such operating systems started as small, simple, and limited systems, and then grew beyond their original scope. MS-DOS is an example of such a system. It was originally designed and implemented by a few people who had no idea that it would become so popular.

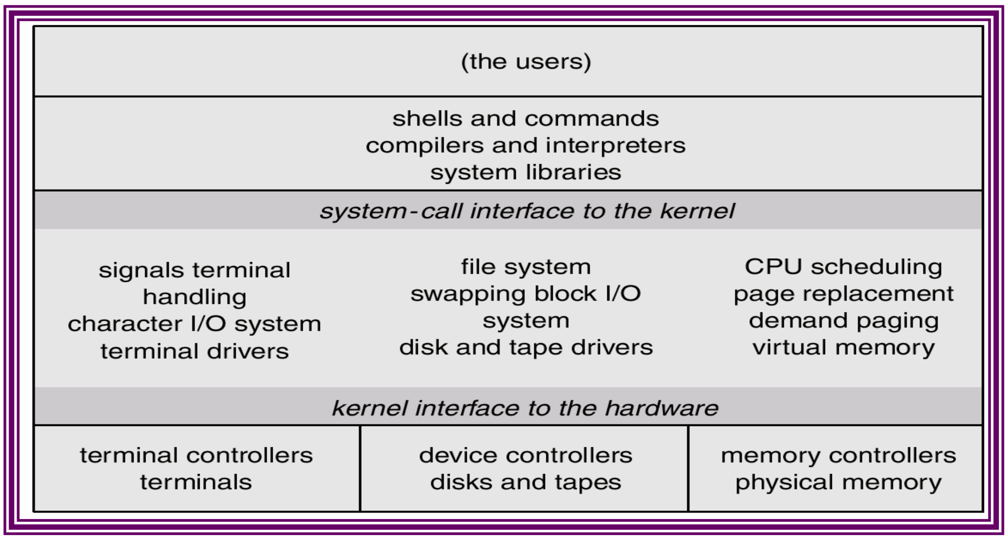
In MS-DOS, the interfaces and levels of functionality are not well separated. For instance, application programs are able to access the basic I/O routines to write directly to the display and disk drives. Such freedom leaves MS-DOS vulnerable to errant (or malicious) programs, causing entire system crashes then user programs fail. Of course, MS-DOS was also limited by the hardware of its era. Because the Intel 8088 for which it was written provides no dual mode and no hardware protection, the designers of MS-DOS had no choice but to leave the hardware accessible.

OS Layer





MS-Dos Layer Structure



UNIX system structure

******

Virtual Machines:

A computer system is made up of layers. The hardware is the lowest level in all such systems. The kernel running at the next level uses the hardware instructions to create a set of system calls for use by outer layers. The systems programs above the kernel are therefore able to user either system calls or hardware instructions. The application programs may view everything under them in the hierarchy as though the latter were part of the machine itself. This layered approach is taken to its logical conclusion in the concept of a Virtual machine. The VM approach on the other hand, does not provide any additional function, but rather provides an interface that is identical to the underlying bare hardware.

A major difficulty with the virtual machine approach involves disk systems. Suppose that the physical machine has three disk drives but wants to support seven virtual machines.

The virtual machine software can run in monitor mode. The virtual machine itself can execute in only user mode.

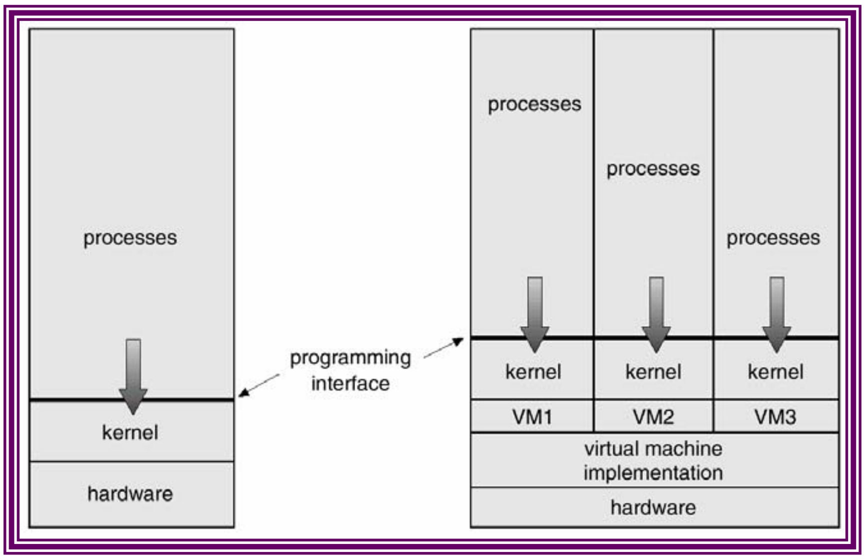


Fig: Non Virtual Machine **Fig:Virtual Machine**

Benefits:

On the other hand, there is no direct sharing of resources. To provide sharing, two approaches have been implemented. First, it is possible to share a minidisk. This scheme is modeled after a physical shared disk, but is implemented by software. With this technique, files can be shared. Second, it is possible to define a network of virtual mechanism each of which can send information over the virtual communications network.

*****

System design and Implementation:

Design Goals:

The first problem in designing a system is to define the goals and specifications of the system. At the highest level, the design of the system will be affected by the choice of hardware and type of system: batch, time-shared, single user multi user, distributed real time, or general purpose.

Beyond this highest design level, the requirements may be much harder to specify. The requirements can be divided into two basic groups: user goals and system goals.

Mechanisms and policies:

One important principle is the separation of policy from mechanism. Mechanisms determine how to do something. In contrast, policies decide what will be done. For example, a mechanism for ensuring CPU protection is the timer construct. The decision of how long the timer is set for a particular user, on the other hand, is a policy decision. OS take the separation of mechanism and policy to extreme, by implementing a basic set of primitive building blocks.

Implementation:

Once an operating system is designed, it must be implemented. Traditionally, operating systems have been written in assembly language. However, that is generally no longer true. OS can now be written in higher level languages. The advantage of using high level language, or at least a systems implementation language, for implementing OS are the same as those accrued when the language is used for application programs. The first system that was not written in assembly language was probably of Master Control Program(MCP) for Burroughs computers.

Explain System Generation

OS are 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 configured or generated for each specific computer site. This process is sometimes known as system generation (SYSGEN).

The following kinds of information must be determined.

What CPU is to be used? What Options are installed? For multiple CPU systems, each CPU must be described.
How much memory is available? Some systems will determine this value themselves by referencing memory location after memory location until an illegal address fault is generated.
What devices are available? The system will need to know how to address each device, the device interrupt number, the device's type and model, and any special device characteristics.
What Operating System options are used? These options or values might include how many buffers of which sizes should be used, what type of CPU scheduling algorithm is desired, what the maximum number of processes to be supported is, and so on.

Define Bootstrap Program

Does the hardware know where the kernel is, or how to load it? The procedure of starting a computer by loading the kernel is known as **booting** the system. On most computer systems, there is a small piece of code, stored in ROM, known as the **bootstrap program.**

*****

Processes

Def: - **Process:-** A process is a program in execution. The execution of process must progress in a sequential manner. i.e at any time at most one instruction is executed on behalf of the process.

A process generally also includes the process stack, contain temporary data and a data section contain variables.

Process states:- As process executes it changes state. Each process may be in one of the following states.

New: - The process is being created.

Running: - Instructions are being executed.

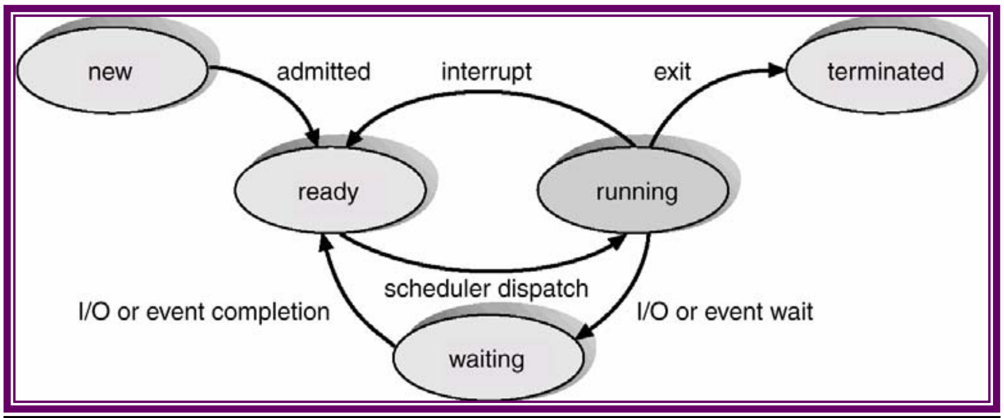
Waiting state: - The process is waiting for an I/O or an event to occur.

Ready state: - The process is waiting to be assigned to a processor

Terminated:- The process has finished execution.

Only one process can be in running state at any time. Many processes may be in all other states.

Process State Diagram

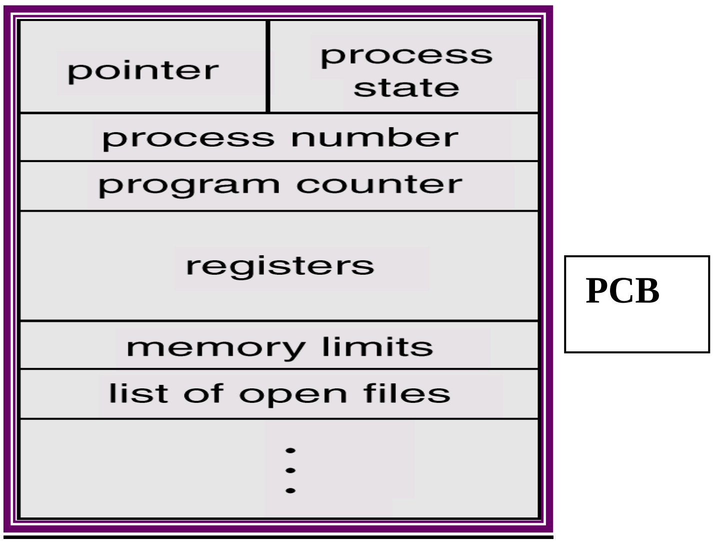


Process control block (PCB): Each process is represented in the operating system by a process control block. This is also called as the **task control block. It contains many** pieces of information. They are as follows.

1. Process state: The state may be new, ready, running, wait or terminate.

2. Program counter:- The P.C indicates the address of the next instruction to be executed.

3. C.P.U registers:- The contents of registers may vary depending on the computer architecture. It contains accumulators, index registers, base registers, any conditional code will be stored, also saves the state of the process when an interrupt can occur, to allow the process to be continued correctly afterward.



4. C.P.U Scheduling:- Includes the process priority, pointers to scheduling queues and any scheduling information.

5. Memory management information:- This includes the value of base and limit registers, the page tables or the segment tables.

6. Accounting information:- This includes the amount of C.P.U and real time used, time limits, account numbers, job or process numbers and so on….

7. I/O status information:- The information includes the list of I/O devices allocated to this process, a list of open files and so on…

A PCB simply is a repository for any information of a particular process from time to time.

*****

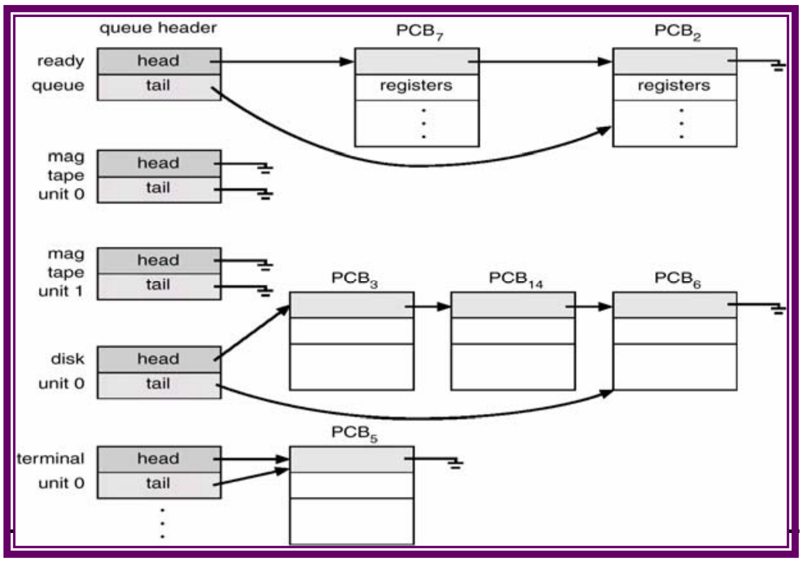
Explain about Process scheduling

Scheduling queue:-

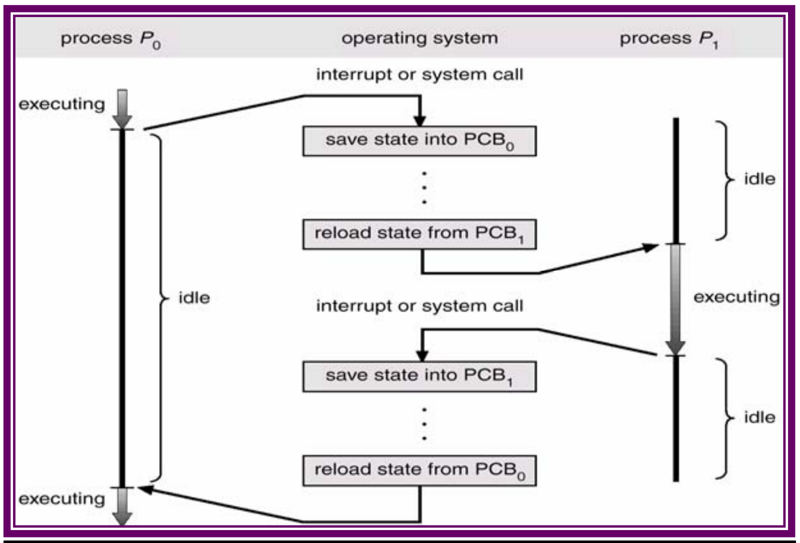
i) Job queue:- As process enter the system, they are all put in to the queue called job queue.

ii) Ready queue:- The process that are residing in main memory and are ready and waiting to execute are kept in a list called ready queue. This queue is generally stored as a linked list. A ready queue header will contain pointers to the first and last PCB's in the list. Each PCB has a pointer field that points to the next process in the ready queue.

iii) Device queue:- some times many processes need to wait for an I/O device such as disk, printer or tape etc… the list of processes waiting for a particular I/O device has its own device queue. Each device has its own device queue.



Device Queues



CPU switch from process to process

Queuing diagram:- a common representation for a discussion of process scheduling is a queuing diagram.

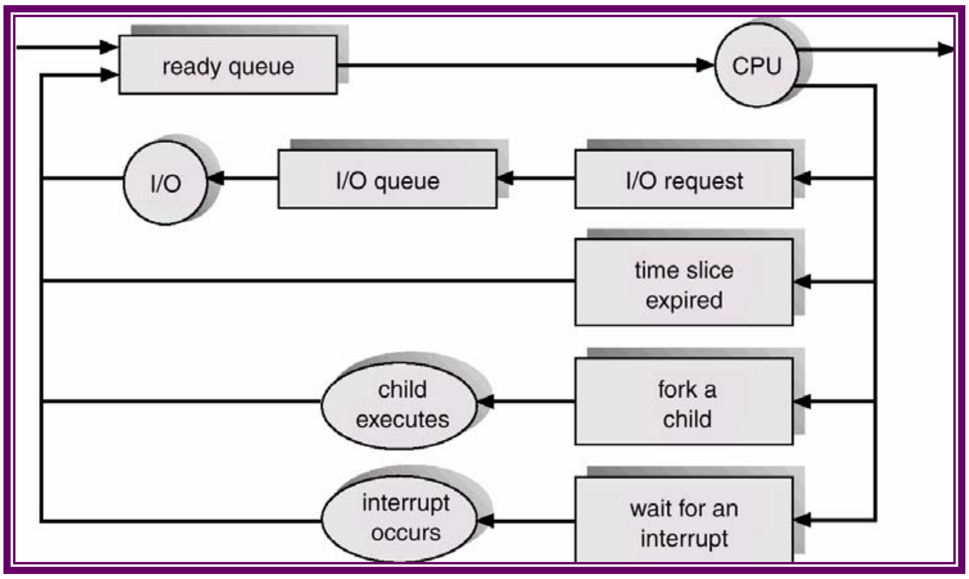
Each rectangle represents the queue. Two types of queues are present:

1) Ready queue 2) set of device queues

The circles represent the resource that serves the queues and arrow indicates the flow of processes in the system. A new process is initially put in the ready queue. It waits until it is allocated the CPU

Once the process is allocated the CPU the process may leave the CPU for the following reasons.

Queuing Diagram



Schedulers:-

A process migrates between the various scheduling queues through out its life time. The OS must select processes from these queues in some fashion. The selection process is carried out by the appropriate Scheduler.

There are 3 types of schedulers.

1) Long term Scheduler Or Job scheduler:-

Long term scheduler selects process from a mass-storage device, and loads them into memory for execution. The long term scheduler executes much less frequently. The long term scheduler controls the degree of multiprogramming.

If the degree of multi programming is constant then the average rate of process creation must be equal to the average departure rate of processes leaving the system. Thus the L.T.S needs to be invoked only when a process leaves the system.

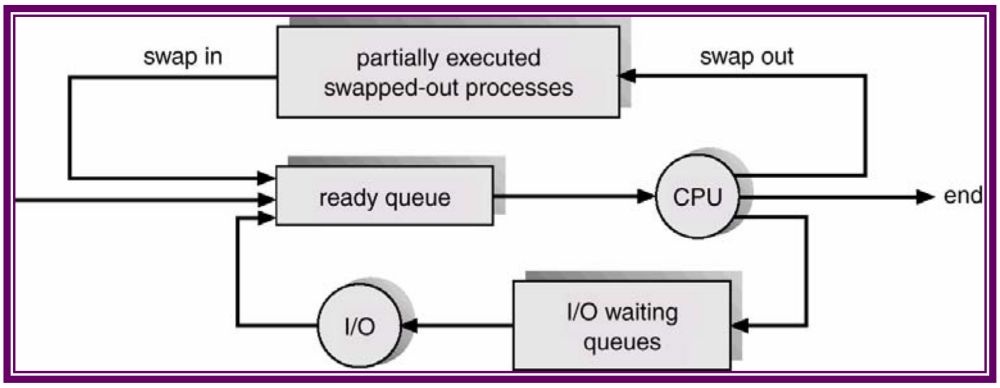
It is important that long term scheduler must select jobs in a careful way. It should not select all process of I/O bound process is one that spends more of its time doing I/O than it spends doing computations. If all processes are CPU bound, device queue are always empty. So, long term scheduler must select a good process mix of I/O and CPU bound.

2) Short term Scheduler (c.p.u scheduler) :- The short term scheduler selects from among the processes that are ready to executes, and allocates the CPU to one of them.

The short term scheduler must select a new process for the c.p.u quite frequently. The short term scheduler must be very fast. Otherwise the c.p.u time may be wasted.

3) **Medium term scheduler:-** in some systems long term scheduler may be absent or minimal. In these systems simply puts every new process in memory for the short term scheduler. But at some stage when the degree of multiprogramming is over or increased then the CPU utilization drops.

Now the medium term scheduler removes one process from memory to decrease the degree of multiprogramming. At a later time the process can be brought again for its execution. This is called swapping. The process is swapped out and swapped in by the M.T.S.



Context Switch:- Switching the CPU to another process requires saving the state of the old process and loading the saved state for the new process. This task is known as context switch.

They are highly dependent on hardware support. Context switch time is pure overhead, because the system does no useful work while switching. Its speed varies from machine to machine, depending on the memory speed, the number of registers which must be copied, and the existence of special instructions. Typically, the sped ranges from 1 to 1000 microseconds.

Explain about Operation on process

Thus the O.S must provide a mechanism for process creation and termination.

Process creation:- A process may creates several new processes by using the system call CREATE- process.

The creating process may in turn create other processes, forming a tree of processes.

In general, a process will need certain resources to accomplish its task. When a process creates a sub process, the sub process may be able to obtain its resources directly from the O.S 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 among several of its children.

When a process creates a new process, two possibilities exist in terms of execution.

1) The parent continues to execute concurrently with its children.

2) The parent waits until some or all of its children terminated.

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

1) The child process is the duplicate of the parent process.

2) The child process has a program loaded into it.

Process termination:- A process terminates when it finishes executing its last statement and ask the O.S to delete it by using the EXIT system call.

At that point, the process may return data or output to its parent process. All of the resources of the process, including physical and virtual memories, open files and I/O buffers are de allocated by the O.S.

A process can cause the termination of another process via an appropriate system call. Ex:- ABORT.

Usually, such a system call can be invoked by only the parent of the process that is to be terminated.

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

1) A child has exceed its usage of some of the resources it have been allocated.

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

3) The parent is exiting and O.S does not allow a child to continue if its parent terminates.

If a process terminates either normally or abnormally all its children's must also be terminated. This phenomenon is referred to as **cascading termination.**

******

Inter Process Communication (IPC):

Processes executing concurrently in the operating system may be either independent processes or cooperating processes. There are two models if IPC. They are **i) Shared Memory ii) Message Passing**

A process is **independent** if it can not affect or affected by the other processes executing in the system. I.e. any process that does not share any data with any other process is independent.

A process is **cooperating** if it can affect or 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.**

1) Information sharing:- Several users may be interested in the same piece of information we must provide an environment to allow concurrent access to these of resources.

2) 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. Such speed up can be achieved only if the computer has multiple processing elements.

3) Modularity:- To make the system in modular fashion, dividing the system function into separate processes.

4) Convenience:- Even an individual user may have many tasks to work on at one time. For instance, a user may be editing, printing and compiling in parallel.

Concurrent execution requires cooperation among the processes and requires mechanism to allow processes to communicate with each other and to synchronize their action.

IPC through shared memory:

Producer- consumer problem is a common paradigm for cooperating processes. A producer process produce information that is consumed by a consumer process. We must have available a buffer of items that can be filled by the producer and emptied by the consumer. The producer and consumer must be synchronized. The buffer may be provided by the O.S or explicitly coded by the application programmer with the use of shared memory.

The producer and consumer can share the following variables.

```
#define BUFFER_SIZE 10

Typedef struct {

    . . .

} item;

item buffer[BUFFER_SIZE];

int in = 0;

int out = 0;
```

The shared buffer is implemented as a circular array with two logical pointers. In, out.

In:- points to the next free position in the buffer.

Out:- points to the first full position in the buffer.

The buffer is empty when **in=out** and the buffer is full when

in+1 % BUFFER_SIZE = = out.

The code for the producer and consumer processes as follows :

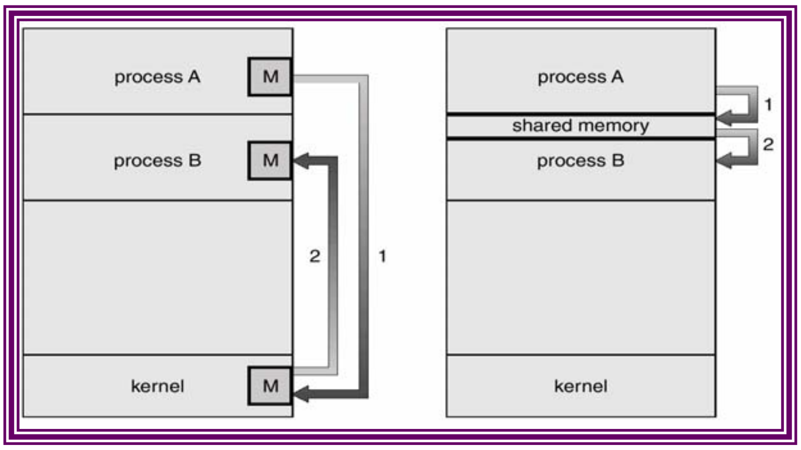
The producer process has a local variable (nextproduced) in which the new item to be produced is stored.

```
item nextProduced;

    do {

            while (((in + 1) % BUFFER_SIZE) == out)

                    ; /* do nothing */

            buffer[in] = nextProduced;

            in = (in + 1) % BUFFER_SIZE;

    }; while(TRUE)
```

The consumer process has a local variable (nextconsumed) in which the item to be consumed is stored.

```
    item nextConsumed;

    do {

            while (in == out)

                    ; /* do nothing */

            nextConsumed = buffer[out];

            out = (out + 1) % BUFFER_SIZE;

    }; while(TRUE)
```

Fig: Message Passing Fig: Shared Memory



Inter process communication through Message Passing:

Inter process communication provides a mechanism to allow processes to communicate and to synchronize their action. IPC is best, provided by a message system.

Basic structure:- The function of a message system is to allow processes to communicate with each other with out the need to alternative to shared variables. An IPC facility provides at least the two operations: **send and receive.** Message sent by a process can be of either fixed or variable size.

The methods for logical implementation of link and the send/ receive operations.

1) Direct or indirect communication.

2) Symmetric or asymmetric communication.

3) Synchronous or Asynchronous communication

4) Automatic or explicit buffering.

Direct communication:- Each process that wants to communicate must explicitly name the sender of the communication. In this scheme, the send and receive primitives are defined as follows.

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.

1) A link is associated with exactly two processes.

2) A link is established automatically between every pair of processes that wants to communicate. The processes need to know only each others identity to communicate.

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

4) The link may be unidirectional, but is usually bi-directional.

Indirect communication:- The messages are sent to and received from mailboxes. A process can communicate with some other process via a number of different mail boxes. Two processes can communicate only if the processes have a shared mail box.

Send (A, message): send a message to mail box A.

Receive (A, message): Receive a message from mail box A.

In this section a communication link has the following properties.

A link is established between a pair of processes only if they have a shared mail box.

1) A link may be associated with more than two processes.

2) Between each pair of communicating processes, there may be number of different links, each link corresponding to one mail box.

3) A link may be either unidirectional or bi-directional.

Now, suppose that processes p1, p2 and p3 all share mail box A. process p1 sends a message to A while p2 & p3 each execute a receive from 'A'. Which process will receive the message sent by p1? This can be restored in a variety of ways.

i) Allow a link to be associated with at most two processes.

ii) Allow at most one process at a time to execute a receive operation.

iii)Allow the system to select arbitrary which process will receive the message. The system may identify the receiver to the sender.

In **Symmetric communication** both the sender process and receiver process must name the other to communicate.

In **Asymmetric communication**, only the sender names the recipient; the recipient is not required to name the sender. In this, the sender and receiver primitives are 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.

<u>Synchronous or Asynchronous communication</u>

Message passing may be either blocking or non blocking also known as **synchronous** or **Asynchronous.**

Blocking Send: The sending process is blocked until the message is received by the receiving process or by mail box.

Nonblocking send: The sending process sends the message and resumes operation.

Blocking Receive: The receiver blocks until a message is available.

Nonblocking receive: The receiver retrieves either a valid message or a null.

<u>Buffering:</u>

A link has some capacity that the number of messages that can be store in it temporally. This can be viewed as a queue of messages attached to the link. Basically there are three ways that such a queue can be implemented.

Zero capacity :- The queue has maximum length 0, thus the link can't have any message waiting in it. In this case, the sender must wait until the recipient receives the message.

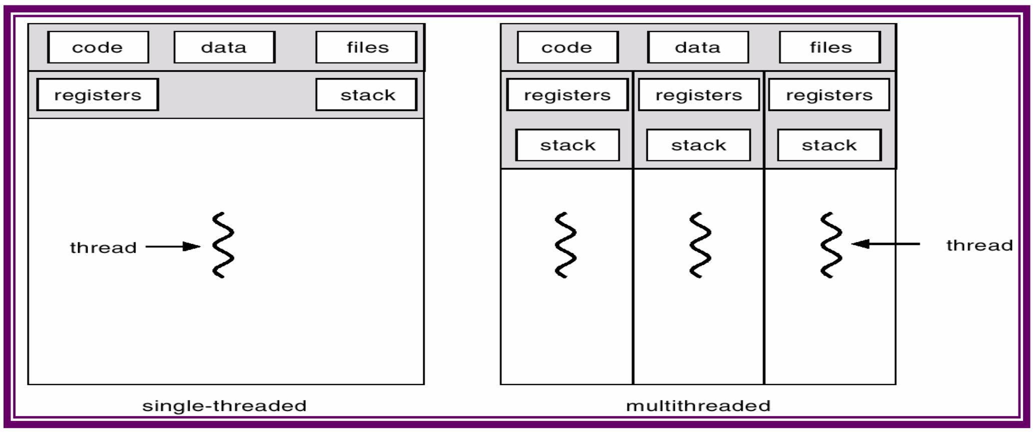
Bounded capacity :- The queue has finite length n, thus at most 'n' messages can in it. If the queue is not full, when a new message is sent, the latter is placed in the queue, and sender can continue execution with out waiting.

Un bounded Capacity:- Queue has potentially infinite. Thus any no of messages can wait in it. Suppose process P sends, a message to process Q and can continue its execution only after the message is received. Process P executes the sequence.

Zero capacity is known as Explicit buffering and the other two are known as Automatic buffering.

Multi Threaded Programming (Threads)

A Thread is a basic unit of CPU utilization; It comprises a thread ID, a program counter, a register set, and a stack. It shares with other threads belonging to the same process its code section, data section, and other OS resources such as open files and signals. A traditional (or heavy weight process) process has a single thread of control. If a process has multiple threads of control, it can perform more than one task at a time.



Thread Advantages:

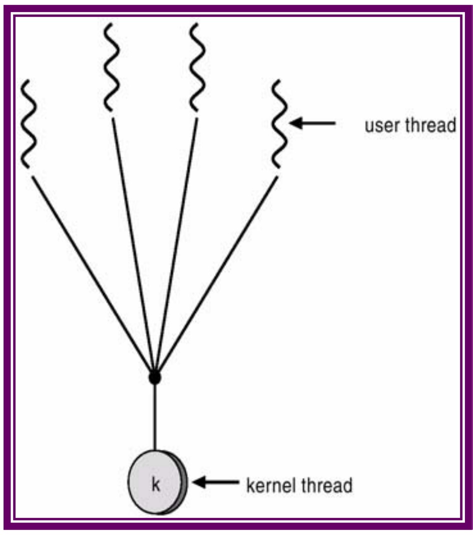
1) **Responsiveness:** Multithreading an interactive application may allow a program to continue running even if part of it is blocked or is performing a lengthy operation, there by increasing responsiveness to the user.
2) Resource sharing: The benefit of sharing code and data is that it allows an application to have several different threads of activity within the same address space.
3) Economy: Allocating memory and resources for process creation is costly. Because threads share resources of the process to which they belong, it is more economical to crate and context-switch threads.
4) **Utilization of multiprocessor architectures:** The benefits of multithreading can be greatly increased in a multiprocessor architecture, where threads may be running in parallel on different processors.

Multithreading Models:

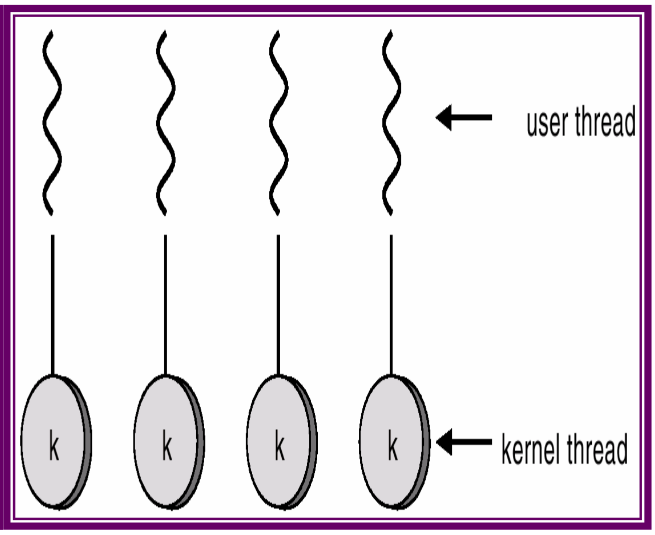
Support for threads may be provided at user level (**user threads**) or kernel level (**kernel threads).** User threads are supported above the kernel and are managed without kernel support, where as kernel threads are supported and are managed directly by the operating system. There are different ways to establish the relation ship between user and kernel level threads. They are…

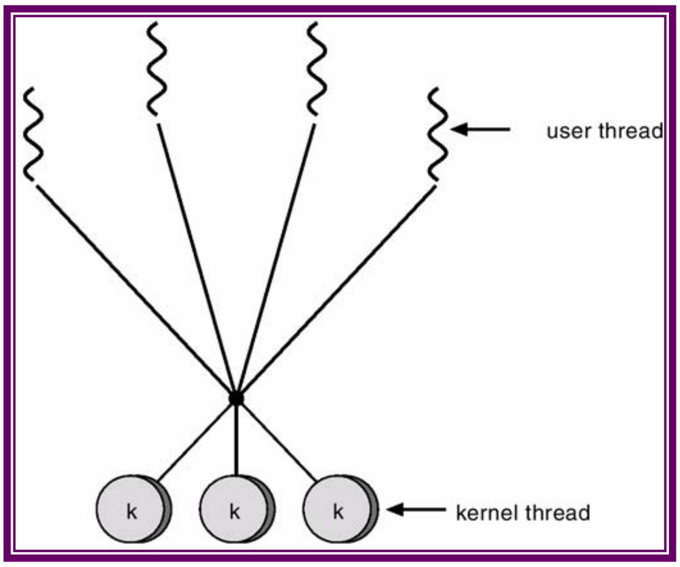
1) Many-to-One model: This model maps many user level threads to one kernel thread. Thread management is done by the thread library in user space, so it is efficient.

Many to One



One to One





Many To Many

2) One to One Model: This model maps each user thread to a kernel thread. It provides more concurrency than the many-to-many model by allowing another thread to run when a thread makes a blocking system call. It also allows multiple threads to run in parallel on multiple processors. Disadvantage is to create kernel thread each time when user thread is created.

3) Many-to-Many Model: This model multiplexes many user-level threads to a smaller or equal number of kernel threads. Number of kernel threads may be specific to application and/or machine. When a thread performs a blocking system call, the kernel can schedule another thread for execution.

Thread Issues:

1) The fork( ) and exec( ) system call: If one thread in a program calls fork( ) system call then one of the two things may happen. In one version, all threads will be duplicated and in another version only the thread that invoked fork( ) system call will only be duplicated. If a thread invokes exec( ) system call, the program specified in the parameter to exec( ) will replace the entire process including all threads.
2) Thread Cancellation: It is the task of terminating a thread before its task is completed. A thread that is to be cancelled is known as **target thread.**
If one thread immediately terminates the target thread, it is known as **Asynchronous cancellation.** If target thread periodically checks whether it should terminate, then it is known as **Deferred cancellation.**

3) Signal Handling: A signal is used to notify a process that a particular event has occurred. A signal may be received either synchronously or asynchronously. Examples of synchronous signal include illegal memory access and division by zero. Synchronous signals are delivered to the same process that performed the operation that caused the signal. When a signal is generated by an event external to a running process, that process receives the signal asynchronously. Every signal may be handled by one of two possible handlers. i) A default signal handler. ii) A user-defined signal handler.

4) Thread Pools: A thread pool is a collection of threads created at startup. In pool, threads will wait for work. When a server receives a request, it awakens a thread from the pool (if one available) and passes it the request to service. Once the thread completes its service, it returns to the pool and awaits more work. If the contains no available thread, the server waits until one becomes free.

5 Thread-Specific Data : Threads belonging to a process share the data of the process. Indeed, this sharing of data provides one of the benefits of multithreaded programming. However, in some circumstances, each thread might need its own copy of certain data. Such data is called **Thread-specific data.**

******

5. CPU SCHEDULING

CPU scheduling is the basis of multi programmed operating system. Scheduling is a fundamental operating system function.

CPU Scheduler (short term scheduler):-

When ever the CPU becomes idle, the O.S must select one of the processes in the ready queue to be executed. The selection process is carried out by the short-term scheduler. The scheduler selects one among the processes in memory that are ready to execute and, allocate the CPU to one of them.

Ready queue is not necessarily a FIFO queue, considering various scheduling algorithms, a ready queue may be implemented as a FIFO queue, a priority queue, a tree or simply an unordered linked list.

Preemptive scheduling:- CPU scheduling decisions may take place under the following 4 conditions.

1) When a process switches from running state to waiting state.

2) When a process switches from running state to ready state.

3) When a process switches from waiting state to ready state.

4) When a process terminates.

When scheduling takes only under 1 and 4. Then such scheduling is known as ***non preemptive scheduling***. Scheduling takes place under 2 & 3 is known as ***preemptive scheduling.***

In non preemptive scheduling once the CPU is allocated to a process, the process will never release CPU until its task is over or need to wait for an I/O.

Another component involved in the CPU scheduling function is the **DISPATCHER.** It is the module that gives the control of the CPU to the process selected by the short term scheduler.

Scheduling Criteria:-

In choosing which algorithm to use in a particular situation, we must consider the property of the various algorithms depends up on the criteria.

1) CPU Utilization:- We want to keep the CPU busy as possible. CPU utilization may range from 0 to 100 percent. In real systems it should range from 40% to 90%.
2) Through put: - The number of processes that are completed in unit time is called through put.
3) Turn around time: - The interval from the time of submission to the time of completion is known as TURN around time.
4) Waiting time: - Waiting time is the sum of the periods spent waiting in the ready queue. The time waited for I/O is not considered as waiting time
5) Response time:- The time from the submission of a request until the first response is produced is known as Response Time.

Scheduling algorithms:-

CPU scheduling deals with the problem of deciding which of the processes in the ready queue is to allocate the CPU. There are many CPU algorithms.
1) FCFS (first come first serve)
2) SJF (shortest job first)
3) Priority Scheduling algorithm
4) Round Robin Scheduling Algorithm.

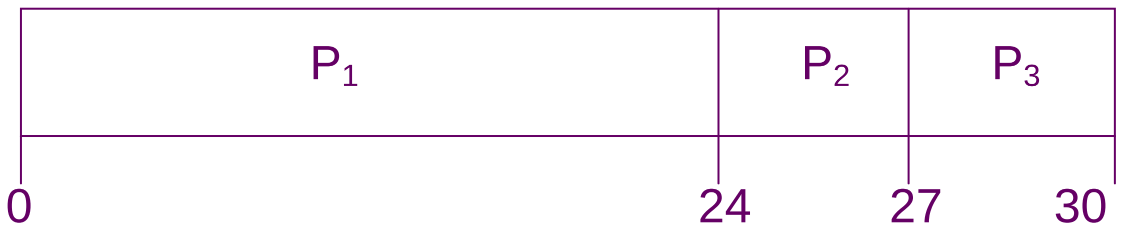
1. FCFS:-

The simplest algorithm is FCFS. Here the CPU is allocated as the process request. i.e the process that request the CPU first is allocated the CPU first. The implementation of the FCFS policy is easily managed with FIFO queue.

When a process enters the ready queue its PCB is linked on to the tail of the queue. When CPU is free it is allocated to the process at the head of the queue. The running process is then removed from the queue. The code for the FCFS is also very simple and easy to under stand but the average waiting time and average turn-around time is high.

EX: - processes	burst time
P1	24
P2	3
P3	3

Gantt Chart:-



The waiting time for the P1 is 0
The waiting time for the P2 is 24
The waiting time for the P3 is 27
The *average waiting* time = (0+24+27)/3 = 17 milliseconds.

The turn around time for P1 = 24
The turn around time for P2 = 27
The turn around time for P3 = 30
The *average turnaround* time = (24+27+30)/3= 27 milliseconds.
NOTE:- once the CPU has been allocated to the process that process keeps, the CPU until it release the CPU .
Convoy effect:- when one big process is executing by the CPU all other processes will be in waiting state. This is known as convoy effect.
FCFS algorithm is non preemptive.

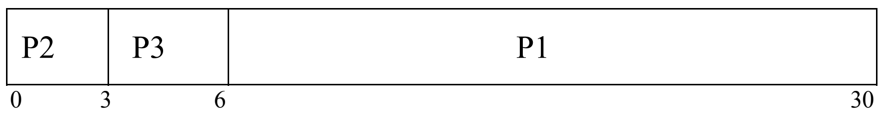
Shortest Job First scheduling:-

In this scheduling algorithm the *shortest job is executed first and then the next job and so on….* SJF is the optimal (best) of all the algorithms.
If two processes have the same length FCFS algorithm is used to break the tie.

Ex:-

process	**cpu burst time**
P1	24
P2	3
P3	3

GANTT chart:-



Waiting time for P1= 6
Waiting time for the P2=0
Waiting time for P3= 3
Average waiting time = (6+0+3)/3=9/3=3milliseconds.

Turn around time of P1= 30
Turn around time of P2= 3
Turn around time of P3= 6
Average turn around time= (30+3+6)/3= 13 milliseconds.

Advantages:-

1) The average waiting time is minimum.
2) This algorithm is optimal.

Disadvantages:-

1) To predict the length of the next CPU burst.

It may be either preemptive or non preemptive. This algorithm is just a special case of priority algorithm. The preemptive SJF scheduling algorithm is some times **called Shortest- remaining time first.**

SRTF (Shortest reaming time first):-

In this scheduling algorithm all the processes may not come at the same time to ready queue, scheduling the processes as they arrive, using the concept of SJF is known as shortest remaining time first.
This comes under the preemptive algorithm.

Ex:-

process	**arrival time**	**cpu burst time**
P1	0	8
P2	1	4
P3	2	9
P4	3	5

Gantt chart:-

The average waiting time = ((10-1)+(1-1)+(17-2)+(5-3))/4=26/4=6.5milliseconds.

Priority scheduling algorithm:-

A priority is associated with each process, and the CPU is allocated to the process with the highest priority. Equal –priority processes are scheduled in FCFS order.

As an example, consider the following set of processes, assumed to have arrived at time 0, in the order p1,p2, p3,p4,p5 with the length of the CPU burst time given in milliseconds.

Processes	**Burst time**	**Priority**
P1	10	3
P2	1	1
P3	2	3
P4	1	4
P5	5	2

Gantt chart:

Priorities can be defined either internally or externally. Priority scheduling can be either preemptive or non preemptive. When a process arrives at the ready queue, its priority is compared with the priority of the currently running process. A preemptive priority scheduling algorithm will preempt the CPU if the priority of the newly arrived process is higher than the priority of the currently running process. A non preemptive scheduling algorithm will simply put the new process at the head of the ready queue.

A major problem with priority scheduling algorithm is *indefinite blocking or starvation.* A process that is ready to run but lacking the CPU can be considers blocked, waiting for CPU. Here always the higher priority processes are executed and lower priorities processes are wait long time for CPU.

A solution to the problem of *indefinite blockage* of lower priority processes is **Aging.** Aging is technique of gradually increasing the priority of processes that wait in the system for a long time.

Round- Robin Scheduling:-

It is especially for the time sharing systems .It is similar to the FCFS scheduling, but preemption is added to switch between processes. A small unit of time, called *time quantum or time slice*. The ready queue is treated as a circular queue. The CPU scheduler goes around the ready queue allocating the CPU to each process for a time interval up to 1 time quantum.

To implement the RR scheduling, we keep the ready queue as a FIFO queue of processes. New processes are added to the tail of the ready queue. The CPU scheduler picks the first process from the ready queue, sets a timer to interrupt after 1 time quantum, and dispatches the process.

Consider the following set of processes that arrive at time 0, with the length of the CPU bursts time gives in milliseconds.

Process	**burst time**
P1	24
P2	3
P3	3

Here the time quantum is 4 milliseconds.

GANTT Chart:-

The waiting time for P1=(10-4) milliseconds
The waiting time for the P2= 4 milliseconds
The waiting time for P3= 7 milliseconds
The average waiting time is= 17/3=5.66 milliseconds

If a process CPU burst exceeds time quantum, that process is preempted and is put back in the ready queue.

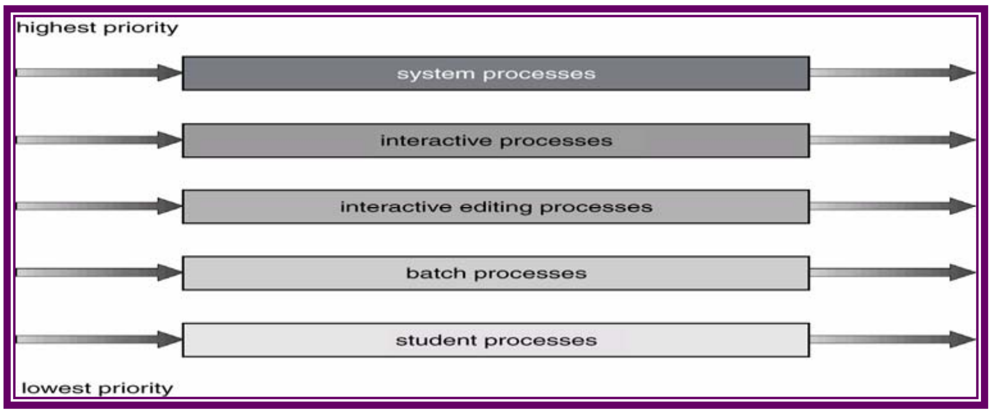
{If we use a time quantum of 4 milliseconds, then process p1 gets the first 4 milliseconds. Since it require another 20millisecons. It is preempted after the first time quantum, and the CPU is given to the next process in the queue, process p2, since the process p2 does not need 4 m.s, it quits before its time quantum expires, the CPU is then given to the next process, process p3. Once each process has received 1 time quantum, the CPU returns to processp1 for additional time quantum.}

The performance of the RR algorithm depends heavily on the size of the time quantum. At one execute, if the time quantum is very large, the RR policy is the same as the FCFS policy. If the quantum is very small, the RR approach is called processor sharing.

Multilevel queue scheduling:-

Another class of scheduling algorithm has been created for the processes. Which are easily classified in to different groups? A common division is made between foreground processes and background processes. These two types have different response time and have different scheduling needs. In the diagram, The first three kinds of processes comes under fore ground processes and last two comes under back ground processes.

A multilevel queue scheduling algorithm partitions the ready queue in to several separate queues. The processes are permanently assigned to one queue based on the some property of the processes; each queue has its own scheduling algorithm. For example, separate queues might be used for foreground and back ground processes. The foreground queue might be scheduled by an RR algorithm. Which background queue is scheduled by an FCFS algorithm?



Multilevel Feed back Queue Scheduling:-

In a multilevel queue scheduling algorithm, processes are permanently assigned to a queue on entry to the system. Processes do not move between the queues. This setup has the advantage of low scheduling, but is inflexible.

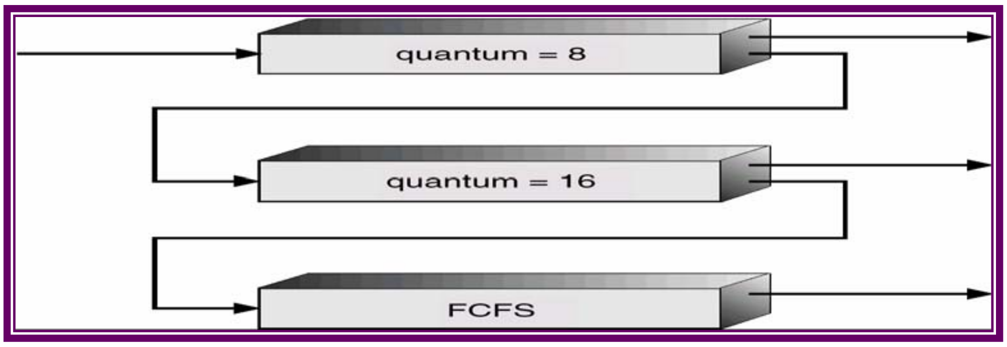
Multilevel feed back queue scheduling, however allows a process to move between queues. If a process uses too much CPU time, it will be moved to a lower-priority queue. Similarly, a process that waits too long in a lower priority queue may be moved to a higher priority queue. This form of aging prevents starvation.

Ex:- consider a multilevel feed back queue scheduler with three queues, numbered from 0 to 2. The scheduler first executes all processes in queue 0. Only when queue 0 is empty will it execute processes in queue 1. Similarly, process that arrives for queue 1 will only be executed if queue 0 is empty. A process that arrives for queue 1 will preempt a process in queue 2.A process in queue 1 will in turn be preempted by a process arriving for queue 0.

A process entering the ready queue is put in queue 0. A process in queue 0 is given a time quantum of 8m.s. if it does not finish with in this time; it is moved to the tail of queue of 1. If queue 0 is empty, the process at the head of queue 1 is given a quantum of 16 m.s. If it does not complete it is preempted and is put in to queue 2. Processes in queue 2 are run on an FCFS basis, only when queues 0 and 1 are empty.

The multilevel feed back queue scheduler is defined by the following parameters.

1) The number of queues.
2) The scheduling algorithm for each queue.
3) The method used to determine when to upgrade a process to a higher priority queue.
4) The method used to determine when to demote a process to a lower priority queue.
5) The method used to determine which queue a process will enter when that process needs services.



Multiple – processor Scheduling:-

If multiple CPUs are available, the scheduling problem is corresponding more complex. Where, the processors are identical in terms of their functionality. Any available processor can then be used to run any processor in the queue.

If the several identical processors are available, then load sharing can occur. It would be possible to provide a separate queue for each processor. In this case, how ever one processor could be ideal, with an empty queue, while another processor was very busy. To prevent this situation we use a common ready queue. All processes go in to one queue and are scheduled on to any available processor.

Real – time –scheduling:-

Real time computing is divided in to two types. Hard-real-time systems are generally required to complete a critical task with in a guaranteed amount of time.

Soft real time computing is less restrictive. Implementing soft real time functionality requires careful design of the scheduler and related aspects of the O.S. first the system must have priority scheduling, and real time processes must have the highest priority.

The problem is that many O.S are forced to wait for either a system call to complete or for an I/O block to take place before doing a context switch. The dispatch latency in such systems can be long, since some system calls are complex and some I/O devices are slow.

To keep the dispatcher latency low, we need to allow system calls to be perceptible. There are several ways to achieve this goal. One is to insert preemptive points in long duration system calls, which checks to see whether a high priority process needs to be run. Preemption points can be placed at only “safe” location in the kernel. Where, kernel data structure is not being modified. Another method for dealing with preemptive is to make the entire kernel preempted.

******

PROCESS SYNCHRONIZATION

A cooperating process is one that can affect or be affected by the other processes executing in the system. Cooperating processes may either directly share a logical address space (that is, both code and data), or be allowed to share data only through files.

1 Background

Let us return to the shared-memory solution to the bounded-buffer problem solution allows at most n - 1 items in the buffer at the same time. One possibility is to add an integer variable counter, initialized to 0. Counter is incremented every time we add a new item to the buffer, and is decremented every time we remove one item from the buffer. The code for the producer process can be modified as follows:

```
repeat
produce an item in nextp
while counter = n do no-op;
buffer[in] := nextp;
in := in + 1 mod n;
counter := counter + 1;
until false;
```

The code for the consumer process can be modified as follows:

```
repeat
while counter = 0 do no-op;
nextc:-buffer[out];
out := out + i mod n;
counter := counter — 1;
consume the item in nextc
until false;
```

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 the producer and consumer processes execute the statements "counter := counter + 1" and "counter := counter - 1" concurrently. Following the execution of these two statements, the value of the variable counter may be 4, 5, or 6. The only correct result is counter = 5, which is generated correctly if the producer and consumer execute separately.

The statement "counter := counter + 1" may be implemented in machine language is:

```
Register1 := counter;
Register1 := register1 + 1;
counter := register1
```

Where register1 is a local CPU register. Similarly, the statement "counter := counter - 1" is implemented as follows:

```
Register2 := counter
Register2 := register2 - 1;
Counter := register2
```

The concurrent execution of the statements "counter := counter + 1" and "counter := counter - 1" is equivalent to a sequential execution where the lower level statements presented previously are interleaved in some arbitrary order. One such interleaving is:

T0: producer execute	register1= counter	{register1 = 5}
T1: producer execute	register1= register1 + 1	{register1 = 6}
T2: consumer execute	register2= counter	{register2 = 5}
T3: consumer execute	register2= register2 - 1	{register2 = 4}
T4: producer execute	counter= register1	{counter = 6}
TS: consumer execute	counter= register2	{counter = 4}

We would arrive at this incorrect state because we allowed both processes to manipulate the variable counter concurrently. A situation like this, where several processes access and manipulate the same data concurrently, and the outcome of the execution depends on the particular order in which the access takes place, is called a race condition. To guard against the race condition above, we need to ensure that only one process at a time can be manipulating the variable counter. To make such a guarantee, we require some form of synchronization of the processes.

2 The Critical-Section Problem

Consider a system consisting of n processes {P0, P1,..., Pn-1}. Each process has a segment of code, called a critical section, in which the process may be changing common variables, updating a table, writing a file, and so on. The important feature of the system is that, when one process is executing in its critical section, no other process is to be allowed to execute in its critical section. Thus, the execution of critical sections by the processes is mutually exclusive in time. The critical-section problem is to design a protocol that the processes can use to cooperate.

General structure of a typical process Pi:

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

Each process must request permission to enter its critical section. The section of code implementing this request is the entry section. The critical section may be followed by an exit section. The remaining code is the remainder section. A solution to the critical-section problem must satisfy the following three requirements:

- **Mutual Exclusion:** If process Pi is executing in its critical section, then no other processes can be executing in their critical sections.
- **Progress:** If no process is executing in its critical section and there exist some processes that wish to enter their critical sections, then only those processes that are not executing in their remainder section can participate in the decision of which will enter its critical section next, and this selection cannot be postponed indefinitely.
- **Bounded Waiting:** There exist a bound on the number of times that other processes are allowed to enter their critical sections after a process has made a request to enter its critical section and before that request is granted.

Peterson's Solution:

By combining the key ideas of algorithm 1 and algorithm 2, we obtain a correct solution to the critical-section problem, where all three requirements are met. The processes share two variables:

```
var flag: array [0..1] of boolean;
turn: 0..1;
```

Initially flag[0] =flag[l] = false, and the value of turn is immaterial (but is either 0 or 1).

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

Process Pi first sets flag[i] to be true, and then asserts that it is the other process' turn to enter if appropriate (turn = j). If both processes try to enter at the same time, turn will be set to both i and j; at roughly the same time. Only one of these assignments will last; the other will occur, but will be overwritten immediately. The eventual value of turn decides which of the two processes is allowed to enter its critical section first. We now prove that this solution is correct. We need to show that:

1. Mutual exclusion is preserved,
2. The progress requirement is satisfied,
3. The bounded-waiting requirement is met.

To prove property 1, we note that each Pj enters its critical section only if either flag[j] = false or turn = i Also note that, if both processes can be executing in their critical sections at the same time, then flag[0]=flag[1]=true. These two observations imply that P0 and PI could not have executed successfully their while statements at about the same time, since the value of turn can be either 0 or 1, but cannot be both. Hence, one of the processes say Pj—must have executed successfully the while statement, whereas Pf had to execute at least one additional statement ("turn = j"). However, since, at that time, flag[j] = true, and turn = i, and this condition will persist as long as Pj is in its critical section, the result follows: Mutual exclusion is preserved.

To prove properties 2 and 3, we note that a process Pi, can be prevented from entering the critical section only if it is stuck in the while loop with the condition flag[j] = true and turn = j; this loop is the only one. If Pj is not ready to enter the critical section, then flag[j] = false, and Pj can enter its critical section. If Pj has set flag[j] - true and is also executing in its while statement, then either turn = i or turn = j. If turn = i, then Pf will enter the critical section. If turn = j, then Pj will enter the critical section. However, once Pj exits its critical section, it will reset flag[j] to false, allowing P, to enter its critical section. If Pj resets flag[j] to true, it must also set turn = i. Thus, since Pj does not change the value of the variable turn while executing the while statement, Pi will enter the critical section (progress) after at most one entry by Pj.

3 Synchronization Hardware

The critical-section problem could be solved simply in a uni-processor environment if we could disallow interrupts to occur while a shared variable is being modified. In this manner, we could be sure that the current sequence of instructions would be allowed to execute in order without preemption. No other instructions would be run, so no unexpected modifications could be made to the shared variable.
Unfortunately, this solution is not feasible in a multiprocessor environment.

Disabling interrupts on a multiprocessor can be time-consuming, as the message is passed to all the processors. This message passing delays entry into each critical section, and system efficiency decreases. Many machines therefore provide special hardware instructions that allow us either to test and modify the content of a word, or to swap the contents of two words, atomically. We can use these special instructions to solve the critical section problem in a relatively simple manner.

The important characteristic is that this instruction is executed atomically that is, as one uninterruptible unit. Thus, if two Test-and-Set instructions are executed simultaneously (each on a different CPU), they will be executed sequentially in some arbitrary order.
The definition of the Test-and-Set instruction:

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

If the machine supports the Test-and-Set instruction, then we can implement mutual exclusion by declaring a Boolean variable lock, initialized to false.
Mutual-exclusion implementation with Test-and-Set:

```
do {
            while (TestAndSet(&lock)) ; // do no-op
                critical section
            lock = false;
                remainder section
  }while(TRUE)
```

The common data structures are

```
    var waiting: array [0..n — 1] of boolean;
    lock: boolean;
```

These data structures are initialized to false.

The definition of the Swap instruction:

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

Mutual-exclusion implementation with the Swap instruction:

```
do {
            key = true;
            while (key == true)
                    Swap(lock,key);
                critical section
            lock = false;
                remainder section
   } while(TRUE)
```

Bounded-waiting mutual exclusion with Test-and-Set:

```
    var j:=0..n -1;
    key: boolean;

    do{
        waiting[i] = true;
        key = TRUE;
        while (waiting[i] && key)
            key = Test-and-Set(&lock);
        waiting[i] = FALSE;
                // critical section
        j = (i+1) % n;
        while ((j!=i) && (! waiting[j])
            j = (j+1)%n;
        if (j == i)
            lock = FALSE
        else
            waiting[j] = FALSE;
                // remainder section
    }While (TRUE);
```

4 Semaphores

We can use a synchronization tool, called a semaphore. A semaphore S is an integer variable that, apart from initialization, is accessed only through two standard atomic operations: wait and signal. These operations were originally termed P (for wait; from the Dutch proberen, to test) and V (for signal; from verhogen, to increment). The classical definitions of wait and signal are:

```
 wait(S):      while S < 0 do no-op;
               S := S - 1;
signal(S):     S := S + 1;
```

When one process modifies the semaphore value, no other process can simultaneously modify that same semaphore value. In addition, in the case of the wait(S), the testing of the integer value of S (S < 0), and its possible modification (S: = S - 1), must also be executed without interruption.

4.1 Usage

We can use semaphores to deal with the w-process critical-section problem. The n processes share a semaphore, mutex (standing for mutual exclusion), initialized to 1. Mutual-exclusion implementation with semaphores:

```
do{
    wait(mutex);
            critical section
    signal(mutex);
            remainder-section
} while (1);
```

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

```
S1;
signal(synch);
```

In process P1, and the statements:

```
wait(synch);
S2;
```

In process P2. Because synch is initialized to 0, P2 will execute S2 only after P1 has invoked signal(synch), which is after S1.

4.2 Implementation

The main disadvantage of the mutual-exclusion solutions of the semaphore definition is that they all require busy waiting. While a process is in its critical section, any other process that tries to enter its critical section must loop continuously in the entry code. This continual looping is clearly a problem in a real multi programming system, where a single CPU is shared among many processes. Busy waiting wastes CPU cycles that some other process might be able to use productively. This type of semaphore is also called a spinlock

Spinlocks are useful in multiprocessor systems. The advantage of a spinlock is that no context switch is required when a process must wait on a lock, and a context switch may take considerable time. Thus, when locks are expected to be held for short times, spinlocks are useful. When a process executes the wait operation and finds that the semaphore value is not positive, it must wait. However, rather than busy waiting, the process can block itself. .The block operation places a process into a waiting queue associated with the semaphore, and the state of the process is switched to the waiting state. Then, control is transferred to the CPU scheduler, which selects another process to execute.

A process that is blocked, waiting on a semaphore S, should be restarted when some other process executes a signal operation. The process is restarted by a wakeup operation, which changes the process from the waiting state to the ready state. The process is then placed in the ready queue. To implement semaphores under this definition, we define a semaphore as a record:

```
type semaphore = record
value: integer;
L: list of process;
end;
```

Each semaphore has an integer value and a list of processes. When a process must wait on a semaphore, it is added to the list of processes. A signal operation removes one process from the list of waiting processes, and awakens that process:

The semaphore operations can now be defined as

```
wait(S):
    S.value--;
    if (S.value < 0) {
        add-this-process-to-S.L;
        block( );
    }
signal(S):
    S.value++;
    if (S.value <= 0) {
        remove-a-process-P-from-S.L;
        wakeup(P); }
```

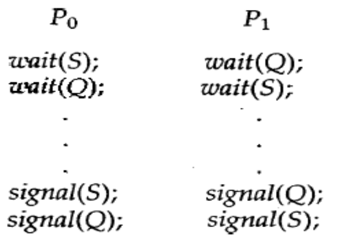
The block operation suspends the process that invokes it. The wakeup(P) operation resumes the execution of a blocked process P. These two operations are provided by the operating system as basic system calls. The list of waiting processes can be easily implemented by a link field in each process control block (PCB). Each semaphore contains an integer value and a pointer to a list of PCBs. One way to add and remove processes from the list, which ensures bounded waiting, would be to use a first-in, first-out (FIFO) queue, where the semaphore contains both head and tail pointers to the queue.

The critical aspect of semaphores is that they are executed atomically. We must guarantee that no two processes can execute wait and signal operations on the same semaphore at the same time. This situation is a critical-section problem, and can be solved in either of two ways:

1. In a uni-processor environment (that is, where only one CPU exists), we can simply inhibit interrupts during the time the wait and signal operations are executing. This scheme works in a uni-processor environment because, once interrupts are inhibited, instructions from different processes cannot be interleaved. Only the currently running process executes, until interrupts are re enabled and the scheduler can regain control.
2. In a multiprocessor environment, inhibiting interrupts does not work. Instructions from different processes (running on different processors) may be interleaved in some arbitrary way. If the hardware does not provide any special instructions, we can employ any of the correct software solutions for the critical section problem, where the critical sections consist of the wait and signal procedures.

4.3 Deadlocks and Starvation

The implementation of a semaphore with a waiting queue may result in a situation where two or more processes are waiting indefinitely for an event that can be caused by only one of the waiting processes. The event in question is the execution of a signal operation. When such a state is reached, these processes are said to be deadlocked. To illustrate this, we consider a system consisting of two processes, P0 and P1, each accessing two semaphores, S and Q, set to the value 1:



```
      P0               P1

   wait(S);         wait(Q);
   wait(Q);         wait(S);
      .                .
      .                .
      .                .
   signal(S);       signal(Q);
   signal(Q);       signal(S);
```

Suppose that P0 executes wait(S), and then PI executes wait(Q). When P0 executes wait(Q), it must wait until PI executes signal(Q). Similarly, when P1 executes wait(S), it must wait until P0 executes signal(S). Since these signal operations cannot be executed, P0 and PI are deadlocked. Another problem related to deadlocks is indefinite blocking or starvation, a situation where processes wait indefinitely within the semaphore. Indefinite blocking may occur if we add and remove processes from the list associated with a semaphore in LIFO order.

4.4 Binary Semaphores

A binary semaphore is a semaphore with an integer value that can range only between 0 and 1. A binary semaphore can be simpler to implement than a counting semaphore, depending on the underlying hardware architecture. We will now show how a counting semaphore can be implemented using binary semaphores. Let S be a counting semaphore. To implement it in terms of binary semaphores we need the following data structures:

```
var S1: binary-semaphore;
    S2: binary-semaphore;
    C: integer;
```

Initially S1 = 1, S2 = 0, and the value of integer C is set to the initial value of the counting semaphore S. The wait operation on the counting semaphore S can be implemented as follows:

```
wait(S1);
C--;
if (C < 0) {
            signal(S1);
            wait(S2);
}
signal(S1);
```

The signal operation on the counting semaphore S can be implemented as follows:

```
wait(S1);
C--;
if (C < 0) {
            signal(S1);
            wait(S2);
}
signal(S1);
```

5 Classical Problems of Synchronization
5.1 The Bounded-Buffer Problem

We assume that the pool consists of n buffers, each capable of holding one item. The mutex semaphore provides mutual exclusion for accesses to the buffer pool and is initialized to the value 1. The empty and full semaphores count the number of empty and full buffers, respectively. The semaphore empty is initialized to the value n; the semaphore full is initialized to the value 0.

The structure of the producer process:

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

The structure of the consumer process:

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

5.2 The Readers and Writers Problem

A data object (such as a file or record) is to be shared among several concurrent processes. Some of these processes may want only to read the content of the shared object, whereas others may want to update (that is, to read and write) the shared object. We distinguish between these two types of processes by referring to those processes that are interested in only reading as readers, and to the rest as writers. Obviously, if two readers access the shared data object simultaneously, no adverse effects will result. However, if a writer and some other process (either a reader or a writer) access the shared object simultaneously, chaos may ensue. To ensure that these difficulties do not arise, we require that the writers have exclusive access to the shared object. This synchronization problem is referred to as the readers-writers problem. The readers-writers problem has several variations, all involving priorities:

1. The simplest one, referred to as the first readers-writers problem, requires that no reader will be kept waiting unless a writer has already obtained permission to use the shared object. In other words, no reader should wait for other readers to finish simply because a writer is waiting. The second readers-writers problem requires that, once a writer is ready, that writer performs its write as soon as possible.

2. In other words, if a writer is waiting to access the object, no new readers may start reading.

In the solution to the first readers-writers problem, the reader processes share the following data structures:

```
var mutex, wrt: semaphore;
readcount: integer;
```

The semaphores *mutex* and *wrt* are initialized to 1; *readcount* is initialized to 0. The semaphore *wrt* is common to both the reader and writer processes. The *mutex* semaphore is used to ensure mutual exclusion when, the variable *readcount* is updated. The *readcount* variable keeps .track of how many processes are currently reading the object. The semaphore wrt functions as a mutual exclusion semaphore for the writers. It also is used by the first or last reader that enters or exits the critical section. It is not used by readers who enter or exit while other readers are in their critical sections.

The structure of a writer process:

```
wait(wrt);
writing is performed
signal(wrt);
```

The structure of a reader process:

```
do{
        wait(mutex);
        readcount++;
        if (readcount == 1)
                wait(rt);
        signal(mutex);
                …
            reading is performed
                …
        wait(mutex);
        readcount--;
        if (readcount == 0)
            signal(wrt);
        signal(mutex);
}while(TRUE);
```

If a writer is in the critical section and n readers are waiting, then one reader is queued on wrt, and n - 1 readers are 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 waiting writer.

5.3 The Dining-Philosophers Problem

Consider five philosophers who spend their lives thinking and eating. The philosophers share a common circular table surrounded by five chairs, each belonging to one philosopher. In the center of the table there is a bowl of rice, and the table is laid with five single chopsticks. When a philosopher thinks, she does not interact with her colleagues. From time to time, a philosopher gets hungry and tries to pick up the two chopsticks that are closest to her. A philosopher may pick up only one chopstick at a time. Obviously, she cannot pick up a chopstick that is already in the hand of a neighbor. When a hungry philosopher has both her chopsticks at the same time, she eats without releasing her chopsticks. When she is finished eating, she puts down both of her chopsticks and starts thinking again.

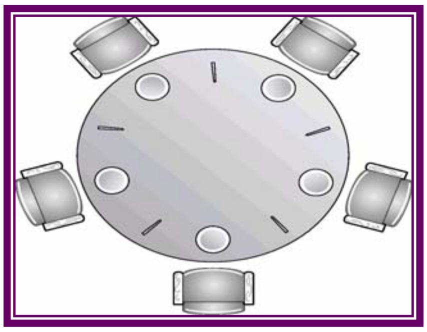
One simple solution is to represent each chopstick by a semaphore. A philosopher tries to grab the chopstick by executing a wait operation on that semaphore; she releases her chopsticks by executing the signal operation on the appropriate semaphores. Thus, the shared data are:

```
        var chopstick: array [0..4] of semaphore;
```

Where all the elements of chopstick are initialized to 1.

The structure of philosopher i:

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



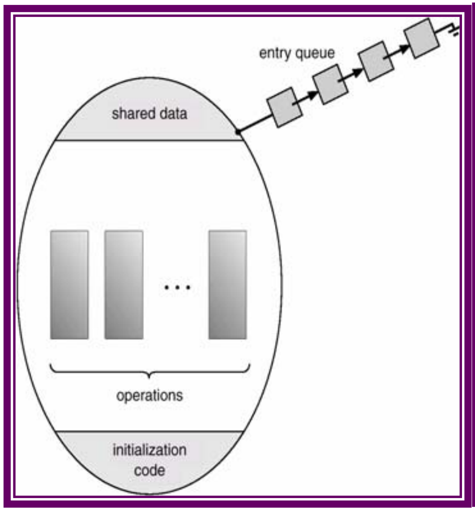
Suppose that all five philosophers become hungry simultaneously, and each grabs her left chopstick. All the elements of chopstick will now be equal to 0. When each philosopher tries to grab her right chopstick, she will be delayed forever. We present a solution to the dining-philosophers problem that ensures freedom from deadlocks.

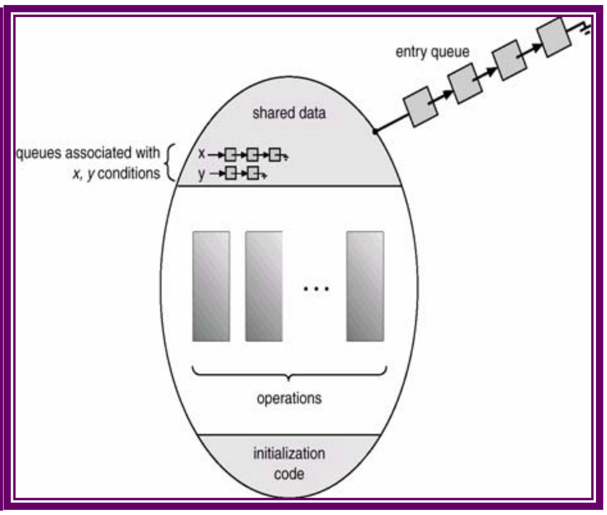
- Allow at most four philosophers to be sitting simultaneously at the table.
- Allow a philosopher to pick up her chopsticks only if both chopsticks are available.
- Use an asymmetric solution; that is, an odd philosopher picks up first her left chopstick and then her right chopstick, whereas an even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

6 Monitors

Another high-level synchronization construct is the monitor type. A monitor is characterized by a set of programmer-defined operators. The representation of a monitor type consists of declarations of variables whose values define the state of an instance of the type, as well as the bodies of procedures or functions that implement operations on the type. The syntax of a monitor is:

```
monitor monitor-name
        {
            shared variable declarations
            procedure body P1 (…) {
                . . .
            }
            procedure body P2 (…) {
                . . .
            }
            procedure body Pn (…) {
                . . .
            }
            {
                initialization code
            }
        }
```





The monitor construct ensures that only one process at a time can be active within the monitor. Consequently, the programmer does not need to code this synchronization constraint explicitly we need to define additional synchronization mechanisms. These mechanisms are provided by the condition construct. A programmer who needs to write her own tailor-made synchronization scheme can define one or more variables of type condition:

var x,y: Condition;

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

x.wait;

Means that the process invoking this operation is suspended until another process invokes

x.signal;

The x.signal operation resumes exactly one suspended process. If no process is suspended, then the signal operation has no effect; that is, the state of x is as though the operation was never executed. Now suppose that, when the x.signal operation is invoked by a process P, there is a suspended process Q associated with condition x. Clearly, if the suspended process Q is allowed to resume its execution, the signaling process P must wait. Otherwise, both P and Q will be active simultaneously within the monitor. Note, however, that both processes can conceptually continue with their execution. Two possibilities exist:

1. P either waits until Q leaves the monitor, or waits for another condition.
2. Q either waits until P leaves the monitor, or waits for another condition.

A deadlock-free solution to the dining-philosophers problem:

Recall that a philosopher is allowed to pick up her chopsticks only if both of them are available. To code this solution, we need to distinguish between three states in which a philosopher may be. For this purpose, we introduce the following data structure:

var state:array[0..4] of (thinking, hungry, eating);

Philosopher i can set the variable state[i] = eating only if her two neighbors are not eating (state[i+4 mod 5] != eating and state[i+1 mod 5] != eating). We also need to declare:

var self: array [0..4] of condition

Where philosopher i can delay herself when she is hungry, but is unable to obtain the chopsticks she needs. We are now in a position to describe our solution to the dining-philosopher problem. The distribution of the chopsticks is controlled by the monitor dp, which is an instance of the monitor type dining-philosophers.

A monitor solution to the dining-philosopher problem:

```
monitor dp
        {
                enum {thinking, hungry, eating} state[5];
                condition self[5];
                void pickup(int i)              // following slides
                void putdown(int i)   // following slides
                void test(int i)                // following slides
                void init() {
                        for (int i = 0; i < 5; i++)
                                state[i] = thinking;
                }
        }
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();
                }
        }
```

Each philosopher, before starting to eat, must invoke the operation pickup. This may result in the suspension of the philosopher process. After the successful completion of the operation, the philosopher may eat. Following this, the philosopher invokes the putdown operation, and may start to think. Thus, philosopher i must invoke the operations pickup and putdown in the following sequence:

dp.pickup(i);

...

eat
...
dp.putdown(i);

7 Atomic Transactions

The mutual exclusion of critical sections ensures that the critical sections are executed atomically. That is, if two critical sections are executed concurrently, the result is equivalent to their sequential execution in some unknown order.

7.1 System Model

A collection of instructions (operations) that performs a single logical function is called a transaction. A major issue in processing transactions is the preservation of atomicity despite the possibility of failures within the computer system. A transaction is a program unit that accesses and possibly updates various data items that may reside on the disk within some files. From our point of view, transactions are simply a sequence of read and write operations, terminated by either a commit operation or an abort operation. A commit operation signifies that the transaction has terminated its execution successfully, whereas an abort operation signifies that the transaction had to cease its normal execution due to some logical error. A terminated transaction that has completed its execution successfully is committed; otherwise, it is aborted. The state of the data accessed by an aborted transaction must be restored to what it was just before the transaction started executing. We say that such a transaction has been rolled back. Various types of storage media are distinguished by their relative speed/capacity, and resilience to failure:

- **Volatile storage**: Information residing in volatile storage does not usually survive system crashes. Examples of such storage are main and cache memory. Access to volatile storage, is extremely fast, both because of the speed of the memory access itself and because it is possible to access directly any data item in volatile storage.
- **Nonvolatile storage**: Information residing in nonvolatile storage usually survives system crashes. Examples of media for such storage are disk and magnetic tapes. Disks are more reliable than is main memory, but are less reliable than are magnetic tapes. Both disks and tapes, however, are subject to failure, which may result in loss of information. Currently, nonvolatile storage is slower than volatile storage by several orders of magnitude, because disk and tape devices are electromechanical and require physical motion to access data.
- **Stable storage**: Information residing in stable storage is never lost. To implement an approximation of such storage, we need to replicate information in several nonvolatile storage caches (usually disk) with independent failure modes, and to update the information in a controlled manner

7.2 Log-Based Recovery

One way to ensure atomicity is to record, on stable storage, information describing all the modifications made by the transaction to the various data it accessed. The most widely used method for achieving this form of recording is write-ahead logging. 'The system maintains, on stable storage, a data structure called the log. Each log record describes a single operation of a transaction write, and has the following fields:

- **Transaction name**: The unique name of the transaction that performed the write operation
- **Data item name**: The unique name of the data item written
- **Old value**: The value of the data item prior to the write
- **New value**: The value that the data item will have after the write

Before a transaction Ti starts its execution, the record <Ti starts > is written to the log. During its execution, any write operation by Ti, is preceded by the writing of the appropriate new record to the log. When Ti commits, the record <Ti commits > is written to the log. Using the log, the system can handle any failure that does not result in the loss of information on nonvolatile storage. The recovery algorithm uses two procedures:

- **undo(Ti)**: which restores the value of all data updated by transaction Ti to the old values
- **redo(Ti)**:, which sets the value of all data updated by transaction Ti, to the new values

The set of data updated by Ti and their respective old and new values can be found in the log. This classification of transactions is accomplished as follows:

1. Transaction Ti needs to be undone if the log contains the record <Ti starts>, but does not contain the record <Tj commits >.
2. Transaction Ti needs to be redone if the log contains both the record <Ti starts> and the record <Ti commits>

7.3 Checkpoints

When a system failure occurs, we must consult the log to determine those transactions that need to be redone and those that need to be undone. There are two major drawbacks to this approach:

1. The searching process is time-consuming.
2. Most of the transactions that, according to our algorithm, need to be redone, have already actually updated the data that the long says they need to modify. Although redoing the data modifications will cause no harm, it will nevertheless cause recovery to take longer.

To reduce these types of overhead, we introduce the concept of checkpoints. During execution, the system maintains the write-ahead log. In addition, the system periodically performs checkpoints, which require the following sequence of actions to take place:

1. Output all log records currently residing in volatile storage onto stable storage.
2. Output all modified data residing in volatile storage to the stable storage.
3. Output a log record <checkpoint> onto stable storage.

The presence of a <checkpoint> record in the log allows the system to streamline its recovery procedure.

Consider a transaction Ti that committed prior to the checkpoint. The <Ti commits> record appears in the log before the <checkpoint> record. Any modifications made by Ti must have been written to stable storage either prior to the checkpoint, or as part of the checkpoint itself. Thus, at recovery time, there is no need to perform a redo operation on Ti The recovery operations that are required are as follows:

- For all transactions Tk in T such that the record <Tk commits > appears in the log, execute redo(Tk).
- For all transactions Tk in T that have no <Tk commits > record in the log, execute undo(Tk).

7.4 Concurrent Atomic Transactions

Because each transaction is atomic, the concurrent execution of transactions must be equivalent to the case where these transactions executed serially in some arbitrary order. This property, called Serializability, can be maintained by simply executing each transaction within a critical section. There are a number of different concurrency-control algorithms to ensure Serializability.

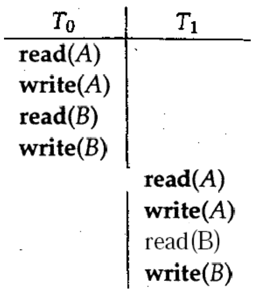
7.4.1 Serializability

Consider a system with two data items A and B that are both read and written by two transactions T0 and T1. Suppose that these transactions are executed atomically in the order T0 followed by T1. This execution sequence, which is called a schedule.

Schedule 1: A serial schedule in which T0 is followed by T1

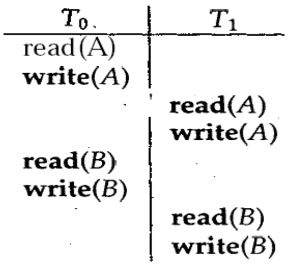
A schedule where each transaction is executed atomically is called a serial schedule. Each serial schedule consists of a sequence of instructions from various transactions where the instructions-belonging to one single transaction appear together in that schedule. If we allow the two transactions to overlap their execution, then the resulting schedule is no longer serial. A non-serial schedule does not necessarily imply that the resulting execution is incorrect. To see that this is the case, we need to define the notion of conflicting operations. Consider a schedule S in which there are two consecutive operations of and Oj of transactions Ti and Tj, respectively. We say that Oj and Oj conflict if they access the same data item, and at least one of these operations is write operation.

Schedule - 1



T_0	T_1
read(A)	
write(A)	
read(B)	
write(B)	
	read(A)
	write(A)
	read(B)
	write(B)

Schedule - 2



T_0	T_1
read(A)	
write(A)	
	read(A)
	write(A)
read(B)	
write(B)	
	read(B)
	write(B)

Schedule 2: A concurrent serializable schedule

As the write(A) operation of TI does not conflict with the read(B) operation of T0, we can swap these operations to generate an equivalent schedule. Regardless of the initial system state, both schedules produce the same final system state. Continuing with this procedure of swapping non-conflicting operations, we get:

- Swap the read(B) operation of T0 with the read(A) operation of T1.
- Swap the write(B) operation of T0 with the write(A) operation of T1.
- Swap the write(B) operation of T0 with the read(A) operation of T1.

If a schedule S can be transformed into a serial schedule S by a series of swaps of non-conflicting operations, we say that a schedule S' is conflict serializable. Thus, schedule 2 is conflict serializable, because it can be transformed into the serial schedule 1.

7.4.2 Locking Protocol

One way to ensure serializability is to associate with each data item to require that each transaction follow a locking protocol that governs how locks are acquired and released. There are various modes in which a data item can be locked. In this section, we restrict our attention to two modes:

- **Shared**: If a transaction Ti has obtained a shared-mode lock (denoted by S) on data item Q, then Ti can read this item, but it cannot write Q.
- **Exclusive**: If a transaction Ti has obtained an exclusive-mode lock (denoted by X) on data item Q, then Ti can both read and write Q.

To access a data item Q, transaction Ti must first lock Q in the appropriate mode. If Q is not currently locked, then the lock is granted, and Ti can now access it. However, if the data item Q is currently locked by some other transaction, then Ti may have to wait. More specifically, suppose that Ti requests an exclusive lock on Q. In this case, Ti must wait until the lock on Q is released. If Ti requests a shared lock on Q, then Ti must wait if Q is locked in exclusive mode.

One protocol that ensures serializability is the two-phase locking protocol. This protocol requires that each transaction issue lock and unlock requests in two phases:

- **Growing phase**: A transaction may obtain locks, but may not release any lock.
- **Shrinking phase**: A transaction may release locks, but may not obtain any new locks.

Initially, a transaction is in the growing phase. The transaction acquires locks as needed. Once the transaction releases a lock, it enters the shrinking phase, and no more lock requests can be issued.

7.4.3 Timestamp-Based Protocols

Another method for determining the serializability order is to select an ordering among transactions in advance. The most common method for doing so is to use a timestamp ordering scheme. With each transaction Ti in the system, we associate a unique fixed timestamp, denoted by TS(Ti). This timestamp is assigned by the system before the transaction Ti starts execution. If a transaction Tj has been assigned timestamp TS(Ti), and later on a new transaction Tj enters the system, then TS(Ti) < TS(Tj).

There are two simple methods for implementing this scheme:

1. **Use the value of the system clock as the timestamp**: that is, a transaction's timestamp is equal to the value of the clock when the transaction enters the system. This method will not work for transactions that occur on separate systems or for processors that do not share a clock.
2. **Use a logical counter as the timestamp**: that is, a transaction's timestamp is equal to the value of the counter when the transaction enters the system. The counter is incremented after a new timestamp is assigned.

The timestamps of the transactions determine the serializability order. Thus, if TS(Ti) < TS(Tj), then the system must ensure that the produced schedule is equivalent to a serial schedule in which transaction Ti appears before transaction Tj. To implement this scheme, we associate with each data item Q two timestamp values:

- **W-timestamp(Q)**, which denotes the largest timestamp of any transaction that executed write(Q) successfully
- **R-timestamp(Q)**, which denotes the largest timestamp of any transaction that executed read(Q) successfully

The timestamp-ordering protocol ensures that any conflicting read and write operations are executed in timestamp order. This protocol operates as follows:

- Suppose that transaction Ti issues read(Q):
  1. If TS(Ti) < W-timestamp(Q), then this state implies that Ti needs to read a value of Q which was already overwritten. Hence, the read operation is rejected, and Ti is rolled back.
  2. If TS(Ti) > W-timestamp(Q), then the read operation is executed, and R-timestamp(Q) is set to the maximum of R-timestamp(Q) and TS(Ti).
- Suppose that transaction Ti issues write(Q):
  1. If TS(Ti) < R-timestamp(Q), then this state implies that the value of Q that Ti is producing was needed previously and Ti assumed that this value would never be produced. Hence, the write operation is rejected, and Ti is rolled back.
  2. If TS(Ti) < W-timestamp(Q) then this state implies that Ti, is attempting to write an obsolete value of Q. Hence, this write operation is rejected, and Ti is rolled back.
  3. Otherwise, the write operation is executed.

****

DEADLOCKS

A process requests resources; if the resources are not available at that time, the process enters a wait state. It may happen that waiting processes will never again change state, because the resources they have requested are held by other waiting processes. This situation is called a deadlock.

1. System Model:

If a process requests an instance of a resource type, the allocation of any instance of the type will satisfy the request. If it will not, then the instances are not identical, and the resource type classes have not been defined properly. For example, a system may have two printers.

A process must request a resource before using it, and must release the resource after using it. A process may request as many resources as it requires carrying out its designated task. Obviously, the number of resources requested may not exceed the total number of resources available in the system. In other words, a process cannot request three printers if the system only has two. Under the normal mode of operation, a process may utilize a resource in only the following sequence:

1. Request: If the request cannot be granted immediately, then the requesting process must wait until it can acquire the resource.
2. Use: The process can operate on the resource.
3. Release: The process releases the resource.

The request and release of resources are system calls. A system table records whether each resource is free or allocated, and, if a resource is allocated, to which process. If a process requests a resource that is currently allocated to another process, it can be added to a queue of processes waiting for this 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 by only another process in the set. The resources may be either physical resource (for example, printers, tape drives, memory space, and CPU cycles) or logical resources (for example, files, semaphores, and monitors).

2 Deadlock Characterization:

Various methods for dealing with the deadlock problem, we shall describe features that characterize deadlocks.

2.1 Necessary Conditions

A deadlock situation can arise if the following four conditions hold simultaneously in a system:

1. 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.
2. 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.
3. No preemption: Resources cannot be preempted; that is, resources can be released only voluntarily by the process holding it, after that process has completed its task.
4. Circular wait: A set {P0, P1, ..., Pn} of waiting processes such that P0 is waiting for a resource that is held by P1, P1 is waiting for a resource that is held by P2,..., Pn-1 is waiting for a resource that is held by Pn, and Pn is waiting for a resource that is held by P0.

2.2 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 = {P1, P2,

..., Pn}, the set consisting of all the active processes in the system, and R = {R1, R2, ..., Rm}, the set consisting of all resource types in the system.

A directed edge from process Pi to resource type Rj is denoted by Pi→Rj. it signifies that process Pi has requested an instance of resource type Rj and is currently waiting for that resource. 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 Pi→Rj is called, a request edge; a directed edge Rj→Pi is called an assignment edge.

Ex: let the sets P, R, and E:

P = {P1, P2, P3}
R = {R1, R2, R3, R4}
E = {P1→R1, P2→R3, R1→P2, R2→P2, R2→P1, R3→P3}

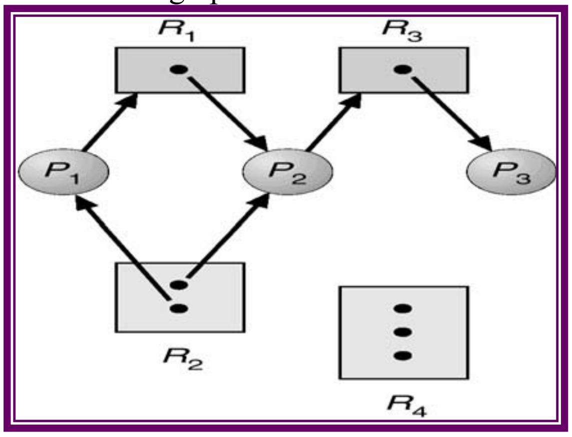
Resource instances:

One instance of resource type R1, Two instances of resource type R2, One instance of resource type R3, Three instances of resource type R4

Process states:

- Process P1 is holding an instance of resource type R2, and is waiting for an instance of resource type R1.
- Process P2 is holding an instance of R1 and R2, and is waiting for an instance of resource type R3.
- Process P3 is holding an instance of R3.

Resource allocation graph is as shown.



If each resource type has exactly one instance, then a cycle implies that a deadlock has occurred. If the cycle involves only a set of resource types, each of which has only a single instance, then a deadlock has occurred. Each process involved in the cycle is deadlocked. Suppose if process P3 requests an instance of resources type R2. At this point, two minimal cycles exist in the system:

P1→R1→P2→R3→P3--.R2→P1
P2→R3→P3→R2→P2

Processes P1, P2 and P3 are deadlocked. Process P2 is waiting for the resource R3, which is held by process P3. Process P3 is waiting for either process P1 or process P2 to release resource R1. Now consider the resource allocation graph as shown. We have the cycle: P1→R1→P3→R2→P1

3. Methods for Handling Deadlocks

There are three different methods for dealing with the deadlock problem:

- We can use a protocol to ensure that the system will never enter a deadlock state.
- We can allow the system to enter a deadlock state and then recover.
- We can ignore the problem all together, and pretend that deadlocks never occur in the system.

Deadlock prevention is a set of methods for ensuring that at least one of the necessary conditions cannot hold. These methods prevent deadlocks by constraining how requests for resources can be made. Deadlock avoidance, on the other hand, requires that the operating system be given in advance additional information concerning which resources a process will request and use during its lifetime. With this additional knowledge, we can decide for each request whether or not the process should wait. Each request requires that the system consider the resources currently available, the resources currently allowed to each process, and future requests and releases of each process.

4. Deadlock Prevention:

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.

4.1 Mutual Exclusion

The mutual-exclusion condition must hold for non-sharable resources. For example, a printer cannot be simultaneously shared by several processes. Sharable resources, on the other hand, do not require mutually exclusive access, and thus cannot be involved in a deadlock. Read-only files are a good example of a sharable resource. If several processes attempt to open a read-only file at the same time, they can be granted simultaneous access to the file. A process never needs to wait for sharable resources.

4.2 Hold and Wait

One protocol that can be used requires each process to request and be allocated all its resources before it begins execution. We can implement this provision by requiring that system calls requesting resources for a process precede all other system calls.

An alternative protocol allows a process to request resources only when process has none. 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.

Consider a process that copies data from a tape drive to a disk file, sorts the disk 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 tape drive, disk file, and the 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 tape drive and disk file. It copies from the tape drive to the disk, and then releases both the tape 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.

There are two main disadvantages to these protocols.

1. Resource utilization may be low
2. Starvation is possible

4.3 No Preemption

We can use the following protocol. If a process that is holding some resources requests another resource that cannot be immediately allocated to it, then all resources currently being held are preempted. In other words, these resources are implicitly released. The preempted resources are added to the list of resources for which the process is waiting. The process will be restarted only when it can regain its old resources, as well as the new ones that it is requesting.

Alternatively, if a process requests some resources, we first check whether they are available. If they are, we allocate them. If they are not available, we check whether they are allocated to some other process that is waiting for additional resources. If so, we preempt the desired resources from the waiting process and allocate them to the requesting process. If the resources are not either available or held by a waiting process, the requesting process must wait.

4.4 Circular Wait

One way to ensure that the circular-wait 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. Let R = {R1, R2,…, 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. Formally, we define a one-to-one function F: R→ N, where N is the set of natural numbers. For example, if the set of resource types R includes tape drives, disk drives, and printers, then the function F might be defined as follows:

F(tape drive)	= 1
F(disk drive)	= 5
F(Printer)	= 12

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 Ri. After that, the process can request instances of resource type Rj if and only if F(Ri) > F(Rj). 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 Rj, it has released any resources Ri such that F(Ri)≥F(Rj)

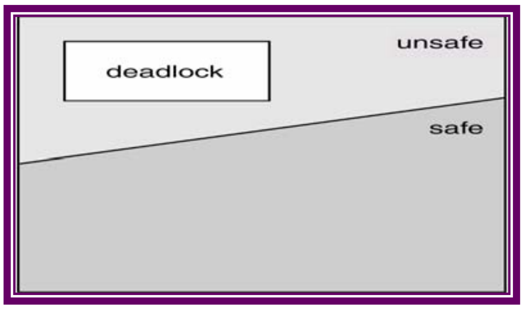
5. Deadlock Avoidance:

Possible side effects of preventing deadlocks by this method, however, 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. The simplest and most useful model requires that each process declare the maximum number of resources of each type that it may need. A deadlock avoidance algorithm dynamically examines the resource-allocation state to ensure that there can never be a circular-wait condition. The resource-allocation state is defined by the number of available and allocated resources, and the maximum demands of the processes.

5.1 Safe State

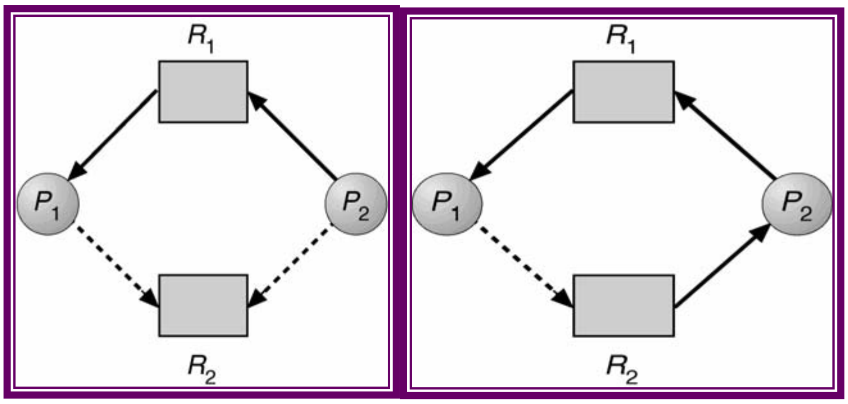
A state is safe if the system can allocate resources to each process in some order and still avoid a deadlock. More formally, a system is in a safe state only if there exists a safe sequence. A sequence of processes <P1, P2, ..., Pn> is a safe sequence for the current allocation state if, for each Pi, the resources that Pi can still make can be satisfied by the currently available resources plus the resources held by all Pj, with j<i.

A safe state is not a deadlock state. Conversely, a deadlock state is an unsafe state. Not all unsafe states are deadlocks. An unsafe state may lead to a deadlock. As long as the state is safe, the operating system can avoid unsafe states. In an unsafe state, the operating system cannot prevent processes from requesting resources such that a deadlock occurs: The behavior of the processes controls unsafe states.



5.2 Resource-Allocation Graph Algorithm

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 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 → Pi is reconverted to a claim edge Pi → Rj.



Suppose that process Pi requests resource Rj. The request can be granted only if converting the request edge Pi → 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 in an unsafe state. Therefore, process Pj will have to wait for its requests to be satisfied. If P1 requests R2, and P2 requests R1, then a deadlock will occur.

5.3 Banker's Algorithm

The deadlock-avoidance algorithm that we describe next is applicable to such a system, but is less efficient than the resource-allocation graph scheme. This algorithm is commonly known as the banker's algorithm. 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 Available[j] = k, there are k instances of resource type Rj available.
- Max: An n x m matrix defines the maximum demand of each process. If Max [i,j] = k, then P; may request at most k instances of resource type Rj.
- Allocation: An n x m matrix defines the number of resources of each type currently allocated to each process. If Allocation [i,j] = k, then process Pi is currently allocated k instances of resource type Rj.
- Need: An n x m matrix indicates the remaining resource need of each process. If Need[i,j] = k, then Pi may need k more instances of resource type Rj to complete its task.

  Note that Need[i,j] = Max[i,j] - Allocation[i,j].

5.3.1 Safety Algorithm

The algorithm for finding out whether or not a system is in a safe state can be described as follows:

1. Let Work and Finish be vectors of length m and n, respectively. Initialize
   Work: = Available and Finish[i]:=false for i = 1, 2,..., n.

2. Find an *i* such that:
   Finish[i] = false and $Need_i \leq Work$
   If no such *i* exists, go to Step 4
3. Work := Work + $Allocation_i$
   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 requires an order of m x n^2 operations to detect whether the state is safe state or not.

5.3.2 Resource-Request Algorithm

Let $Request_i$ be the request vector for process P_i. If $Request_i[j] = k$, then process P_i wants k instances of resource type R_j. When a request for resources is made by process P_i, the following actions are taken:

1. If $Request_i \leq Need_i$, go to Step 2. Otherwise, raise an error condition, since the process has exceeded its maximum claim.
2. If $Request_i \leq Available$, go to Step 3. Otherwise, P_i must wait, since the resources are not available.
3. Have the system pretend to have allocated the requested resources to process P_i by modifying the state as follows:
   Available: = Available - $Request_i$
   $Allocation_i$: = $Allocation_i$ + $Request_i$
   $Need_i$:= $Need_i$ - $Request_i$

NOTE: Refer class note book for examples

6. Deadlock Detection:

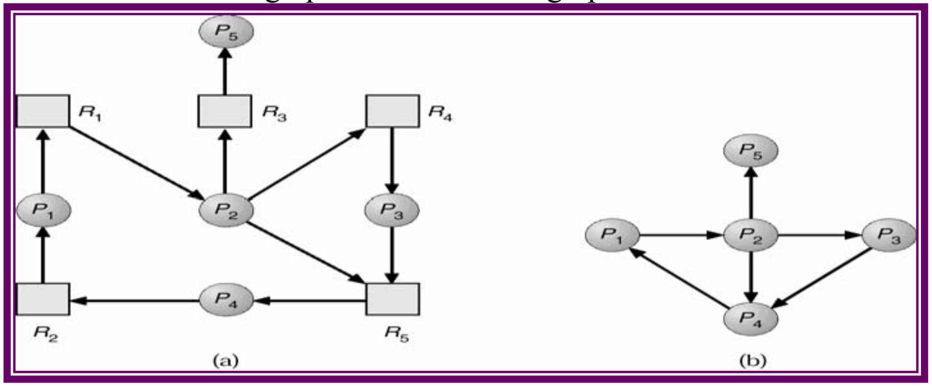
If a system does not employ either a deadlock-prevention or a deadlock-avoidance algorithm, then a deadlock situation may occur. In this environment, the system must provide:

1. An algorithm that examines the state of the system to determine whether a deadlock has occurred
2. An algorithm to recover from the deadlock

6.1 Single Instance of Each Resource Type

If all resources have only a single instance, then we can define a deadlock-detection 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 nodes of type resource and collapsing the appropriate edges. 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→ Rq and Rq→ Pj for some resource Rq.

Resource allocation graph and its wait for graph are as follows:



6.2 Several Instances of a Resource Type

The wait-for graph scheme is not applicable to a resource-allocation system with 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 m matrix indicates the current request of each process. If Request[i,j] = k, then process Pi is requesting k more instances of resource type Rj.

Deadlock Detection Algorithm:

1. Let Work and Finish be vectors of length m and n, respectively. Initialize Work: =Available. For i=0,1,..,n-1, if $Allocation_i \neq 0$, then Finish[i]:=false; otherwise, Finish[i] := true.
2. Find an index i such that:
   Finish[i] = false and $Request_i \leq Work$
   If no such i exists, go to Step 4
3. Work := Work + $Allocation_i$
   Finish[i]:= true
   go to Step 2
4. If Finish[i] = false, for some i, $0 \leq i < n$, then the system is in a deadlock state. Moreover, if Finish[i] =false, then process P_i is deadlocked.

This algorithm requires an order of m x n^2 operations to detect whether the system is in deadlocked state.

NOTE: Refer class note book for examples

6.3 Detection-Algorithm Usage

When should we invoke the detection algorithm? The answer depends on two factors:
1. How often is a deadlock likely to occur?
2. How many processes will be affected by deadlock when it happens?

If deadlocks occur frequently, then the detection algorithm should be invoked frequently. Resources allocated to deadlocked processes will be idle until the deadlock can be broken. In addition, the number of processes involved in the deadlock cycle may grow.

We can invoke the deadlock-detection algorithm every time a request for allocation cannot be granted immediately. In. this case, we can identify not only the set of processes that is deadlocked, but also the specific process that "caused" the deadlock. Of course, invoking the deadlock-detection algorithm for every request may incur a considerable overhead in computation time. A less expensive alternative is simply to invoke the algorithm at less frequent intervals.

7 Recovery from Deadlock:

One possibility is to inform the operator that a deadlock has occurred, and to let the operator deal with the deadlock manually. The other possibility is to let the system recover from the deadlock automatically. There are two options for breaking a deadlock. One solution is simply to abort one or more processes to break the circular wait. The second option is to preempt some resources from one or more of the deadlocked processes.

7.1 Process Termination

To eliminate deadlocks by aborting a process, we use one of two methods. In both methods, the system reclaims all resources allocated to the terminated processes.

- Abort all deadlocked processes: This method clearly will break the deadlock cycle, but at a great expense, since these processes may have computed for a long time, and the results of these partial computations must be discarded, and probably must be recomputed later.
- 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.

If the partial termination method is used, then, given a set of deadlocked processes, we must determine which process (or processes) should be terminated in an attempt to break the deadlock. Many factors may determine which process is chosen, including:

1. What the priority of the process is
2. How long the process has computed, and how much longer the process will compute before completing its designated task
3. How many and what type of resources the process has used (for example, whether the resources are simple to preempt)
4. How many more resources the process needs in order to complete
5. How many processes will need to be terminated
6. Whether the process is interactive or batch

7.2 Resource Preemption

If preemption is required to deal with deadlocks, then three issues need to be addressed:

1. 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. Cost factors may include such parameters as the number of resources a deadlock process is holding, and the amount of time a deadlocked process has thus far consumed during its execution.
2. Rollback: If we preempt a resource from a process, what should be done with that process? Clearly, it cannot continue with its normal execution; it is missing some needed resource. We must roll back the process to some safe state, and restart it from that state.
3. Starvation: How do we ensure that starvation will not occur? That is, how can we guarantee that resources will not always be preempted from the same process?

7.8 Combined Approach to Deadlock Handling

None of the basic approaches for handling deadlocks (prevention, avoidance, and detection) alone is appropriate for the entire spectrum of resource-allocation problems encountered in operating systems.

One possibility is to combine the three basic approaches, allowing the use of the optimal approach for each class of resources in the system. To illustrate this technique, we consider a system that consists of the following four classes of resources:

- Internal resources: Resources used by the system, such as a process control block
- Central memory: Memory used by a user job
- Job resources: Assignable devices (such as a tape drive) and files
- Swappable space: Space for each user job on the backing store

One mixed deadlock solution for this system orders the classes as shown, and uses the following approaches to each class:

- Internal resources: Prevention through resource ordering can be used, since run-time choices between pending requests are unnecessary.

- Central memory: Prevention through preemption can be used, since a job can always be swapped out, and the central memory can be preempted.
- Job resources: Avoidance can be used, since the information needed about resource requirements can be obtained from the job-control cards.
- Swappable space: Pre-allocation can be used, since the maximum storage requirements are usually known.

MEMORY MANAGEMENT

1.1 Address Binding

Usually, a program resides on a disk as a binary executable file. The program must be brought into memory and placed within a process. The collection of processes on the disk that are waiting to be brought into memory for execution forms the *input queue.*

In most cases, a user program will go through several steps before being executed. Addresses may be represented in different ways during these steps. Addresses in the source program are generally symbolic (such as COUNT). A compiler will typically bind these symbolic addresses to re-locatable addresses (such as 14 bytes from the beginning of this module). The linkage editor or loader will in turn bind these re-locatable addresses to absolute addresses (such as 74014). Each binding is a mapping from one address space to another. Classically, the binding of instructions and data to memory addresses can be done at any step along the way:

- Compile time: If it is known at compile time where the process will reside in memory, then absolute code can be generated. For example, if it is known a priori that a user process resides starting at location jR, then the generated compiler code will start at that location and extend up from there. If, at some later time, the starting location changes, then it will be necessary to recompile this code. The MS-DOS .COM-format programs are absolute code bound at compile time.
- Load time: If it is not known at compile time where the process will reside in memory, then the compiler must generate re-locatable code. In this case, final binding is delayed until load time. If the starting address changes, we need only to reload the user code to incorporate this changed value.
- Execution time: *If* the process can be moved during its execution from one memory segment to another, then binding must be delayed until run time. Special hardware must be available for this scheme to work

1.2 Dynamic Loading:

To obtain better memory-space utilization, we can use dynamic loading. With dynamic loading, a routine is not loaded until it is called. All routines are kept on disk in a re-locatable load format. The main program is loaded into memory and is executed. When a routine needs to call another routine, the calling routine first checks to see whether the other routine has been loaded or not. If it has not been, the re-locatable linking loader is called to load the desired routine into memory and to update the program's address tables to reflect this change. Then, control is passed to the newly loaded routine.

The advantage of dynamic loading is that an unused routine is never loaded. Dynamic loading does not require special support from the operating system. It is the responsibility of the users to design their programs to take advantage of such a scheme.

1.3 Dynamic Linking:

Most operating systems support only static linking, in which system language libraries are treated like any other object module and are combined by the loader into the

binary program image. The concept of dynamic linking is similar to that of dynamic loading. Rather than loading being postponed until execution time, linking is postponed. This feature is usually used with system libraries, such as language subroutine libraries.
When this stub is executed, it checks to see whether the needed routine is already in memory. If the routine is not in memory, the program loads it into memory. Either way, the stub replaces itself with the address of the routine, and executes the routine. A library may be replaced by a new version, and all programs that reference the library will automatically use the new version.

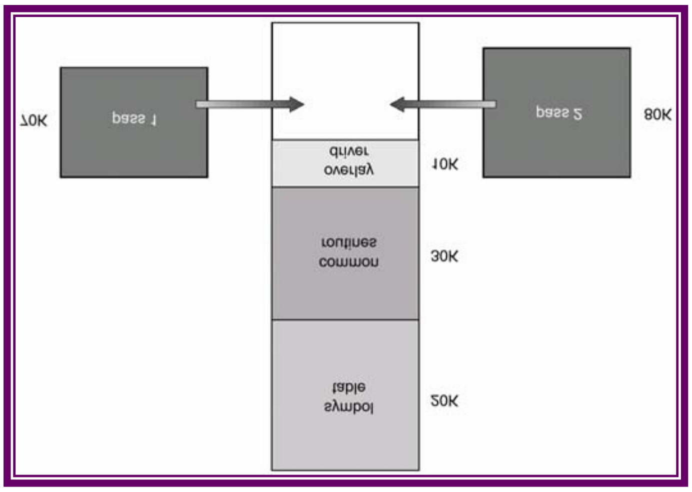
Only programs that are compiled with the new library version are affected by the incompatible changes incorporated in it. Other programs linked before the new library was installed will continue using the older library. This system is also known as shared libraries. Dynamic linking generally requires some help from the operating system.

1.4 Overlays:

If the process size is larger than the amount of memory allocated to it, a technique called *overlays* is sometimes used. The idea of overlays is to keep in memory only those instructions and data that are needed at any given time. When other instructions are needed, they are loaded into space that was occupied previously by instructions that are no longer needed.

Ex: Consider a two-pass assembler. During pass 1, it constructs a symbol table; then, during pass 2, it generates machine-language code. We may be able to partition such an assembler into pass 1 code, pass 2 code, the symbol table and common support routines used by both pass 1 and pass 2. Assume that the sizes of these components are as follows:

Pass 1	70K
Pass 2	80K
Symbol table	20K
Common routines	30K

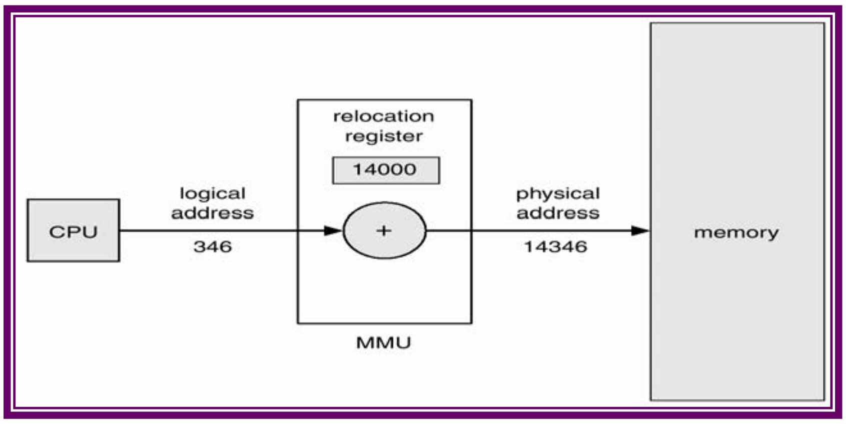


To load everything at once, we would require 200K of memory. If only 150K is available, we cannot run our process. However, notice that pass 1 and pass 2 do not need to be in memory at the same time. We thus define two overlays:
Overlay *A* is the symbol table, common routines, and pass 1, and overlay B is the symbol table, common routines, and pass 2. We add an overlay driver (10K) and start with overlay *A* in memory. When we finish pass 1, we jump to the overlay driver, which reads overlay B into memory, overwriting overlay *A*, and then transfers control to pass 2. Overlay *A* needs only 120K, whereas overlay B needs 130K. As in dynamic loading, overlays do not require any special support from the operating system. They can be

implemented completely by the user with simple file structures, reading from the files into memory and then jumping to that memory and executing the newly read instructions.

1.5 Logical versus Physical Address Space:

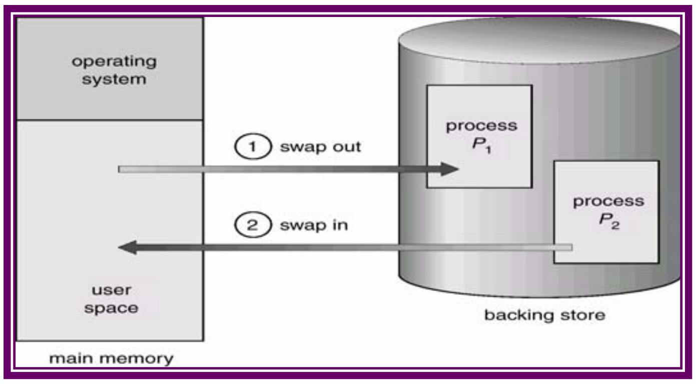
An address generated by the CPU is commonly referred to as a *logical address,* whereas an address seen by the memory unit (that is, the one loaded into the *memory address register* of the memory) is commonly referred to as a *physical address.* The compile-time and load-time address-binding schemes result in an environment where the logical and physical addresses are the same.



However, the execution-time address-binding scheme results in an environment where the logical and physical addresses differ. In this case, we usually refer to the logical address as a virtual address. The set of all logical addresses generated by a program is referred to as a logical address space; the set of all physical addresses corresponding to these logical addresses is referred to as a physical address space. The run-time mapping from virtual to physical addresses is done by the *memory-management unit (MMU),* which is a hardware device. The base register is now called a *relocation* register. 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 never sees the *real* physical addresses. The program can create a pointer to location 346, store it in memory, manipulate it, and compare it to other addresses— all as the number 346. Only when it is used as a memory address is it relocated relative to the base register. The user program deals with *logical* addresses. The memory-mapping hardware converts logical addresses into physical addresses. Logical addresses is in the range 0 to *max,* similarly physical addresses is in the range R + 0 to R + *max* for a base value R. The user generates only logical addresses and thinks that the process runs in locations 0 to *max.*

3 Swapping:

A process, however, can be *swapped* temporarily out of memory to a *backing store,* and then brought back into memory for continued execution. The CPU scheduler will allocate a time slice to some other process in memory. When each process finishes its quantum it will be swapped with another process. A variant of this swapping policy is used for priority-based scheduling algorithms. If a higher-priority process arrives and wants service, the memory manager can swap out the lower-priority process so that it can load and execute the higher-priority process. When the higher-priority process finishes, the lower-priority process can be swapped back in and continued. This variant of swapping is sometimes called *roll out, roll in.*



Normally a process that is swapped out will be swapped back into the same memory space that it occupied previously. This restriction is dictated by the method of address binding. Swapping requires a *backing store*. The backing store is commonly a fast disk. The system maintains a *ready queue* consisting of all processes whose memory images are on the backing store or in memory and are ready to run. The dispatcher checks to see whether the next process in the queue is in memory. If the process is not, and there is no free memory region, the dispatcher swaps out a process currently in memory and swaps in the desired process. For efficient CPU utilization, we want our execution time for each process to be long relative to the swap time. Major part of the swap time is transfer time. The total transfer time is directly proportional to the amount of memory swapped. There are other constraints on swapping. If we want to swap a process, we must be sure that it is completely idle. If a process is waiting for an I/O operation, we may want to swap that process to free up its memory. Assume that the I/O operation was queued because the device was busy. Then, if we were to swap out process P1 and swap in process P2, the I/O operation might then attempt to use memory that now belongs to process P1. The two main solutions to this problem are:

(1) Never to swap a process with pending I/O

(2) To execute I/O operations only into operating-system buffers.

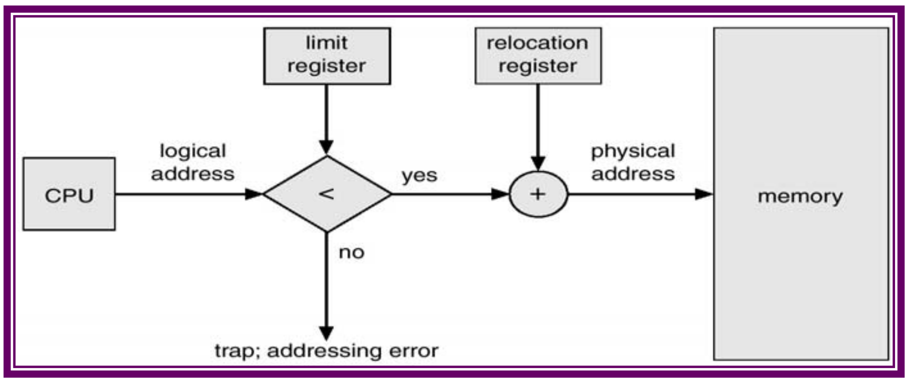
A modification of swapping is used in many versions of UNIX. Microsoft Windows 3.1 operating system, supports concurrent execution of processes in memory. If a new process is loaded and there is insufficient main memory, an old process is swapped to disk.

2 Contiguous Allocation:

The main memory must accommodate both the operating system and the various user processes. The memory is usually divided into two partitions, one for the resident operating system, and one for the user processes.

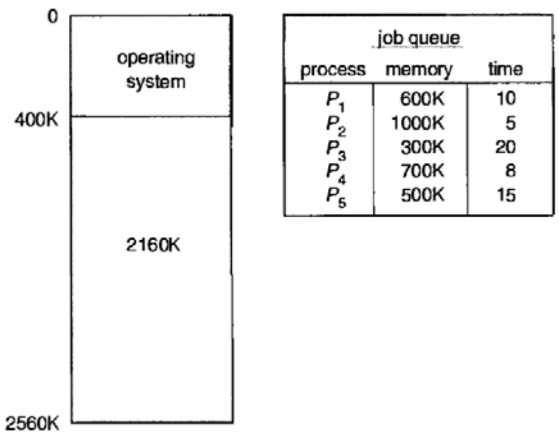
2.1 Single-Partition Allocation

The relocation register contains the value of the smallest physical address; the limit register contains the range of logical addresses (for example, relocation = 100,040 and limit = 74,600). With relocation and limit registers, each logical address must be less than the limit register; the MMU maps the logical address dynamically by adding the value in the relocation register. This mapped address is sent to memory. When the CPU scheduler selects a process for execution, the dispatcher loads the relocation and limit registers with the correct values as part of the context switch. If a device driver is not commonly used, it is undesirable to keep the code and data in memory, as we might be able to use that space for other purposes. Such code is sometimes called transient operating-system code



2.2 Multiple-Partition Allocation

One of the simplest schemes for memory allocation is to divide memory into a number of fixed-sized partitions. Each partition may contain exactly one process. Thus, the degree of multiprogramming is bound by the number of partitions. When a partition is free, a process is selected
from the input queue and is loaded into the free partition. When the process terminates, the partition becomes available for another process. The operating system keeps a table indicating which parts of memory are available and which are occupied. Initially, all memory is available for user processes, and is considered as one large block of available memory, a hole. When a process arrives and needs memory, we search for a hole large enough for this process. If we find one, we allocate only as much memory as is needed, keeping the rest available to satisfy future requests.



job queue

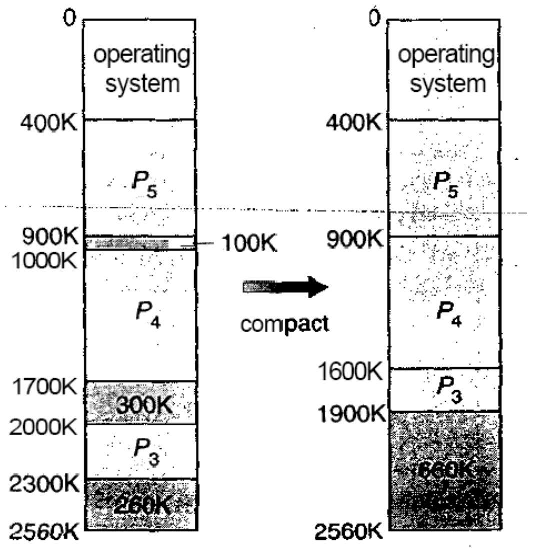
process	memory	time
P_1	600K	10
P_2	1000K	5
P_3	300K	20
P_4	700K	8
P_5	500K	15

For example, assume that we have 2560K of memory available and a resident operating system of 400K. This situation leaves 2160K for user processes. Given the input queue in the figure, and FCFS job scheduling, we can immediately allocate memory to processes P1, P2, and P3, creating the memory map. We have a hole of size 260K that cannot be used by any of the remaining processes in the input queue. at any time a set of holes, of various sizes, scattered throughout memory. When a process arrives and needs memory, we search this set for a hole that is large enough for this process. If the hole is too large, it is split into two: One part is allocated to the arriving process; the other is returned to the set of holes. If the new hole is adjacent to other holes, we merge these adjacent holes to form one larger hole. Dynamic storage allocation problem is how to satisfy a request of size n from a list of free holes. There are many solutions to this problem. The set of holes is searched to determine which hole is best to allocate. First-fit, best-fit, and worst-fit are the most common strategies used to select a free hole from the set of available holes.

- **First-fit**: Allocate the first hole that is big enough. Searching can start either at the beginning of the set of holes or where the previous first-fit search ended. We can stop searching as soon as we find a free hole that is large enough.
- **Best-fit:** Allocate the smallest hole that is big enough. We must search the entire list, unless the list is kept ordered by size. This strategy produces the smallest leftover hole.
- **Worst-fit:** Allocate the largest hole. Again, we must search the entire list, unless it is sorted by size. This strategy produces the largest leftover hole, which may be more useful than the smaller leftover hole from a best-fit approach.

2.3 External and Internal Fragmentation

As processes are loaded and removed from memory, the free memory space is broken into little pieces. External fragmentation exists when enough total memory space exists to satisfy a request, but it is not contiguous; storage is fragmented into a large number of small holes. Depending on the total amount of memory storage and the average process size, external fragmentation may be either a minor or a major problem. Statistical analysis of first-fit, for instance, reveals that, even with some optimization, given N allocated blocks, another 0.5N blocks will be lost due to fragmentation. That is, one-third of memory may be unusable! This property is known as the 50-percent rule.



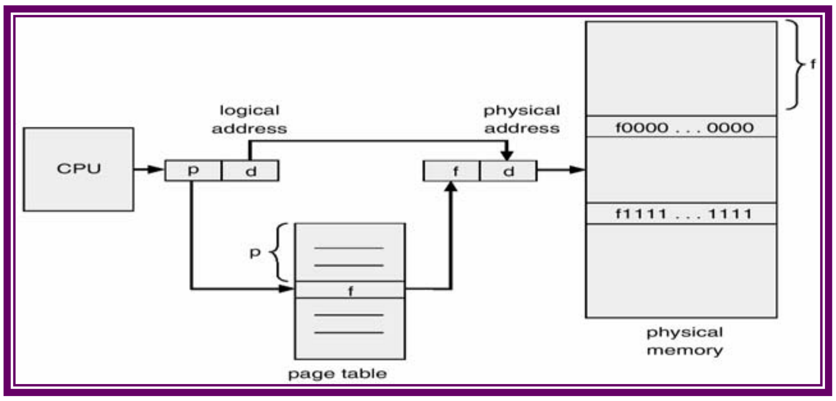
The general approach is to allocate very small holes as part of the larger request. Thus, the allocated memory be slightly larger than the requested memory. The difference between these two numbers is internal fragmentation—memory that is internal to a partition, but is not being used. One solution to the problem of external fragmentation is compaction. The goal is to shuffle the memory contents to place all free memory together in one large block. For example, The three holes of sizes 100K, 300K, and 260K can be compacted into one hole of size 660K. Compaction is possible only if relocation is dynamic and is done at execution time. Simplest compaction algorithm is to move all processes toward one end of memory, all holes move in the other direction, providing one large hole of available memory. This scheme can be expensive.

3 Paging:

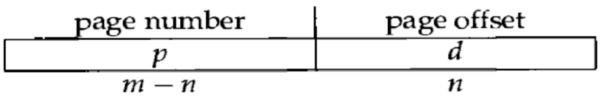
Paging is memory management scheme that permits the physical address space of process to be noncontiguous. Paging avoids the considerable problem of fitting memory chunks of varying sizes onto the backing stores. Paging avoids the considerable problem of fitting the varying-sized memory chunks onto the backing store.

3.1 Basic Method

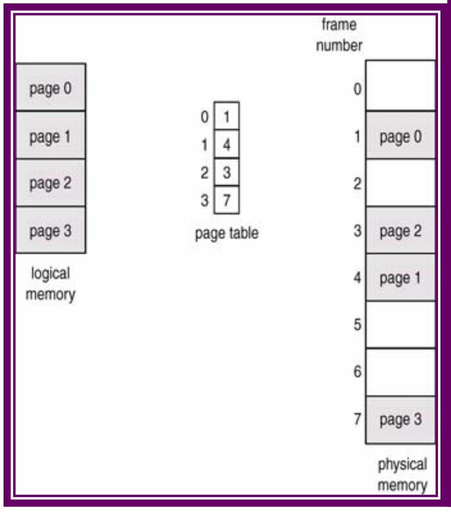
Physical memory is broken into fixed-sized blocks called frames. Logical memory is also broken into blocks of the same size called pages. When a process is to be executed, its pages are loaded into any available memory frames from the backing store. The backing store is divided into fixed-sized blocks that are of the same size as the memory frames. Every address generated by the CPU is divided into two parts: a page number (p) and a page offset (d).

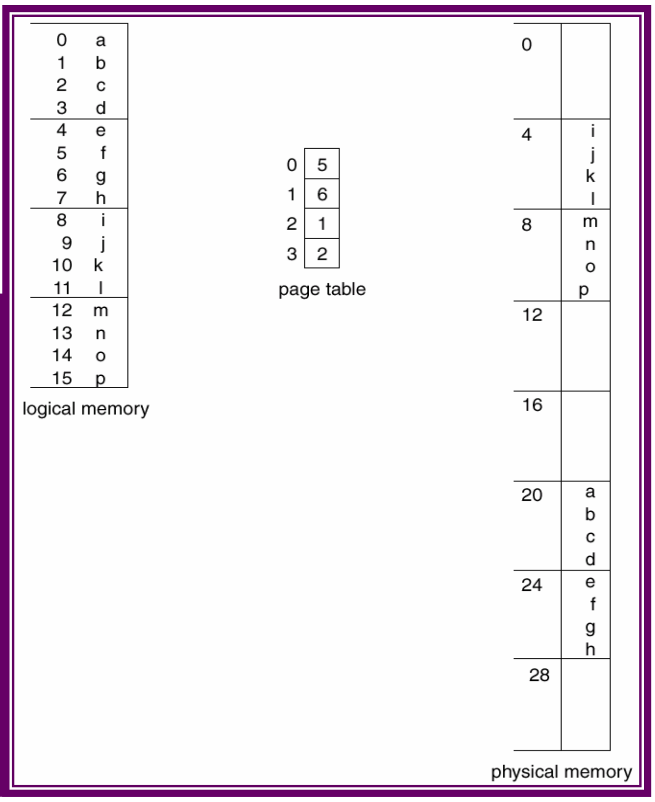


The page number is used as an index into a page table. The page table contains the base address of each page in physical memory. This base address is combined with the page offset to define the physical memory address that is sent to the memory unit. If the size of logical address space is 2m, and a page size is 2^n addressing units (bytes or words), then the high-order m-n bits of a logical address designate the page number, and the n low order bits designate the page offset. Thus, the logical address is as follows:

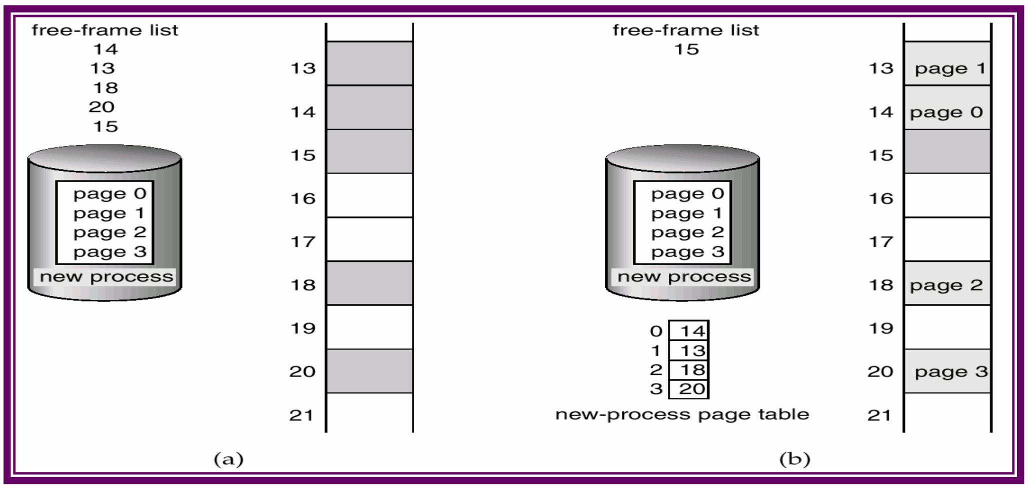


Where p is an index into the page table and d is the displacement within the page. Logical address 0 is page 0, offset 0. Indexing into the page table, we find that page 0 is in frame 5. Thus, logical address 0 maps to physical address 20 (= (5x4) + 0). Logical address 3 (page 0, offset 3) maps to physical address 23 (= (5x4) + 3). Logical address 4 is page 1, offset 0; according to the page table, page 1 is mapped to frame 6. Thus, logical address 4 maps to physical address 24 (= (6 x 4) + 0). Logical address 13 maps to physical address 9.When we use a paging scheme, we have no external fragmentation: Any free frame can be allocated to a process that needs it. However, we may have some internal fragmentation.





Generally, page sizes have grown over time as processes, data sets, and main memory have become larger. Today, pages typically are either 2 or 4 kilobytes. Generally, page sizes have grown over time as processes, data sets, and main memory have become larger. Today, pages typically are either 2 or 4 kilobytes. If the process requires n pages, there must be at least n frames available in memory. If there are n frames available, they are allocated to this arriving process. The first page of the process is loaded into one of the allocated frames, and the frame number is put in the page table for this process. The next page is loaded into another frame, and its frame number is put into the page table, and so on. The user program views that memory as one single contiguous space, containing only this one program.



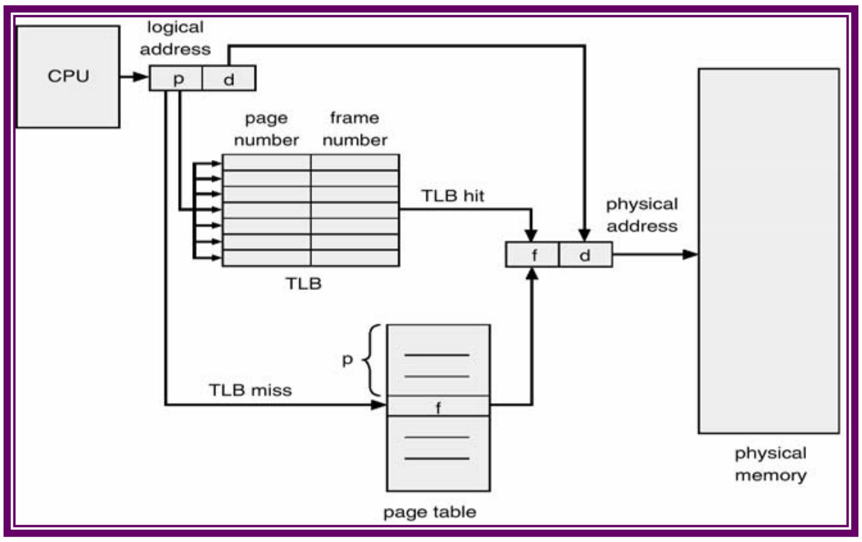
In fact, the user program is scattered throughout physical memory, which also holds other programs. The logical addresses are translated into physical addresses. This mapping is hidden from the user and is controlled by the operating system. It must be aware of the allocation details of physical memory: which frames are allocated, which frames are available, how many total frames there are, and so on. This information is generally kept in a data structure called a frame table. If a user makes a system call and provides an address as a parameter, that address must be mapped to produce the correct physical address. Paging therefore increases the context switch time.

3.2 Structure of the Page Table

Each operating system has its own methods for storing page tables. Most allocate a page table for each process. A pointer to the page table is stored with the other register values in the process control block.

3.2.1 Hardware Support

The hardware implementation of the page table can be done in a number of different ways. In the simplest case, the page table is implemented as a set of dedicated registers. These registers should be built with very high speed logic to make the paging address translation efficient. The CPU dispatcher reloads these registers, just as it reloads the other registers. Instructions to load or modify the page-table registers are, of course, privileged, so that only the operating system can change the memory map. The address consists of 16 bits, and the page size is 8K. The page table, thus, consists of eight entries that are kept in fast registers. Page-table base register (PTBR) points to the page table. Changing page tables requires changing only this one register, substantially reducing context-switch time. The problem with this approach is the time required to access a user memory location. If we want to access location i, we must first index into the page table, using the value in the PTBR offset by the page number for i. This task requires a memory access. It provides us with the frame number, which is combined with the page offset to produce the actual address. We can then access the desired place in memory. With this scheme, two memory accesses are needed to access a byte. The standard solution to this problem is to use a special, small, fast-lookup hardware cache, variously called associative registers or translation look-aside, buffers (TLBs).



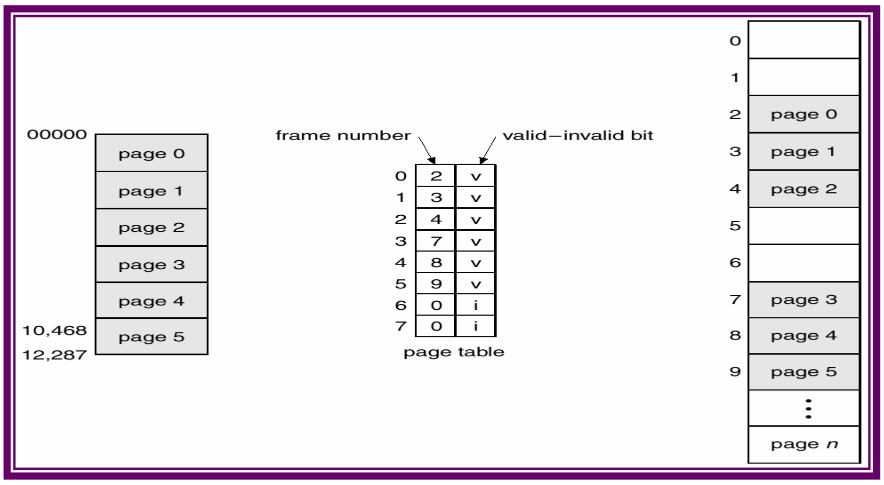
A set of associative registers is built of especially high-speed memory. Each register consists of two parts: a key and a value. When the associative registers are presented with an item, it is compared with all keys simultaneously. If the item is found, the corresponding value field is output. If the page number is not in the associative registers, a memory reference to the page table must be made. When the frame number is obtained, we can use it to access memory. If the TLB is already full of entries, the operating system must select one for replacement. Replacement policies range from least recently used (LRU) to random. Furthermore, some TLBs allow entries to be wired down, meaning that they cannot be removed from the TLB. Typically, TLB entries for kernel code are wired down.

Some TLBs store addresses space identifiers (ASID) in each TLB entry. An ASID uniquely identifies each process and is used to provide address space protection for that process. If the TLB does not support separate ASID, then every time a new page table is selected, the TLB must be flushed translation information. The percentage of times that a page number is found in the associative registers is called the hit ratio. If we fail to find the page number in the associative registers (20 nanoseconds), then we must first access memory for the page table and frame number (100 nanoseconds), and then access the desired byte in memory (100 nanoseconds), for a total of 220 nanoseconds. To find the effective memory-access time, we must weigh each case by its probability:

Effective Access Time = 0.80 x 120 + 0.20 x 220= 140 nanoseconds.

3.2.2 Protection

Memory protection in a paged environment is accomplished by protection bits that are associated with each frame. Normally, these bits are kept in the page table. One bit can define a page to be read and write or read-only. Every reference to memory goes through the page table to find the correct frame number. At the same time that the physical address is being computed, the protection bits can be checked to verify that no writes are being made to a read only page. An attempt to write to a read-only page causes a hardware trap to the operating system. One more bit is generally attached to each entry in the page table: a valid invalid bit. When this bit is set to 'Valid," this value indicates that the associated page is in the process's logical address space, and is thus a legal (valid) page.



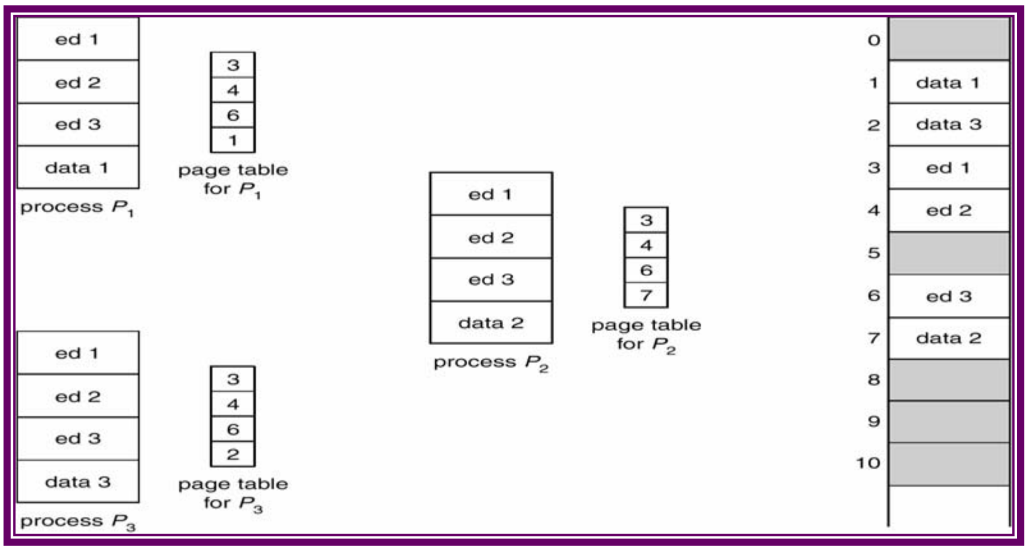
	frame number	valid–invalid bit
0	2	v
1	3	v
2	4	v
3	7	v
4	8	v
5	9	v
6	0	i
7	0	i

If the bit is set to "invalid," this value indicates that the page is not in the process's logical address space. Illegal addresses are trapped by using the valid-invalid bit.

Some systems provide hardware, in the form of page table length register (PTLR) to indicate the size of the page table. This value is checked against every logical address to verify that the address is in the valid range for the process.

3.2.3 Shared Pages:

An advantage of paging is the possibility of sharing common code. This consideration is particularly important in time sharing environment. Consider system that supports 40 users, each of whom executes text editor. If the text editor consists of 150 KB of each and 50KB of data space, we need 8000KB to support the 40 users. If the code is reentrant code, however it can be shared. Three editors each page 50KB in size being shared among three processes. Each process has its own data page.

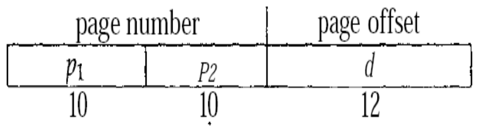


Reentrant code is non self modifying code, it never changes during execution. Only on copy of the editor need be kept in physical memory. Each user's page table maps onto the same physical copy of the editor, but data pages are mapped onto different frames.

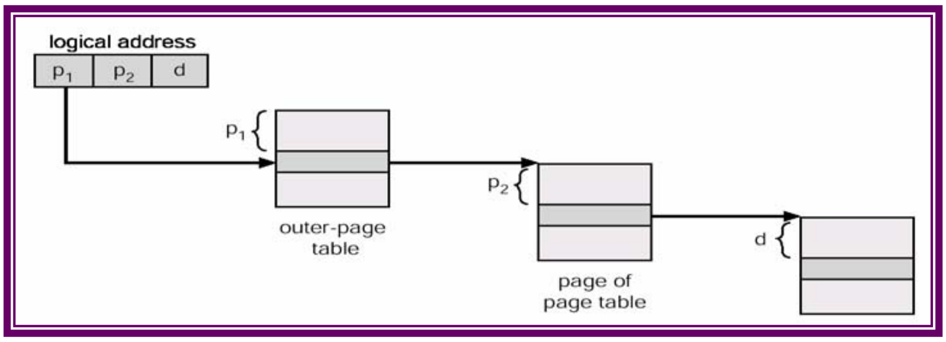
4 Structure of the page table:

4.1 Multilevel Paging/Hierarchical Paging

Most modern computer systems support a very large logical address space (232 to 264). 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 4K bytes (212), then a page table may consist of up to 1 million entries (232/212). Because each entry consists of 4 bytes, each process may need up to 4 megabytes of physical address space for the page table alone. One simple solution to this is to divide the page table into smaller pieces. There are several different ways to accomplish this. One way is to use a two-level paging scheme, in which the page table itself is also paged. To illustrate this, let us return to our 32-bit machine example, with a page size of 4K bytes. 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:



Where p is an index into the outer page table, and p2 is the displacement within the page of the outer page table. Because address translation works from the outer page table inward, this scheme is also known as forward mapped page table.



The VAX architecture supports two level paging. The VAX is a 32-bit machine with page size of 512 bytes. The logical address space of a process is divided into four equal sections, each of which consists of 230 bytes. Each section represents a different part of the logical address space of a process. The first 2 high-order bits of the logical address designate the appropriate section. The next 21 bits represent the logical page number of that section, and the last 9 bits represent an offset in the desired
page. By partitioning the page table in this manner, the operating system can leave partitions unused until a process needs them. An address on the VAX architecture is as follows:

section	page	offset
s	*p*	*d*
2	21	9

Where s designates the section number, p is an index into the page table, and d is the displacement within the page.

The size of a one-level page table for a VAX process using one section still is 221 bits i 4 bytes per entry = 8 megabytes. To further reduce main memory use, the VAX pages the user-process page tables. For a system with a 64-bit logical address space, a two-level paging scheme is no longer appropriate. If we use a two-level paging scheme, then the inner page tables could conveniently be 1 page long, or contain 210 four-byte entries. The addresses would look like:

outer page	inner page	offset
p1	*p2*	*d*
42	10	12

The outer page table will consist of 242 entries, or 2M bytes. The obvious method 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 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 (210 entries, or 212 bytes); a 64-bit address space is still daunting:

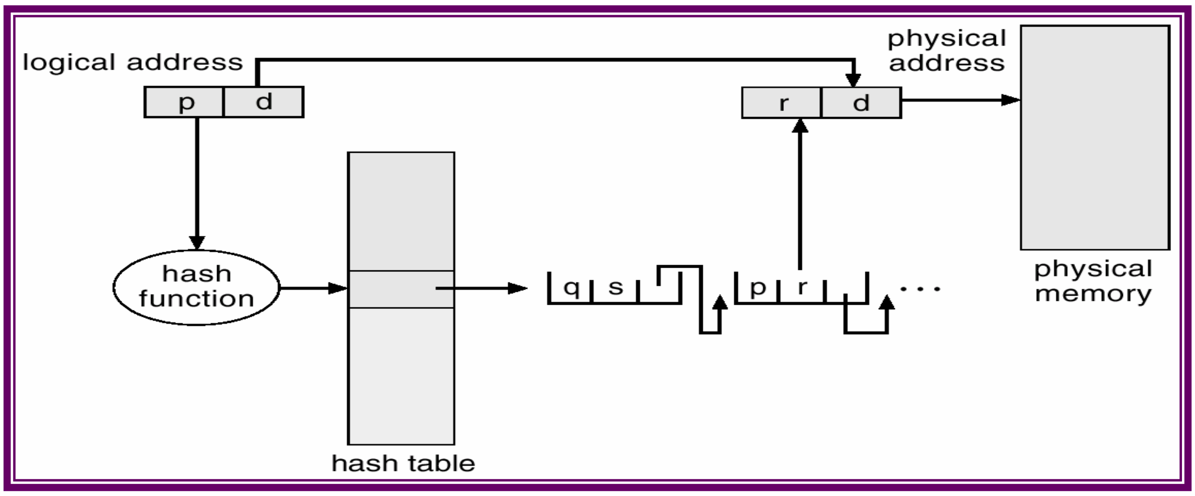
2nd outer page	outer page	inner page	offset
p1	*p2*	*p3*	*d*
32	10	10	12

4.2 Hashed Page Tables:

A common approach for handling address space 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 linked list of elements that hash to the same location. 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.

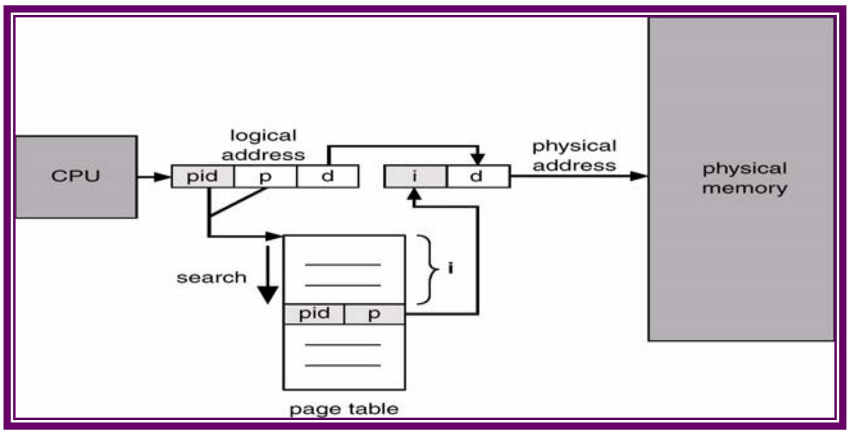
The algorithm works as follows: the virtual page number in the virtual address is hashed into hash table. The virtual page number is compared with field 1 in the first element in the linked list. If there is a match, the corresponding page frame is used to form the desired physical address. If there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

The variation of this scheme that is favorable for 64 bit address space 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 rather than a single page. Therefore, a single page table entry can store mappings for multiple physical page frames. Clustered page table are particularly useful for sparse address spaces, where memory references are noncontiguous and scattered throughout the address space.



4.3 Inverted Page Table

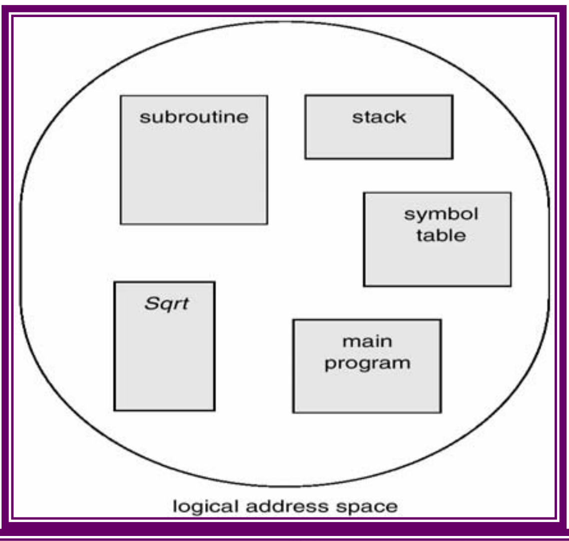
Usually, each process has a page table associated with it. The page table has one entry for each page that the process is using. This table representation is a natural one, since processes reference pages through the pages' virtual addresses. The operating system must then translate this reference into a physical memory address. Since the table is sorted by virtual address, the operating system is able to calculate where in the table the associated physical-address entry is, and to use that value directly. One of the drawbacks of this scheme is that each page table may consist of millions of entries. These tables may consume large amounts of physical memory, which is required just to keep track of how the 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 (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, there is only one page table in the system, and it has only one entry for each page of physical memory. Inverted page tables often require that an address space identifier be stored in each entry of the page table, since the table usually contains several different address spaces mapping physical memory.

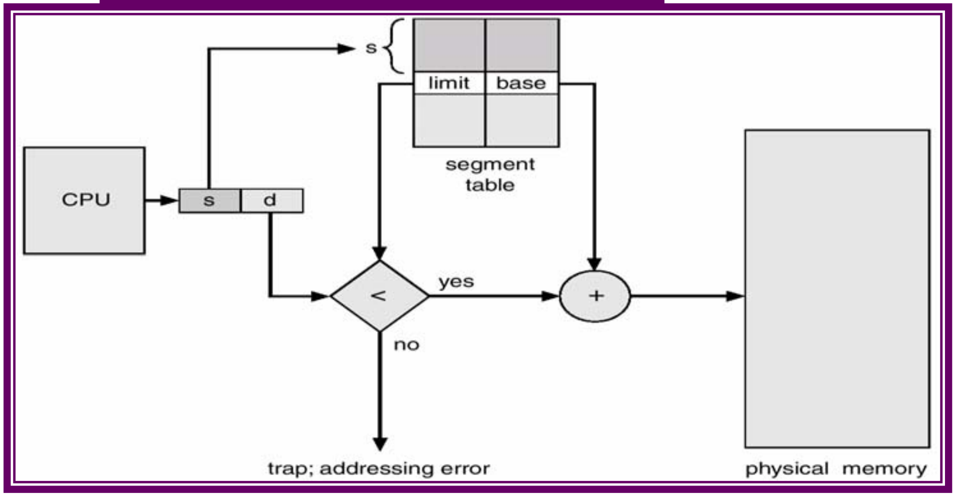


5 Segmentation:

5.1 Basic Method

The user prefers to view memory as a collection of variable-sized segments, with no necessary ordering among segments. Consider how you think of a program when you are writing it. You think of it as a main program with a set of subroutines, procedures, functions, or modules. There may also be various data structures: tables, arrays, stacks, variables, and so on. Each of these modules or data elements is referred to by name. You talk about "the symbol table," "function Sqrt" "the main program," without caring what addresses in memory these elements occupy. 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.



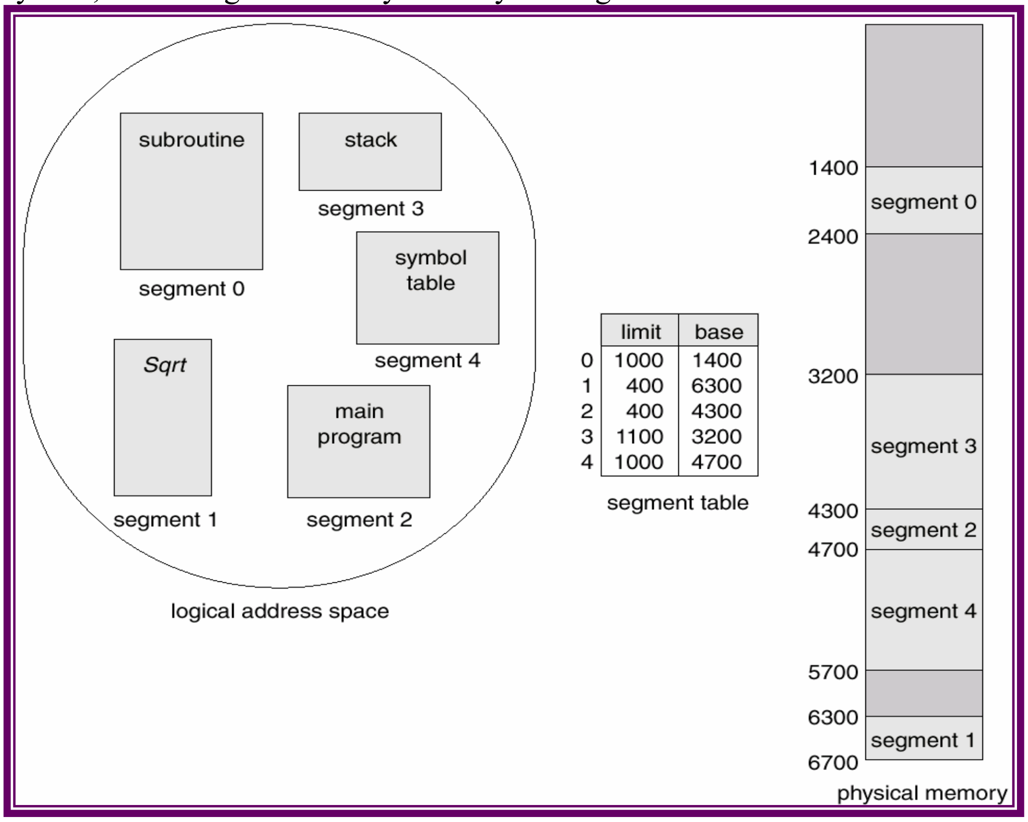


Thus, a logical address consists of a two tuple: *<segment-number, offset>*

Normally, the user program is compiled, and the compiler automatically constructs segments reflecting the input program. A C compiler might create separate segments for

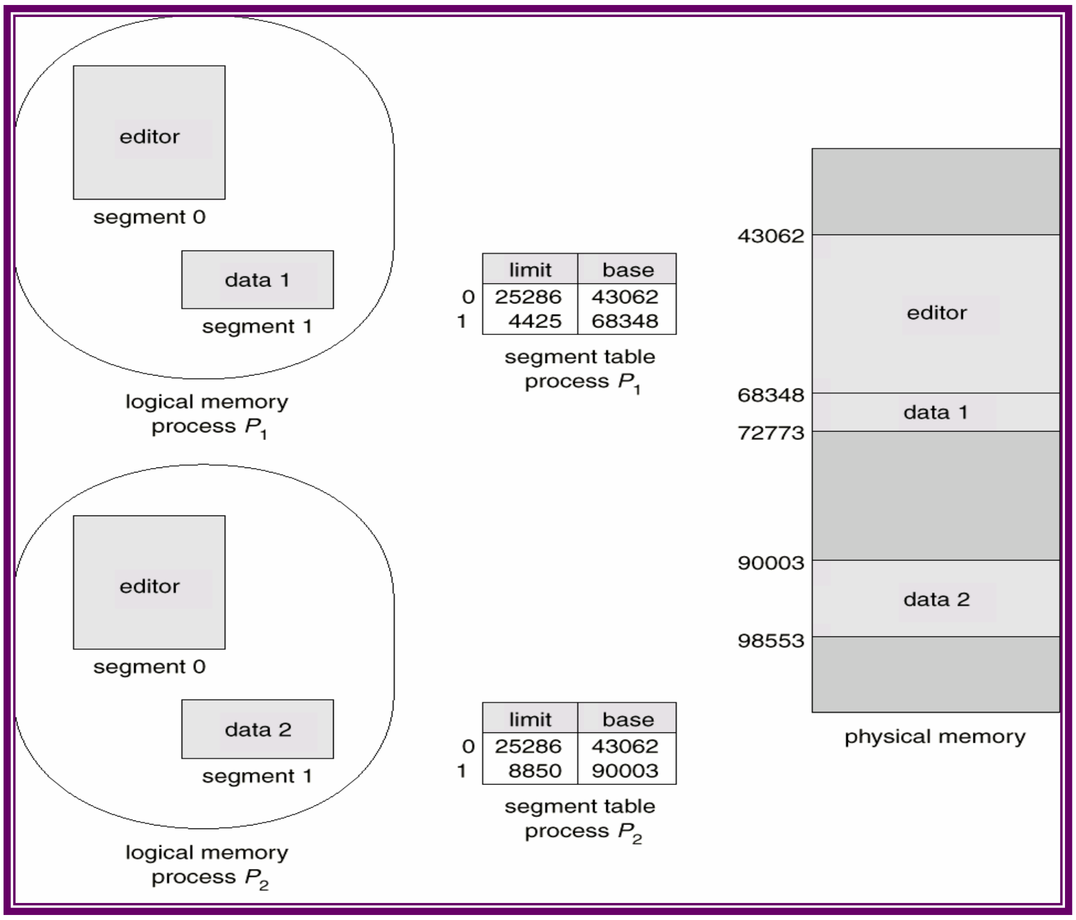
1) The code
2) The global variables
3) The heap, from which memory is allocated
4) The stacks used by each thread
5) The standard C library

Libraries that are linked in during compile time might be assigned separate segments. The loader would take all these segments and assign them segment numbers. We have five segments numbered from 0 through 4. The segments are stored in physical memory as shown. The segment table has a separate entry for each segment, giving the beginning address of the segment in physical memory (the base) and the length of that segment (the limit). For example, segment 2 is 400 bytes long, and begins at location 4300. Thus, a reference to byte 53 of segment 2 is mapped onto location 4300 + 53 = 4353. A reference to segment 3, byte 852, is mapped to 3200 (the base of segment 3) + 852 = 4052. A reference to byte 1222 of segment 0 would result in a trap to the operating system, as this segment is only 1000 bytes long.



	limit	base
0	1000	1400
1	400	6300
2	400	4300
3	1100	3200
4	1000	4700

Sharing of segments



****

VIRTUAL MEMORY

Virtual memory is a technique that allows the execution of processes that may not be completely in memory. The main visible advantage of this scheme is that programs can be larger than physical memory.

1 Background:

The instructions being executed must be in physical memory. The first approach to meeting this requirement is to place the entire logical address space in physical memory. Overlays and dynamic loading can help ease this restriction, but they generally require special precautions and extra effort by the programmer. An examination of real programs shows us that, in many cases, the entire program is not needed. For instance,

- Programs often have code to handle unusual error conditions.
- Arrays, lists, and tables are often allocated more memory than they actually need.
- Certain options and features of a program may be used rarely.

The ability to execute a .program that is only partially in memory would have many benefits:

- A program would no longer be constrained by the amount of physical memory that is available. Users would be able to write programs for an extremely large virtual address space, simplifying the programming task.
- 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.

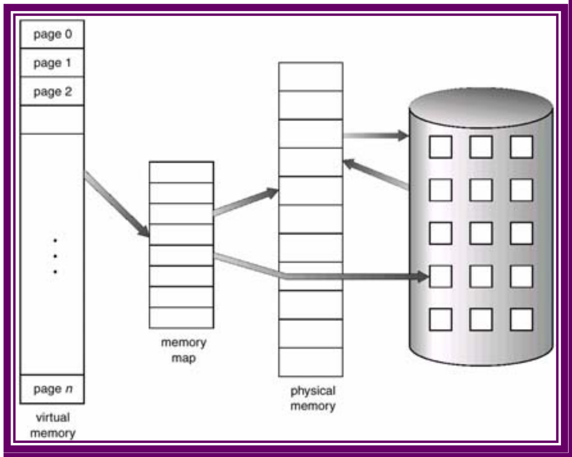
- Less I/O would be needed to load or swap each user program into memory, so each user program would run faster.

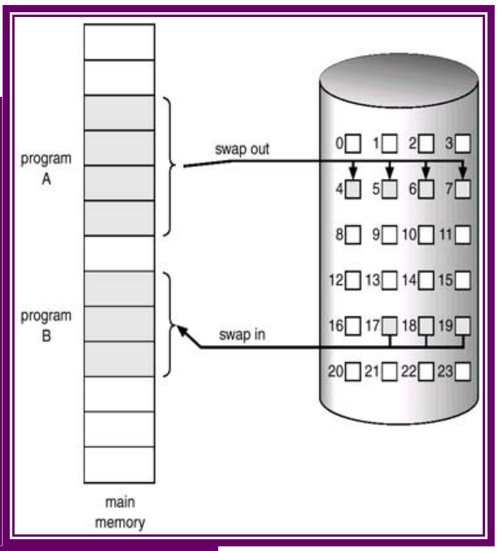
Virtual memory is the separation of user logical memory from physical memory. This separation allows an extremely large virtual memory to be provided for programmers when only a smaller physical memory is available

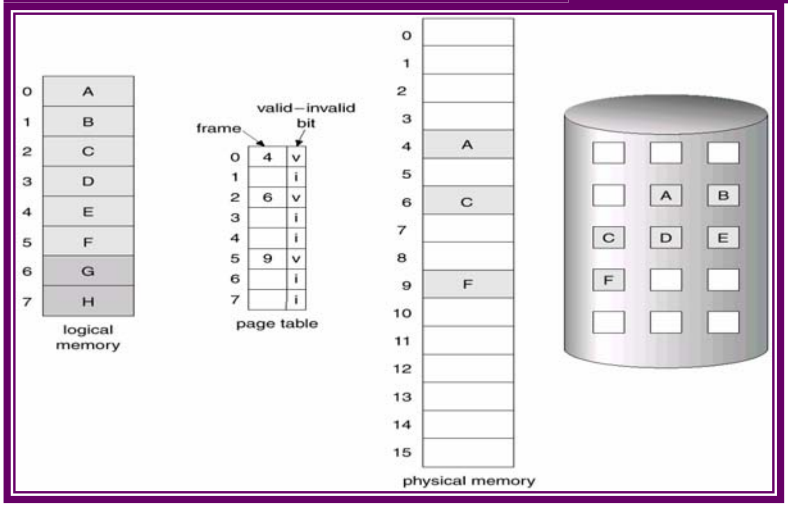
Virtual memory is commonly implemented by demand paging. It can also be implemented in a segmentation system. The user view is segmentation, but the operating system can implement this view with demand paging. Demand segmentation can also be used to provide virtual memory.

2 Demand Paging

A demand-paging system is similar to a paging system with swapping Processes reside on secondary memory (which is usually a disk). When we want to execute a process, we swap it into memory. Rather than swapping the entire process into memory, however, we use a lazy swapper. A lazy swapper never swaps a page into memory unless that page will be needed.

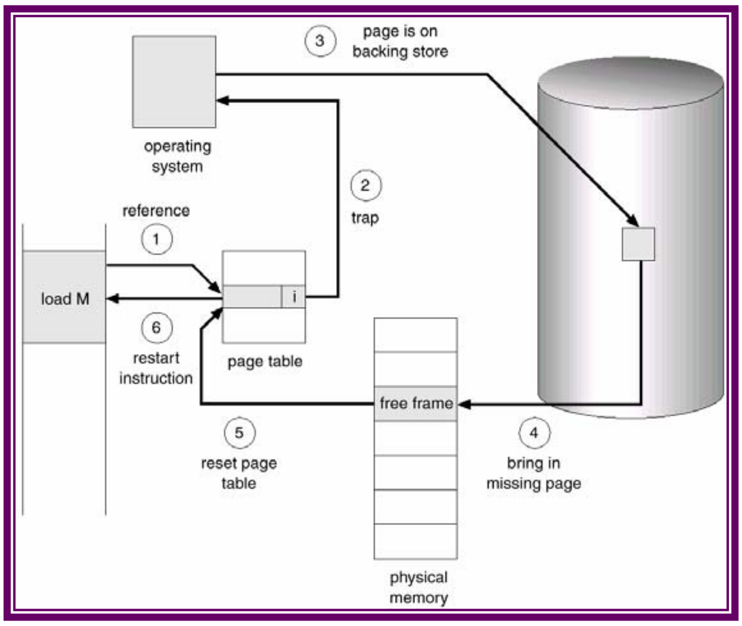






A swapper manipulates entire processes, whereas a pager is concerned with the individual pages of a process. We shall thus use the term pager, rather than swapper, in connection with demand paging. Instead of swapping in a whole process, the pager brings only those necessary pages into memory. Thus, avoids reading into memory pages that will not be used anyway decreasing the swap time and the amount of physical memory needed. With this scheme, we need some form of hardware support to distinguish between those pages that are in memory and those pages that are on the disk.

The valid-invalid bit scheme when this bit is set to "valid," this value indicates that the associated page is both legal and in memory. If the bit is set to "invalid," this value indicates that the page either is not valid or is valid but is currently on the disk. While the process executes and accesses pages that are memory resident, execution proceeds normally. If the process tries to use a page that was not brought into memory, Access to a page marked invalid causes a page-fault trap. This trap is the result of the operating system's failure to bring the desired page into memory. The procedure for handling this page fault is simple.



1. We check an internal table for this process, to determine whether the reference was a valid or invalid memory access.
2. 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 in latter.
3. We find a free frame (by taking one from the free-frame list).
4. We schedule a disk operation to read the desired page into the newly allocated frame.
5. 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.
6. We restart the instruction that was interrupted by the illegal address trap.

The process can now access the page as though it had always been in memory. It is important to realize that, because we save the state (registers, condition code, instruction counter) of the interrupted process when the page fault occurs, we can restart the process in exactly the same place and state except that the desired page is now in memory and is accessible. In the extreme case, we could 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 non-memory-resident page, the process would immediately fault for the page. After this page was brought into memory, the process would continue to execute, faulting as necessary until every page that it needed was actually in memory. At that point, it could execute with no more faults. This scheme is pure demand paging: Never bring a page into memory until it is required.
The hardware to support demand paging is the same as the hardware for paging and swapping:

- Page table: This table has the ability to mark an entry invalid through a valid-invalid bit or special value of protection bits.
- Secondary memory: This memory holds those pages that are not present in main memory. The secondary memory is usually a high-speed disk. It is known as the swap device, and the section of disk used for this. Purpose is known as swap space or backing store.

A crucial issue is the need to be able to restart any instruction after a page fault. 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 re-fetch the instruction, decode it again, and then fetch the operand. As a worst case, consider a three-address instruction such as ADD the content of A to B placing the result in C. The steps to execute this instruction would be:

1. Fetch and decode the instruction (ADD).
2. Fetch A.
3. Fetch B.
4. Add A and B.
5. Store the sum in C.

If we faulted when we tried to store in C, we would have to get the desired page, bring it in, correct the page table, and restart the instruction. The restart would require fetching the instruction again, decoding it again, fetching the two operands again, and then adding again. The major difficulty occurs when one instruction may modify several different locations. This problem can be solved in two different ways. In one solution, the microcode computes and attempts to access both ends of both blocks. If a page fault is going to occur, it will happen at this step, before anything is modified. The move can then take place, as we know that no page fault can occur, since all
the relevant pages are in memory. The other solution uses temporary registers to hold the values of overwritten locations. If there is a page fault, all the old values are written back into memory before the trap occurs. This action restores memory to its state before the instruction was started, so that the instruction can be repeated.

3 Performance of Demand Paging

Demand paging can have a significant effect on the performance of a computer system. Let us compute the effective access time for a demand-paged memory. The memory access time, ma, 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, Let p be the probability of a page fault ($0 < p < 1$). We would expect p to be close to zero that is there will be only a few page faults. The effective access time is then

Effective Access Time = $(1 - p) \times ma + p \times$ page fault time.

A page fault causes the following sequence to occur:

1. Trap to the operating system.
2. Save the user registers and process state.
3. Determine that the interrupt was a page fault.
4. Check that the page reference was legal and determine the location of the page on the disk.
5. Issue a read from the disk to a free frame:
    a. Wait in a queue for this device until the read request is serviced.
    b. Wait for the device seek and/or latency time.
    c. Begin the transfer of the page to a free frame.
6. While waiting, allocate the CPU to some other user (CPU scheduling optional).
7. Interrupt from the disk (I/O completed).
8. Save the registers and process state for the other user (if step 6 executed).
9. Determine that the interrupt was from the disk.

10. Correct the page table and other tables to show that the desired page is now in memory.
11. Wait for the CPU to be allocated to this process again.
12. Restore the user registers, process state, and new page table, and then resume the interrupted instruction.

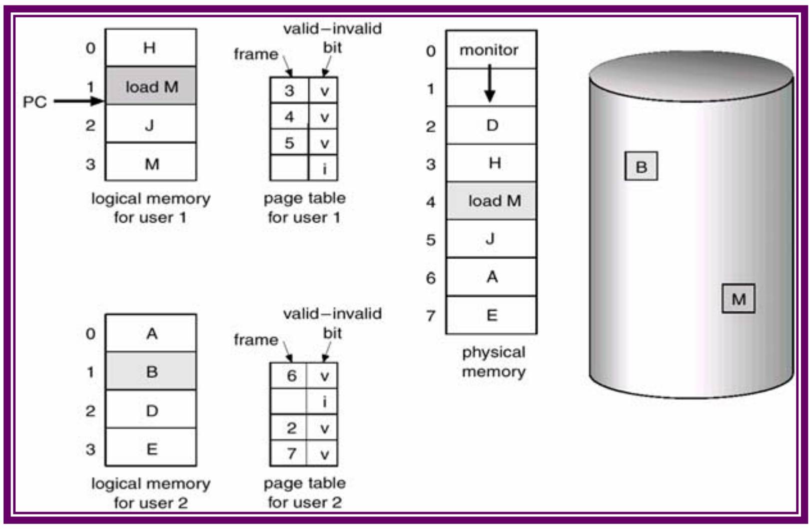
In any case, we are faced with three major components of the page-fault service time:

1. Service the page-fault interrupt.
2. Read in the page.
3. Restart the process.

The first and third tasks may be reduced, with careful coding, to several hundred instructions. One additional aspect of demand paging is the handling and overall use of swap space. Disk I/O to swap space is generally faster than that to the file system. It is faster because swap space is allocated in much larger blocks, and file lookups and indirect allocation methods are not used.

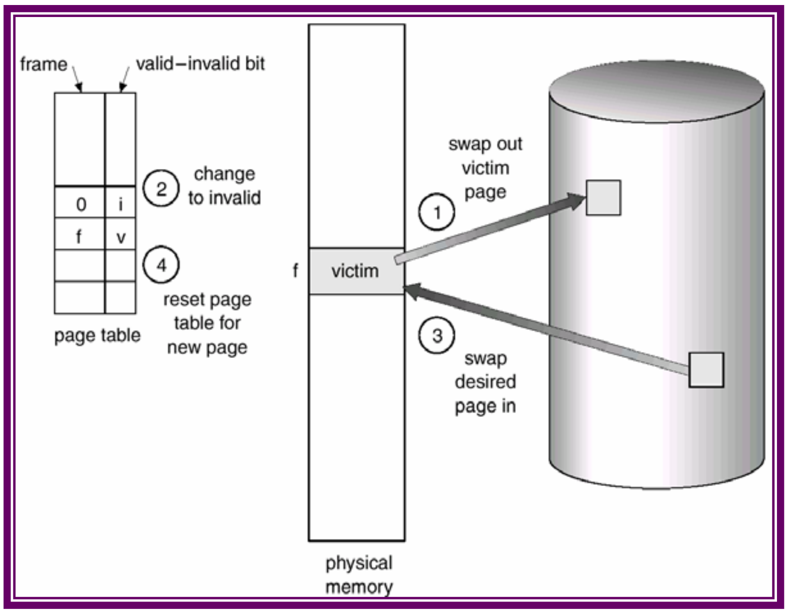
4 Page Replacement

If we increase our degree of multi programming, we are over-allocating memory. If we run six processes, each of which is 10 pages in size, but actually uses only five pages, we have higher CPU utilization and throughput, with 10 frames to spare. It is possible, however, that each of these processes, for a particular data set, may suddenly try to use all 10 of its pages, resulting in a need for 60 frames, when only 40 are available. Over-allocating will show up in the following way. While a user process is executing, a page fault occurs. The hardware traps to the .operating system, which checks its internal tables to see that this is a page fault and not an illegal memory access. The operating system determines where the desired page-is residing on the disk, but then finds there are no free frames on the free-frame list, all memory is in use. The operating system has several options at this point. It could terminate the user process. However, demand paging is something that the operating system is doing to improve the computer system's utilization and throughput. We could swap out a process, freeing all its frames, and reducing the level of multi programming.



Page replacement takes the following approach. If no frame is free, we find one that is not currently being used and free it. We can free a frame by writing its contents to swap space, and changing the page table (and all other tables) to indicate that the page is no longer in memory The freed frame can now be used to hold the page for which the process faulted. The page-fault service routine is now modified to include page replacement:

1. Find the location of the desired page on the disk.
2. Find a free frame:
    a. If there is a free frame, use it.
    b. Otherwise, use a page-replacement algorithm to select a victim frame.
    c. Write the victim page to the disk; change the page and frame tables accordingly.
3. Read the desired page into the (newly) free frame; change the page and frame tables.
4. Restart the user process.



If no frames are free, two page transfers (one out and one in) are required. This situation effectively doubles the page-fault service time and will increase the effective access time accordingly. This overhead can be reduced by the use of a modify (dirty) bit. Each pager frame may have a modify bit associated with it in the hardware. The modify bit for a page is set by the hardware whenever any word or byte in the page is written into, indicating that the page has been modified. When we select a page for replacement, we examine it's modify bit. If the bit is set, we know that the page has been modified since it was read in from the disk. In this case, we must write that page to the disk. If the modify bit is not set, however, the page has not been modified since it was read into memory. Therefore, If the copy of the page on the disk has not been overwritten. Page replacement is basic to demand paging. It completes the separation between logical memory and physical memory.

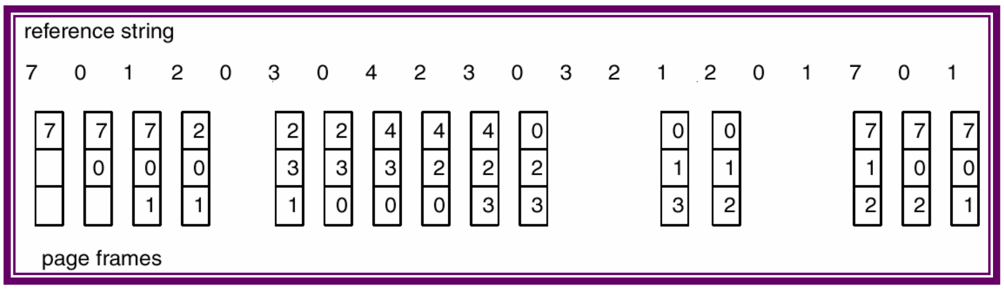
With non-demand paging, user addresses were mapped into physical addresses, allowing the two sets of addresses to be quite different. All of the pages of a process still must be in physical memory, however. With demand paging, the size of the logical address space is no longer constrained by physical memory. We must solve two major problems to implement demand paging: We must develop a frame-allocation algorithm and a page-replacement algorithm. If we have multiple processes in memory, we must decide how many frames to allocate to each process.

5 Page-Replacement Algorithms

Evaluation of an algorithm can be done 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. To reduce the number of data, we note two things. First, for a given page size we need to consider only the page number, not the entire address. Second, if we have a reference to a page p, then any immediately following references to page p will never cause a page fault. Page p will be in memory after the first reference; the immediately following references will not fault.

5.1 FIFO Algorithm

The simplest page-replacement algorithm is a FIFO 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. Notice that it is not strictly necessary to record the time when a page is brought in. Consider example reference string: 7, 0,1,2,0,3,0,4, 2, 3, 0, 3,2,1,2, 0, 1, 7, 0, 1



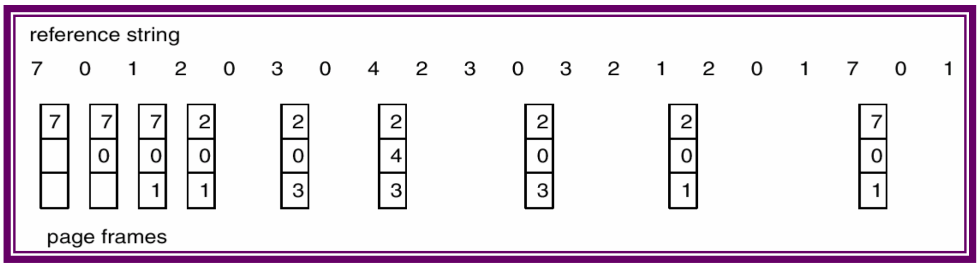
Initially three frames are empty. The first three references (7, 0, 1) cause page faults, and are brought into these empty frames. The next reference (2) replaces page 7, because page 7 was brought in first. The FIFO page-replacement algorithm is easy to understand and program. However, its performance is not always good.
The problems that are possible with a FIFO page-replacement algorithm, we consider the reference string:

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

We notice that the number of faults for four frames (10) is greater than the number of faults for three frames (nine). This result is most unexpected and is known as Belady's anomaly. Belady's anomaly reflects the fact that, for some page-replacement algorithms, the page-fault rate may increase as the number of allocated frames increases. We would expect that giving more memory to a process would improve its performance. In some early research, investigators noticed that this assumption was not always true. Belady's anomaly was discovered as a result.

5.2 Optimal Algorithm

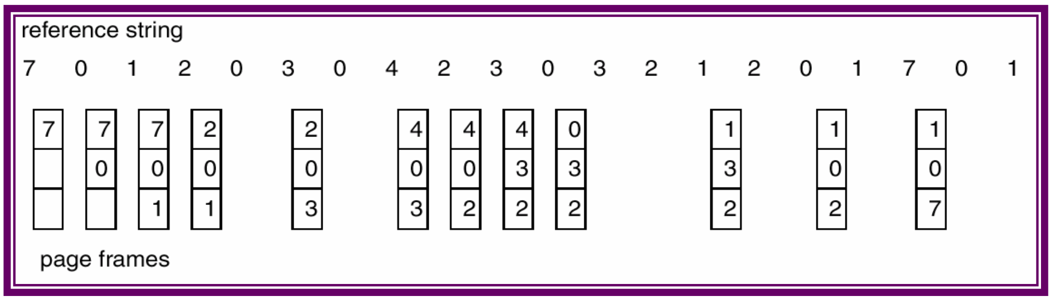
One result of the discovery of Belady's anomaly was the search for an optimal page-replacement algorithm. An Optimal page-replacement algorithm has the lowest page-fault rate of all algorithms. An optimal page-replacement algorithm exists, and has been called OPT or MTN. Replace the page that will not 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. Consider example reference string: 7, 0,1,2,0,3,0,4, 2, 3, 0, 3,2,1,2, 0, 1, 7, 0, 1



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.

5.3 LRU Algorithm

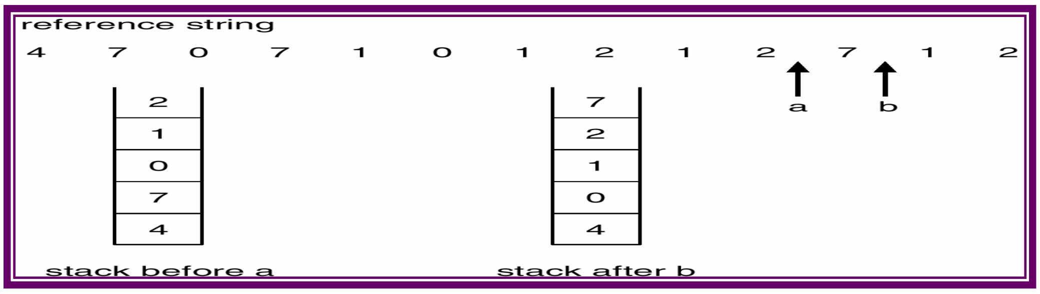
If we use the recent past as an approximation of the near future, then we will replace the page that has not been used for the longest period of time. This approach is the least recently used (LRU) algorithm. LRU replacement associates with each page the time of that page's last use. When a page must be replaced, LRU chooses that page that has not been used for the longest period of time. This strategy is the optimal page replacement algorithm looking backward in time, rather than forward.



When the reference to page 4 occurs, however, LRU replacement sees that, of the three frames in memory, page 2 was used least recently. The most recently used page is page 0, and just before that page 3 was used. 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, of the three pages in memory (0, 3, 4), page 3 is the least recently used. Despite these problems, LRU replacement with 12 faults is still much better than FIFO replacement with 15. The LRU policy is often used as a page-replacement algorithm and is considered to be quite good. The major problem is how to implement LRU replacement.

Two implementations are feasible:

- **Counters:** In the simplest case, 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. We replace the page with the smallest time value. The times must also be maintained when page tables are changed (due to CPU scheduling). Overflow of the clock must be considered.
- **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 top of the stack is always the most recently used page and the bottom is the LRU page. Because entries must be removed from the middle of the stack, it is best implemented by a doubly linked list, with a head and tail pointer.



5.4 LRU Approximation Algorithms

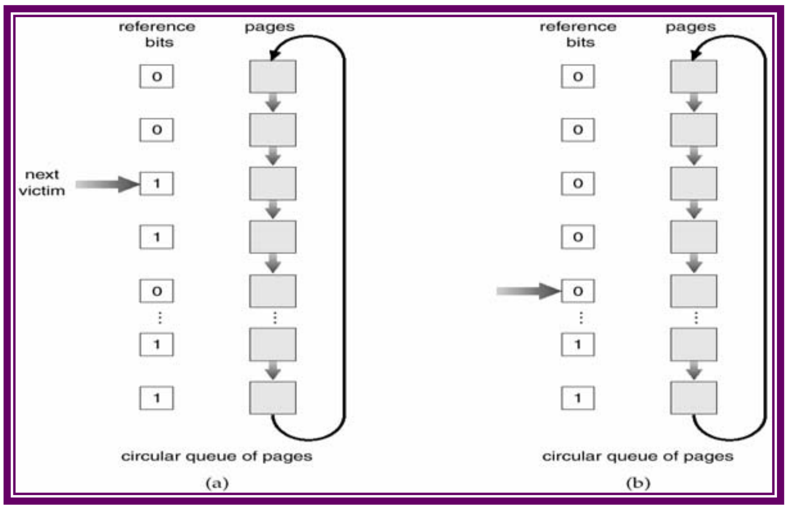
Few computer systems provide sufficient hardware support for true LRU page replacement. Some systems provide no hardware support, and other page replacement algorithms (such as a FIFO algorithm) must be used. Many systems provide some help, however, in the form of a reference bit. The reference bit for a page is set, by the hardware, whenever that page is referenced (either a read or a write to any byte in the page). Reference bits are associated with each entry in the page table. Initially, all bits are cleared (to 0) by the operating system. As a user process executes, the bit associated with each page referenced is set (to 1) by the hardware.

5.4.1 Additional-Reference-Bits Algorithm

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, shifting the other bits right 1 bit, discarding the low-order bit. These 8-bit shift registers contain the history of page use for the last eight time periods. If the shift register contains 00000000, then the page has not been used for eight time periods; a page that is used at least once each period would have a shift register value of 11111111.

5.4.2 Second-Chance Algorithm

When a page has been selected, however, we inspect its reference bit. If the value is 0, we proceed to replace this page. If the reference bit is 1, however, we give that 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 time is reset to the current time. Thus, a page that is given a second chance will not be replaced until all other pages are 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. As it advances, it clears the reference bits. Once a victim page is found, the page is replaced and the new page is inserted in the circular queue in that position.



5.4.3 Enhanced Second-Chance Algorithm

The second-chance algorithm described above can be enhanced by considering both the reference bit and the modify bit (Section 9.4) as an ordered pair. With these 2 bits, we have the following four possible classes:

1. (0,0) neither recently used nor modified—best page to replace
2. (0,1) not recently used but modified—not quite good, because the page will need to be written out before replacement
3. (1,0) recently used but clean—probably will be used again soon
4. (1,1) recently used and modified—probably will be used again, and write out will be needed before replacing it

5.5 Counting Algorithms

There are many other algorithms that can be used for page replacement:

LFU Algorithm: 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.

MFU Algorithm: 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.

5.6 Page Buffering Algorithm

Other procedures are often used in addition to a specific page-replacement algorithm. For example, systems commonly keep a pool of free frames. When a page fault occurs, a victim frame is chosen as before. However, the desired page is read into a free frame from the pool before the victim is written out. This procedure allows the process to restart as soon as possible, without waiting for the victim page to be written out. When the victim is later written out, its frame is added to the free-frame pool. Another modification is to keep a pool of free frames, but to remember which page was in each frame. Since the frame contents are not modified when a frame is written to the disk, the old page can be reused directly from the free-frame pool if it is needed before that frame is reused.

6 Allocation of Frames

Operating system allocates all its buffer and table space from the free-frame list. When this space is not in use by the operating system, it can be used to support user paging. We could try to keep three free frames reserved on the free-frame list at all times. Thus, when a page fault occurs, there is a free frame available to page into. While the page swap is taking place, a replacement can be selected, which is then written to the disk as the user process continues to execute.

6.1 Minimum Number of Frames

There is also a minimum number of frames that can be allocated. Obviously, as the number of frames allocated to each process decreases, the page fault-rate increases, slowing process execution. Besides the undesirable performance properties of allocating only a few frames, there is a minimum number of frames that must be allocated. This minimum number is defined by the instruction-set architecture. The minimum number of frames is defined by the computer architecture. The worst-case scenario occurs in architectures that allow multiple levels of indirection. When the first indirection occurs, a counter is set to 16; the counter is then decremented for each successive indirection for this instruction. If the counter is decremented to 0, a trap occurs (excessive indirection). This limitation reduces the maximum number of memory references per instruction to 17, requiring the same number of frames.

6.2 Allocation Algorithms

The easiest way to split m 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 could 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. If a small student process of 10K and ah interactive database of 127K are the only two processes running in a system with 62 free frames, it does not make much sense to give each process 31 frames. The student process does not need more than 10 frames, so the other 21 are strictly wasted.

Proportional Allocation: 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 fl, frames to process Pi, where ai is approximately: $fl_f=Si/S \text{ x } m$

In both equal and proportional allocation, of course, the allocation to each process may vary according to the multi programming level. If the multi programming level is increased, each process will lose some frames to provide the memory needed for the new process. On the other hand, if the multi programming level decreases, the frames that had been allocated to the departed process can now be spread over the remaining processes

6.3 Global Versus Local Allocation

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; one process can take a frame from another. Local replacement requires that each process select from only its own set of allocated frames.

For example, consider an allocation scheme where we allow high-priority processes to select frames from low-priority processes for replacement. 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 the 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 frames allocated to it.

Under local replacement, the set of pages in memory for a process is affected by the paging behavior of only that process. 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.

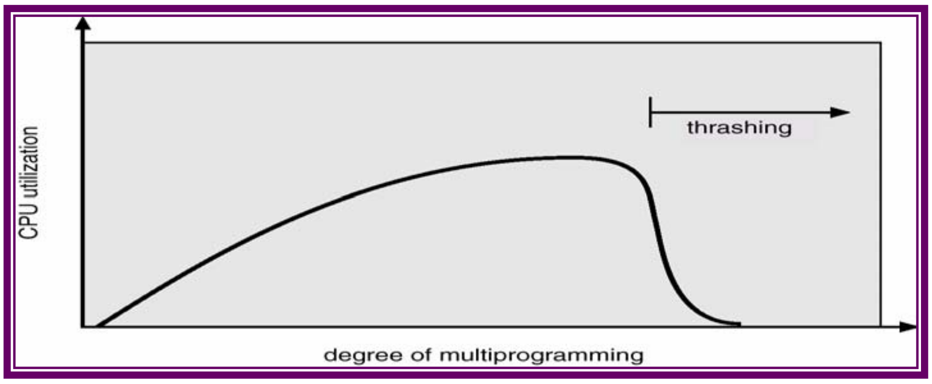
7 Thrashing

High paging activity is called thrashing. A process is thrashing if it is spending more time paging than executing.

7.1 Cause of Thrashing

The operating system monitors CPU utilization. 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, replacing pages with no regard to the process to which they belong. Now suppose 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.



CPU utilization is plotted against the degree of multiprogramming. As the degree of multiprogramming increases, CPU utilization also increases, although more slowly, until a maximum is reached. If the degree of multi programming 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 multi programming.

The effects of thrashing can be limited by using a local (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 also. To prevent thrashing, we must provide a process as many frames as it needs. But how do we know how many frames it "needs"? There are several techniques. The working-set strategy starts by looking at how many frames a process is actually using. This approach defines the locality model of process execution.

For example, when a subroutine is called, it defines a new locality. In this locality, memory references are made to the instructions of the subroutine, its local variables, and a subset of the global variables. When the subroutine is exited, the process leaves this locality, since the local variables and instructions of the subroutine are no longer in active use. We may return to this locality later. Thus, we see that localities are defined by the program structure and its data structures. The locality model states that all programs will exhibit this basic memory reference structure. The locality model states that, as a process executes, it moves from locality to locality. A locality is a set of pages that are actively used together. A program is generally composed of several different localities, which may overlap.

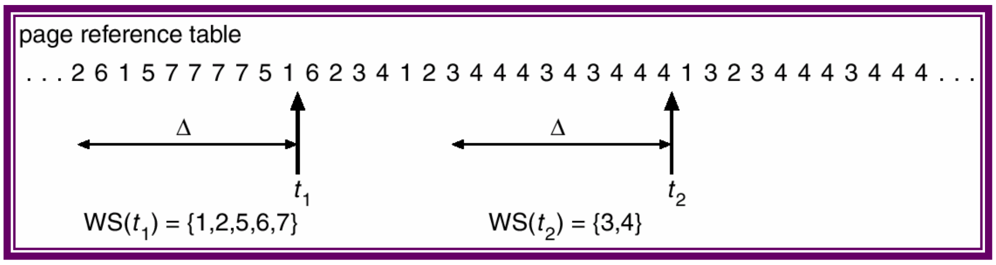
7.2 Working-Set Model

The working-set model is based on the assumption of locality. This model uses a parameter A to define the working-set window. The idea is to examine the most recent A page references. The set of pages in the most recent A page references is the working set. If a page is in active use, it will be in the working set. If it is no longer being used, it will drop from the working set A time units after its last reference. Thus, the working set is an approximation of the program's locality.

The most important property of the working set is its size. If we compute the working-set size WSS_i, for each process in the system, we can then consider:

$D = \sum WSS_i$ Where D is the total demand for frames. Each process is actively using the pages in its working set. Thus, process i needs WSS; frames. If the total demand is greater than the total number of available frames (D > ra), thrashing will occur, because some processes will not have enough frames.

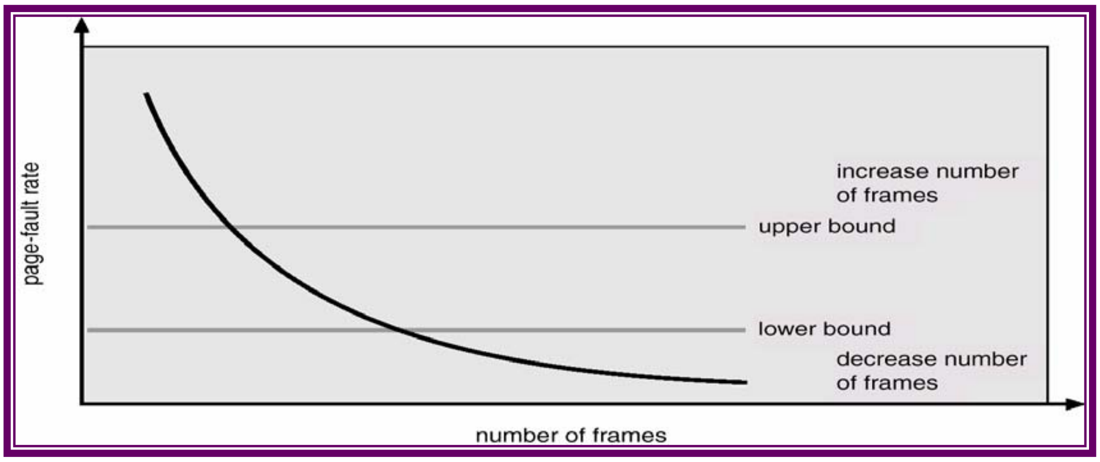
The operating system monitors the working set of each process and allocates to that working set enough frames to provide it with its working-set size. If there are enough extra frames, another process can be initiated. If the sum of the working-set sizes increases, exceeding the total number of available frames, the operating system selects a process to suspend. The process' pages are written out and its frames are reallocated to other processes. The suspended process can be restarted later.



This working-set strategy prevents thrashing while keeping the degree of multiprogramming as high as possible. Thus, it optimizes CPU utilization. The difficulty with the working-set model is keeping track of the working set. The working-set window is a moving window. At each memory reference, a new reference appears at one end and the oldest reference drops off the other end. We can approximate the working-set model with a fixed interval timer interrupt and a reference bit.

7.3 Page-Fault Frequency

The page-fault frequency (PFF) strategy takes a more direct approach. The specific problem is how to prevent thrashing. Thrashing has a high page-fault rate. Thus, we want to control the page-fault rate. When it is too high, we know that the process needs more frames. Similarly, if the page-fault rate is too low, then the process may have too many frames. We can establish upper and lower bounds on the desired page-fault rate. If the actual page-fault rate exceeds the upper limit, we allocate that process another frame; if the page-fault rate falls below the lower limit, we remove a frame from that process. Thus, we can directly measure and control the page-fault rate to prevent thrashing.



****

FILE-SYSTEM INTERFACE

The file system consists of two distinct parts: a collection *of files,* each storing related data, and a *directory structure,* which organizes and provides information about all the files in the system. Some file systems have a third part, *partitions,* which are used to separate physically or logically large collections of directories.

1 File Concept

Computers can store information on several different storage media, such as magnetic disks, magnetic tapes, and optical disks. The operating system abstracts from the physical properties of its storage devices to define a logical storage unit, *the file.* Files are mapped, by the operating system, onto physical devices. These storage devices are usually *nonvolatile,* so the contents are persistent through power failures and system reboots. A file is a named collection of related information that is recorded on secondary storage. Files may be free-form, such as text files, or, may be formatted rigidly. In general, a file is a sequence of bits, bytes, lines, or records whose meaning is defined by the file's creator and user. Many different types of information may be stored in a file: source programs, object programs, executable programs, numeric data, text, payroll records, graphic images, sound recordings, and so on. A file has a certain defined *structure* according to its type. A *text* file is a sequence of characters organized into lines.

1.1 File Attributes

A file is named, for the convenience of its human users, and is referred to by its name. A name is usually a string of characters, such as "example.c". When a file is named, it becomes independent of the process, the user, and even the system that created it. A file has certain other attributes, which vary from one operating system to another, but typically consist of these:

- **Name**: The symbolic file name is the only information kept in human readable form.
- **Type**: This information is needed for those systems that support different types.
- 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 controls 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.

1.2 File Operations

A file is an *abstract data type.* To define a file properly, we need to consider the operations that can be performed on files. The operating system provides 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. The directory entry records the name of the file and the location in the file system.
- **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 location of the file. The system must keep a *write* pointer to the location in the file where the next write is to take place. The write pointer must be updated whenever a write occurs.

- **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 directory entry, and the system needs to keep a *read* pointer to the location in the file where the next read is to take place. Once the read has taken place, the read pointer is updated. Since, in general, a file is either being read or written, most systems keep only one *current-file-position* pointer. Both the read and write operations use this same pointer, saving space and reducing the system complexity.
- **Repositioning within a file:** The directory is searched for the appropriate entry, and the current-file-position is set to a given value. Repositioning within a file does not need to involve any actual I/O. 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 and erase the directory entry.
- **Truncating a file:** There are occasions when the user wants the attributes of a file to remain the same, but wants to erase the contents of the file. Rather than forcing the user to delete the file and then recreate it, this function allows all attributes to remain unchanged but for the file to be reset to length zero.

These six basic operations certainly comprise the minimal set of-required file operations. Other common operations include appending new information to the end of an existing file, and renaming an existing file. 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 will open a file when that file first is used actively. The operating system keeps a small table containing information about all open files.

The open operation takes a file name and searches the directory, copying the directory entry into the open-file table, assuming the file protections allow such access. The open system call will typically return a pointer to the entry in the open file table. This pointer, not the actual file name, is used in all I/O operations, avoiding any further searching, and simplifying the system-call interface.

The implementation of the open and close operations in a multi user environment, such as UNIX, is more complicated. There is a per-process table of all the files that each process has open. Stored in this table is information regarding the use of the file by the process. Typically, the open-file table also has an open count associated with each file, indicating the number of processes which have the file open. Each close decreases this count, and when the count reaches zero, the file is no longer in use, and the file's entry is removed from the open file table.

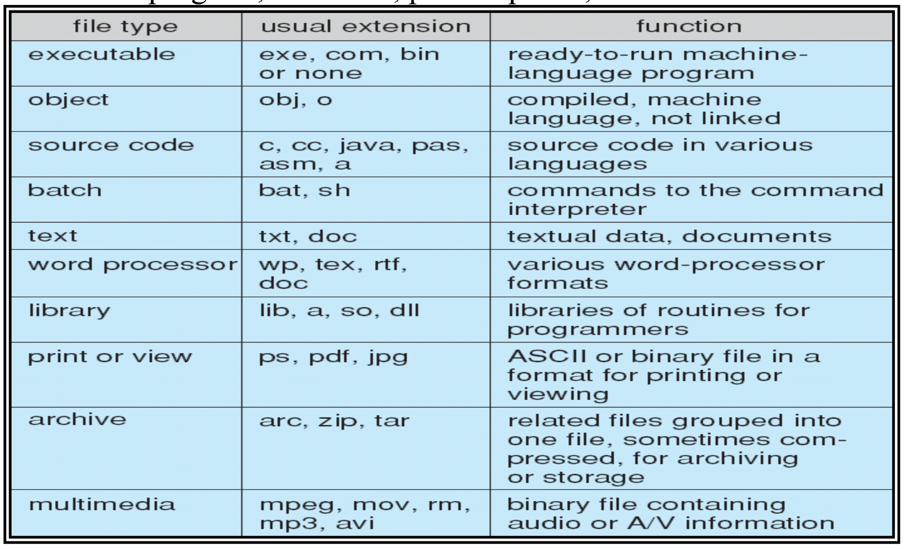
There are several pieces of information associated with an open file:

- **File pointer**: On systems that do not include a file offset as part of the read and write system calls, the system must track the last read/write location as a current-file-position pointer. This pointer is unique to each process operating on the file, and therefore must be kept separate from the on-disk file attributes.
- **File open count**: As files are closed, the operating system must reuse its open-file table entries, or it could run out of space in the table. Because multiple processes may open a file, the system must wait for the last file to close before removing the open-file table entry. This counter tracks the number of opens and closes, and reaches zero on the last close. The system can then remove the entry.
- **Disk location of the file**. Most file operations require the system to modify data within the file. The information needed to locate the file on disk is kept in memory to avoid having to read it from disk for each operation.

Some operating systems provide facilities for locking sections of an open file for multi process access, to share sections of a file among several processes, and even to map sections of a file into memory on virtual-memory systems. This last function is called memory mapping a file and allows a part of the virtual address space to be logically associated with a section of a file. Closing the file, results in all the memory-mapped data being written back to disk and removed from the virtual memory of the process.

1.3 File Types:

If an operating system recognizes the type of *a* file, it can then operate on the file in reasonable ways. 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. Only a file with a .com "," .exe or ".bat" extension can be executed. The ".com" and ".exe" files are two forms of binary executable files, whereas a ".bat" file is a *batch* file containing, in ASCII format, commands to the operating system. MS-DOS recognizes only a few extensions, but application programs also use them to indicate file types in which they are interested. Assemblers expect source files to have an ".asm" extension. Consider the Apple Macintosh operating system. In this system, each file has a type, such as "text" or "pict". Each file also has a creator attribute containing the name of the program that created it. The UNIX system is unable to provide such a feature because it uses a crude *magic number* stored at the beginning of some files to indicate roughly the type of the file: executable program, batch file, postscript file, and so on.



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 compressed, for archiving or storage
multimedia	mpeg, mov, rm, mp3, avi	binary file containing audio or A/V information

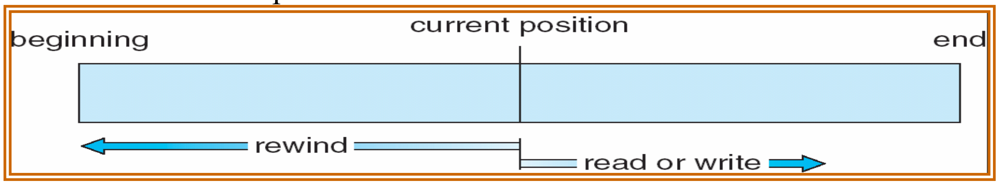
ACCESS METHODS:-

There are several ways that the information in the file can be accessed.

1) Sequential method 2) direct access method 3) other access methods.

<u>**1) Sequential access method:-**</u>

The simplest access method is Sequential Access method. Information in the file is processed in order, one after the other. The bulk of the operations on a file are reads & writes. It is based on a tape model of a file**.**

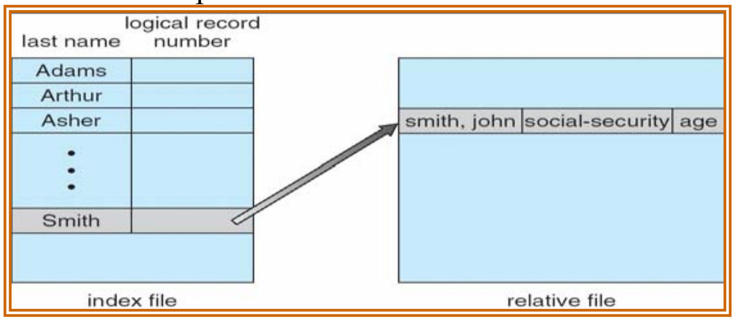


2) Direct access:- or relative access:-

a file is made up of fixed length records, that allow programs to read and write record rapidly in no particular order. For direct access, file is viewed as a numbered sequence of blocks or records. A direct access file allows, blocks to be read & write. So we may read block15, block 54 or write block10. There are no restrictions on the order of reading or writing for a direct access file. It is great useful for immediate access to large amount of information. The file operations must be modified to include the block number as a parameter. We have read n, where n is the block number.

3) Other access methods:-

The other access methods are based on the index for the file. The indexed contain pointers to the various blocks. To find an entry in the file, we first search the index and then use the pointer to access the file directly and to find the desired entry. With large files, the index file itself, may become too large to be kept in memory. One solution is to create an index for the index file. The primary index file would contain pointers to secondary index files which would point to the actual data iteams.

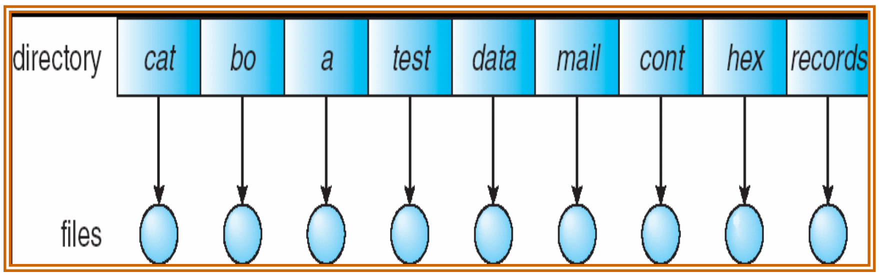


Directory structures:-

Single level directory:-

The simple directory structure is the single level directory. All files are contained in the same directory. Since all files are in same directory, they must have unique names.

In a single level directory there is some limitations. When the no.of files increases or when there is more than one user problems can occur. If the no.of files increases, it becomes difficult to remember the names of all the files.



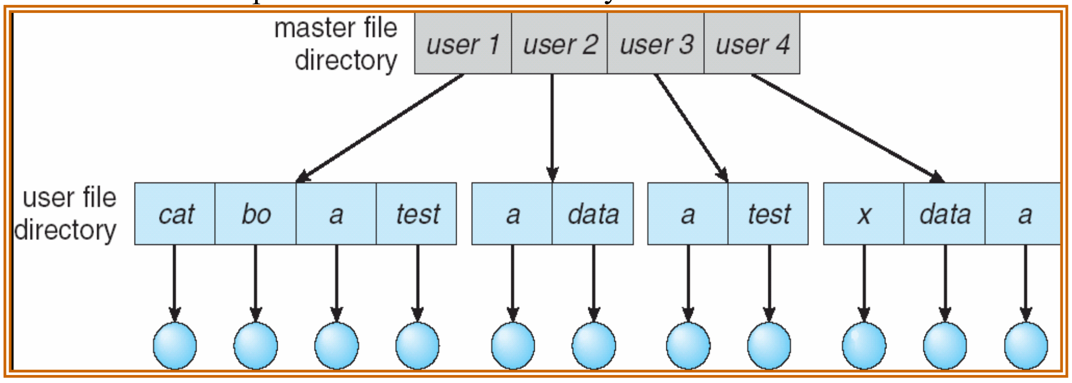
Two-level directory:-

The major disadvantages to a single level directory are the confusion of file names between different users. The standard solution is to create separate directory for each user. In 2-level directory structure, each user has his/her own user file directory (UFD). Each ufd has a similar structure, the user first search the master file directory (MFD) which is indexed by user name and each entry point to the ufd for that user.

To create a file for a user, the O.S searches only that user's ufd to find whether another file of that name exists. To delete a file the O.S only search to the local ufd and it can not accidentally delete another user's file that has the same name.

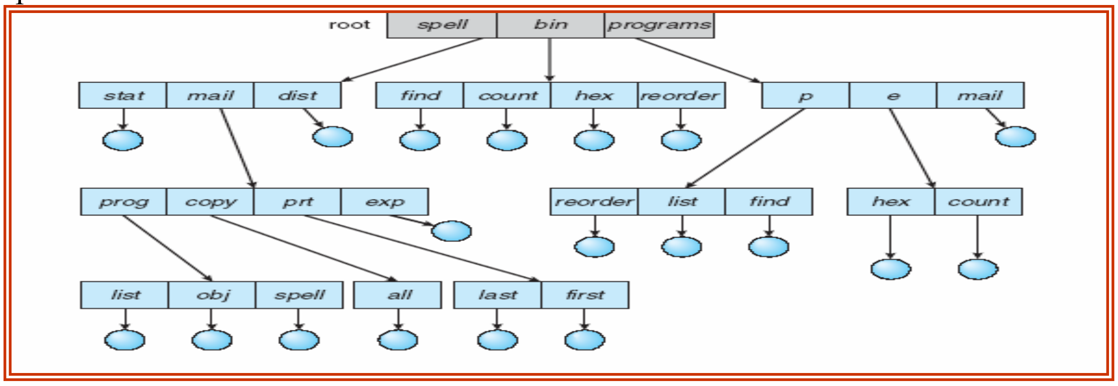
This solves the name collision problem, but it still has another. The disadvantage is when the user wants to cooperate on some task and to access one another's file. Some systems simply do not allow local user files to be accessed by other user. Any file is accessed by using path name. Here the user name and a file name define a path name.

In MS-DOS a file specification is C:/directory name/file name



Tree structured directory:-

This allows users to create their own subdirectories and to organize their files accordingly. Here the tree has a root directory. And every file in the system has a unique path name. A path name is the path from the root, through all the subdirectories to a specified-file.



A directory contains a set of subdirectories or files. A directory is simply another file, but it is treated in a special way. Here the path names can be of two types.

1) Absolute path and 2) relative path.

An absolute path name begins at the root and follows a path down to the specified file, giving the directory name on the path.

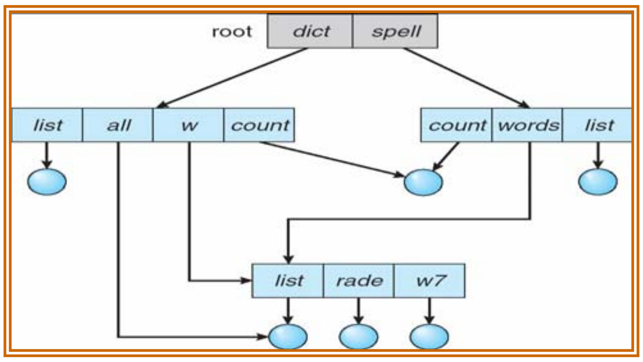
Ex:- root/spell/mail/prt/first.

A relative pathname defines a path from the current directory ex:- prt/first is relative path name.

Acyclic- graph directory:-

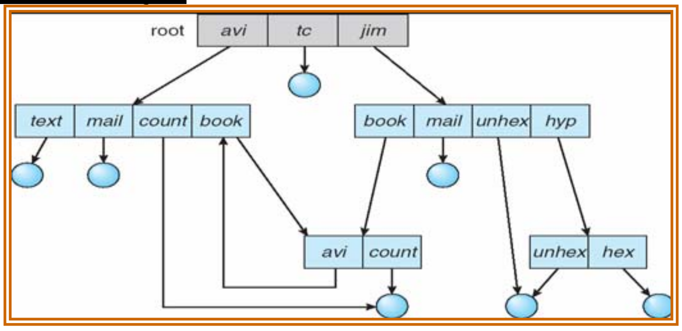
Consider two programmers who are working on a joint project. The files associated with that project can be stored in a sub directory, separating them from other projects and files of the two programmers. The common subdirectory is shared by both programmers. A shared directory or file will exist in the file system in two places at once. Notice that a shared file is not the same as two copies of the file with two copies, each programmer can view the copy rather than the original but if one programmer changes the file the changes will not appear in the others copy with a shared file there is only one actual file, so any changes made by one person would be immediately visible to the other.

A tree structure prohibits the sharing of files or directories. An acyclic graph allows directories to have shared subdirectories and files



It is more complex and more flexible. Also several problems may occur at the traverse and deleting the file contents.

General graph directory:-



A serious problem with using an acyclic-graph structure is ensuring that there are no cycles. When we add links to an existing tree-structured directory, the tree structure is destroyed, resulting in a simple graph structure. The disadvantage in acyclic graph is, if we have just searched a major shared sub directory for a particular file without finding it, we have to search it for twice. If cycles are allowed to exist in the directory, we likewise want to avoid searching any component twice. A poorly designed algorithm might result in infinite loop. To delete a file, we need to use a garbage collection scheme to determine when the last reference for a particular file is deleted and the disk space can be allocated.

Garbage collection involves traversing the entire file system, marking everything that can be accessed. Then, a second pass collects everything that is not marked onto a list of free space. Garbage collection is necessary only because of possible cycles in the graph.

****

FILE SYSTEM IMPLEMENTATION

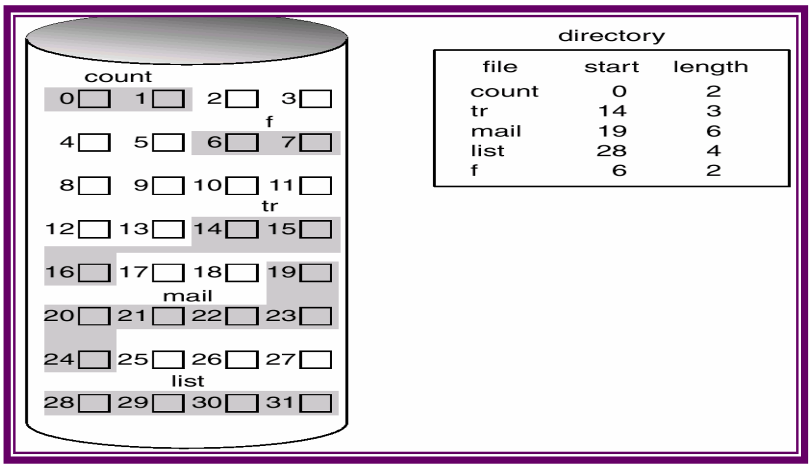
File/Disk Allocation methods:-

There are 3 major methods of allocating disk space.

Contiguous allocation:-

1) The contiguous allocation method requires each file to occupy a set of contiguous block on the disk.
2) Contiguous allocation of a file is defined by the disk address and length of the first block. If the file is 'n' block long and starts at location 'b' , then it occupies blocks b,b+1,b+2,…..,b+n-1;
3) The directory entry for each file indicates the address of the starting block and length of the area allocated for this file.
4) Contiguous allocation of file is very easy to access. For *sequential access*, the file system remembers the disk address of the last block referenced and, when necessary read 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 contagious allocation.

One difficulty with this method is finding space for a new file.



directory

file	start	length
count	0	2
tr	14	3
mail	19	6
list	28	4
f	6	2

a) **External fragmentation:- F**iles are allocated and deleted, the free disk space is broken in to little pieces. The E.F exists when free space is broken in to chunks (large piece) and these chunks are not sufficient to satisfy the request of new file. There is a solution for E.F i.e Compaction. All free blocks are moved to one side and filled blocks to another side.

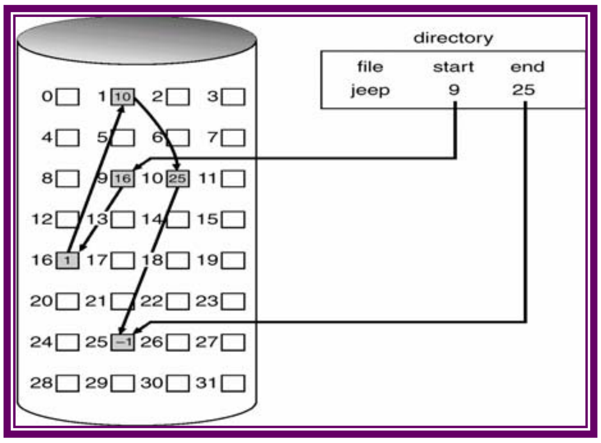
b) Another problem is determining how much space is needed for a file. When file is created the creator must specify the size of that file. This becomes a big problem. Suppose if we allocate too little space to a file, some times it may not sufficient. Suppose if we allocate large space, space may be wasted.

c) Another problem is if one large file is deleted, that large space becomes empty. Another file is loaded in to that space whose size is very small then some space is wasted. Such wastage of space is called internal fragmentation.

2) Linked allocation:-

1) Linked allocation solves all the problems of contagious allocation. With linked allocation, each file is a linked list of disk blocks, the disk block may be scattered any where on the disk.
2) The directory contains a pointer to the first and last blocks of the file.

Fig11.4

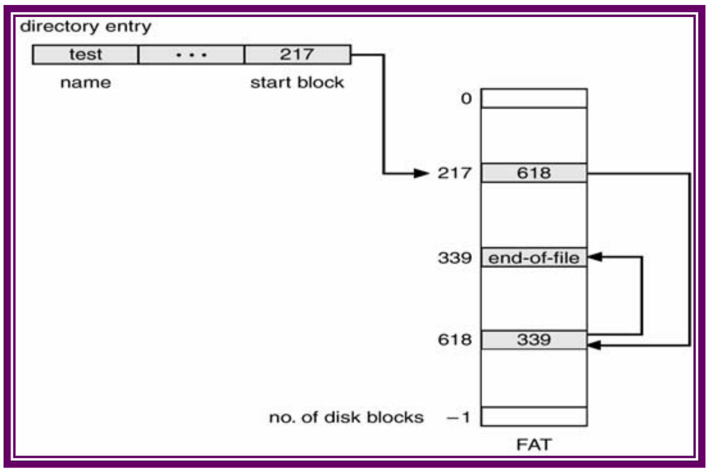


Ex:- a file have five blocks start at block 9, continue at block 16,then block 1, block 10 and finally block 25. Each block contains a pointer to the next block. These pointers are not available to the user.
3) To create a new file we simply create a new entry in directory. With linked allocation, each directory entry has a pointer to the first disk block of the file.
4) There is no external fragmentation in linked allocation. Also there is no need to declare the size of a file when that file is created. A file can continue to grow as long as there are free blocks.
5) But it have disadvantage. The major problem is that it can be used only for sequential access-files.
6) 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. It can not support the direct access.
7) Another disadvantage is it requires space for the pointers. If a pointer requires 4 bytes out of 512 byte block, then 0.78% of disk is being used for pointers, rather than for information.
8) The solution to this problem is to allocate blocks in to multiples, called clusters and to allocate the clusters rather than blocks.
9) Another problem is reliability. The files are linked together by pointers scattered all over the disk what happen if a pointer were lost or damaged.

FAT(file allocation table):-

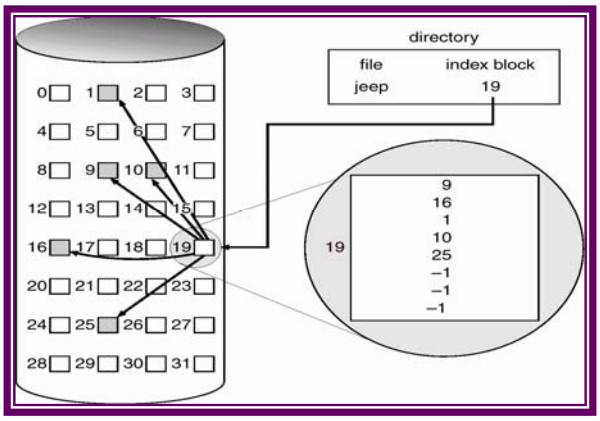
An important variation on the linked allocation method is the use of a file allocation table. The table has one entry for each disk block, and is indexed by block number. The FAT is used much as is a linked list.

The directory entry contains the block number of the first block of the file. The table entry contains the block number then contains the block number of the next block in the file. This chain continuous until the last block, which has a special end of file values as the table entry. Unused blocks are indicated by a ‘0’ table value. Allocation a new block to a file is simple.



3) Indexed allocation:-

1) Linked allocation solves the external fragmentation and size declaration problems of contagious allocation. How ever in the absence of a FAT, linked allocation can not support efficient direct access.
2) The pointers to the blocks are scattered with the blocks themselves all over the disk and need to be retrieved in order.
3) Indexed allocation solves this problem by bringing all the pointers together in to one location i.e *the index block.*
4) 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 ith block of the file.
5) The directory contains the address of the index block.

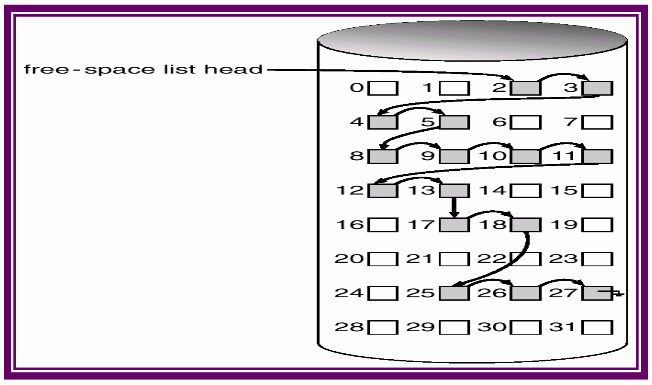


To read the ith block we use the pointer in the ith index block entry to find and read the desired block.

6) When the file is created, all pointers in the index block are set to nil. When the ith block is first written, a block is obtained from the free space manager, and its address is put in the ith index block entry.
7) It supports the direct access with out suffering from external fragmentation, but it suffers from the wasted space. The pointer overhead of the index block is generally greater than the pointer over head of linked allocation.

Free space management:-

1) To keep track of free disk space, the system maintains a free space list. The free space list records all disk blocks that are free.
2) 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.
3) When the file is deleted, its disk space is added to the free space list.
There are many methods to find the free space.
1) **Bit vector:-**
The 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.
Ex:- consider a disk where blocks 2,3,4,5,8,9,10,11,12,13,17,18,25, are free and rest of blocks are allocated the free space bit map would be
0011110011111100011000000010000……..
The main advantage of this approach is that it is relatively simple and efficient to find the first free block or ‘n’ consecutive free blocks on the disk
2) **Linked list:-**
Another approach is to link together all the free disk blocks, keeping a pointer to the first free block in a special location on the disk and caching it in memory. This first block contain a pointer to the next free disk block, and so on. How ever this scheme is not efficient to traverse the list, we must read each block, which requires I/O time. Disk space is also wasted to maintain the pointer to next free space.



3) **Grouping:-**
Another method is 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 another ‘n’ free blocks and so on.
Advantages:- The main advantage of this approach is that the addresses of a large no.of blocks can be found quickly.
4) **Counting:-**
Another approach is counting. Generally several contiguous blocks may be allocated or freed simultaneously. Particularly when space is allocated with the contiguous allocation algorithm rather than keeping a list of ‘n’ free disk address. We can keep the address of 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.

Directory Implementation:-

Linear list:-

1) The simple method of implement ting a directory is to use a linear list of file names with pointers to the data blocks.
2) A linear list of directory entries requires a linear search to find a particular entry.
3) This method is simple to program but is time consuming to execute.
4) 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.
5) To delete a file we search the directory for the named file, then release the space allocated to it.
6) To reuse directory entry, we can do one of several things.
7) We can mark the entry as unused or we can attach it to a list of free directory entries.

Disadvantage:- the disadvantage of a linear list of directory entries is the linear search to find a file.

Hash table:-

Another data structure used for a file directory is a hash table. With this method, a liner list stores the directory entries, but a hash data structure is also used. The hash table takes a value computed from the file name and returns a pointer to the file name in linear list. It can greatly decrease the search time.

****

Mass Storage

Disk Structure:

1. Disk drives are addressed as large 1-dimensional arrays of *logical blocks*, where the logical block is the smallest unit of transfer.
2. The 1-dimensional array of logical blocks is mapped into the sectors of the disk sequentially.
   a) Sector 0 is the first sector of the first track on the outermost cylinder.
   b) Mapping proceeds in order through that track, then the rest of the tracks in that cylinder, and then through the rest of the cylinders from outermost to innermost.

Disk Scheduling:

1. The operating system is responsible for using hardware efficiently — for the disk drives, this means having a fast access time and disk bandwidth.
2. Access time has two major components
   a) ***Seek time*** is the time for the disk are to move the heads to the cylinder containing the desired sector.
   b) ***Rotational latency*** is the additional time waiting for the disk to rotate the desired sector to the disk head.
3. Minimize seek time
4. Seek time ≈ seek distance
5. 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.
6. Several algorithms exist to schedule the servicing of disk I/O requests.
7. We illustrate them with a request queue (0-199).

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

   Head pointer 53

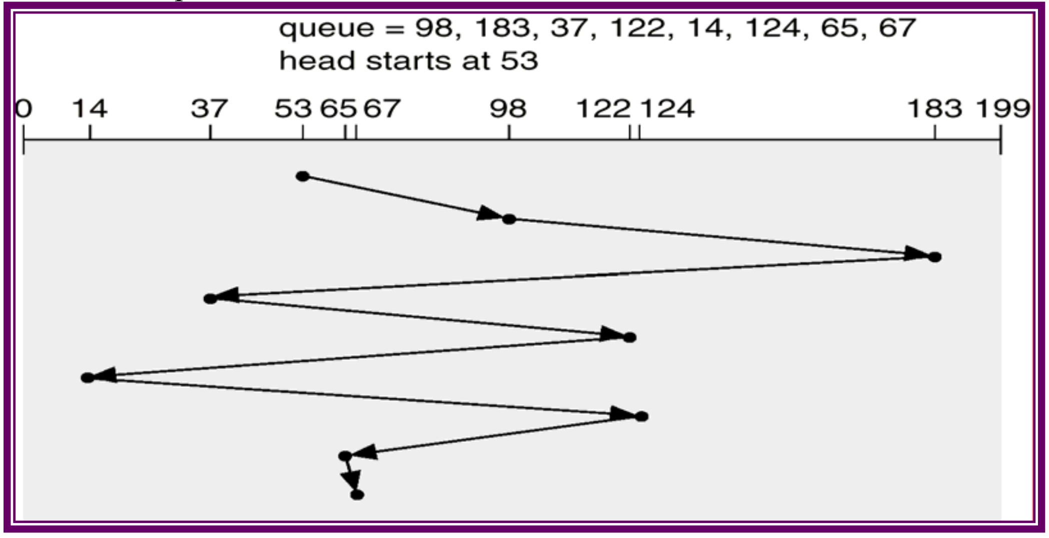
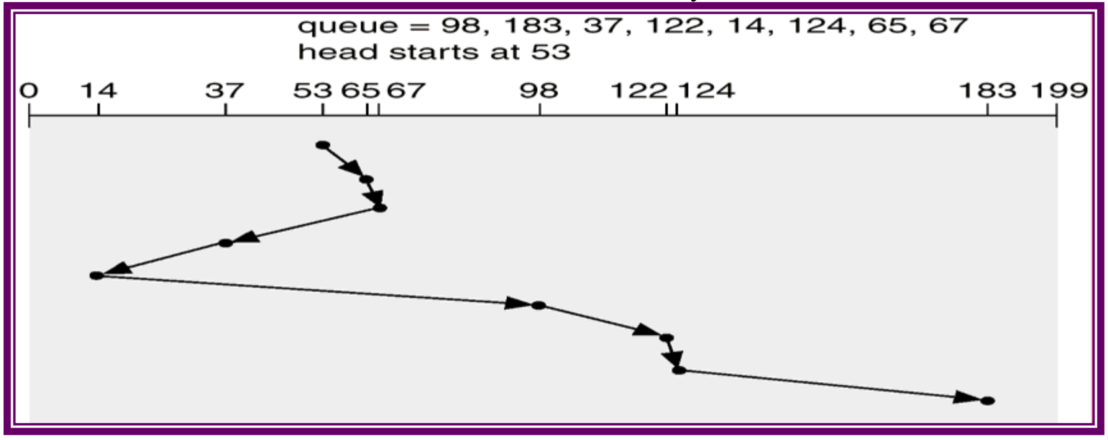


Illustration shows total head movement of 640 cylinders.

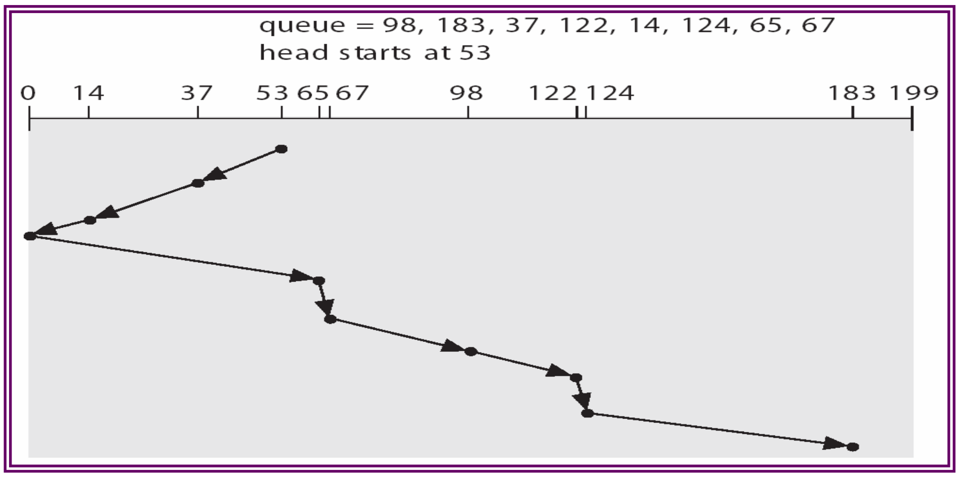
SSTF

1. Selects the request with the minimum seek time from the current head position.
2. SSTF scheduling is a form of SJF scheduling; may cause starvation of some requests.
3. Illustration shows total head movement of 236 cylinders.



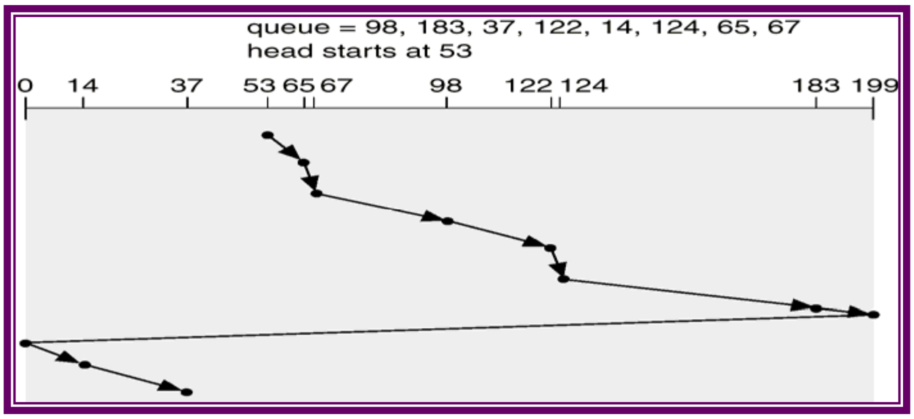
SCAN

1. The disk arm starts at one end of the disk, and moves toward the other end, servicing requests until it gets to the other end of the disk, where the head movement is reversed and servicing continues.
2. Sometimes called the ***elevator algorithm.***
3. Illustration shows total head movement of 208 cylinders.



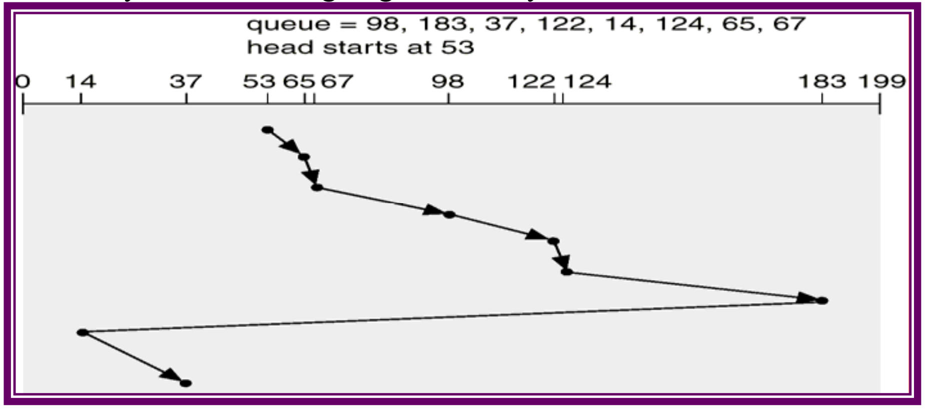
CSCAN

1. Provides a more uniform wait time than SCAN.
2. The head moves from one end of the disk to the other. servicing requests as it goes. When it reaches the other end, however, it immediately returns to the beginning of the disk, without servicing any requests on the return trip.
3. Treats the cylinders as a circular list that wraps around from the last cylinder to the first one.



C-LOOK

1. Version of C-SCAN
2. Arm only goes as far as the last request in each direction, then reverses direction immediately, without first going all the way to the end of the disk.



Swap Space Management

1. Swap-space — Virtual memory uses disk space as an extension of main memory.
2. Swap-space can be carved out of the normal file system,or, more commonly, it can be in a separate disk partition.
3. Swap-space management

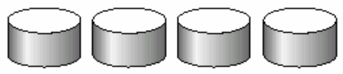
   a) 4.3BSD allocates swap space when process starts; holds *text segment* (the program) and *data segment.*

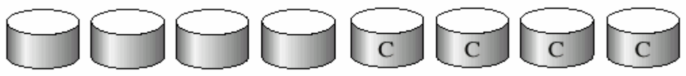
   b) Kernel uses *swap maps* to track swap-space use.

   c) Solaris 2 allocates swap space only when a page is forced out of physical memory, not when the virtual memory page is first created.

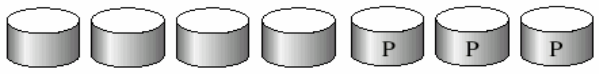
RAID

1. RAID – multiple disk drives provides reliability via redundancy.
2. RAID is arranged into six different levels.
3. Several improvements in disk-use techniques involve the use of multiple disks working cooperatively.
4. Disk striping uses a group of disks as one storage unit.
5. RAID schemes improve performance and improve the reliability of the storage system by storing redundant data.
6. *Mirroring* or *shadowing* keeps duplicate of each disk.
7. *Block interleaved parity* uses much less redundancy.

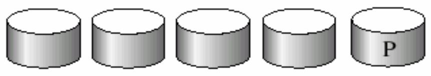

(a) RAID 0: non-redundant striping



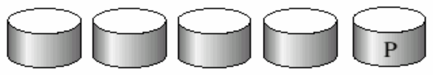
(b) RAID 1: mirrored disks



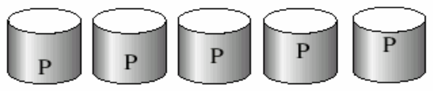
(c) RAID 2: memory-style error-correcting codes



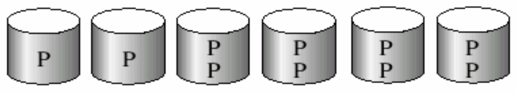
(d) RAID 3: bit-interleaved Parity



(e) RAID 4: block-interleaved parity

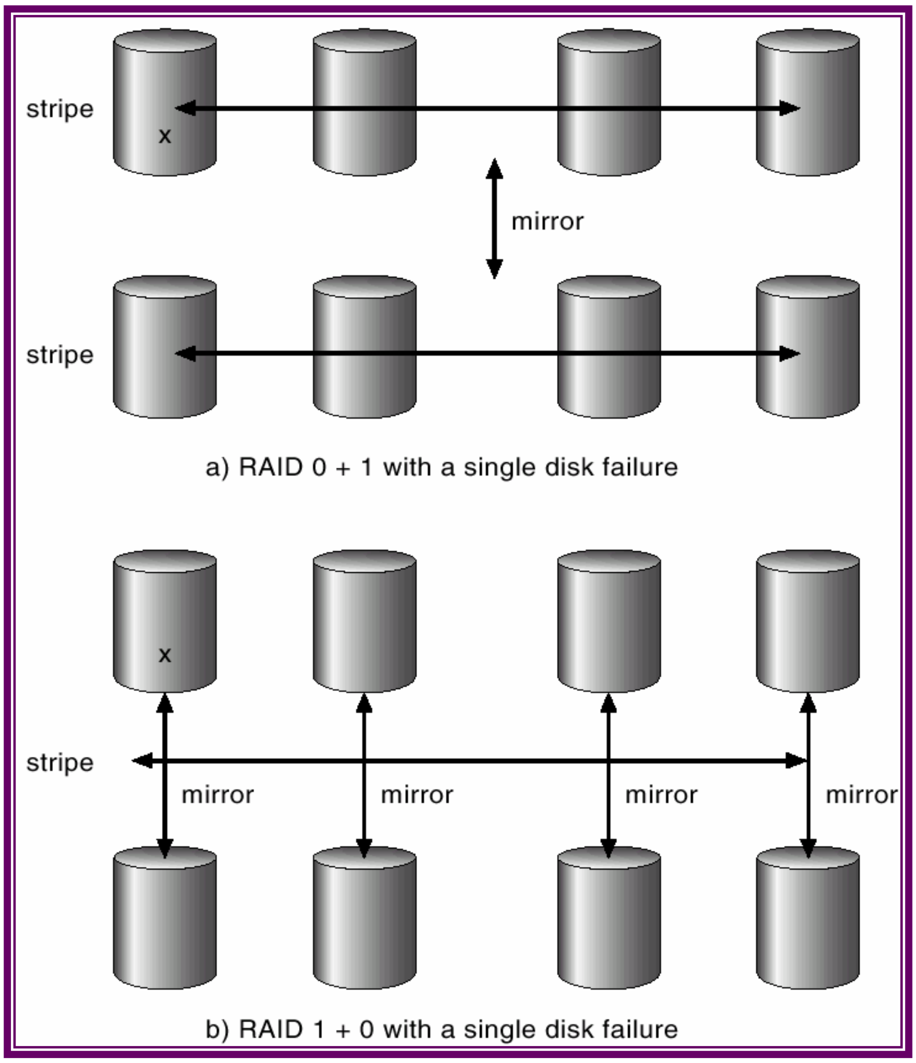


(f) RAID 5: block-Interleaved distributed parity



(g) RAID 6: P + Q redundancy

NOTE: Check your class note book for theory.


stripe
x
mirror
stripe
a) RAID 0 + 1 with a single disk failure
x
stripe
mirror
mirror
mirror
mirror
b) RAID 1 + 0 with a single disk failure

I/O SYSTEM

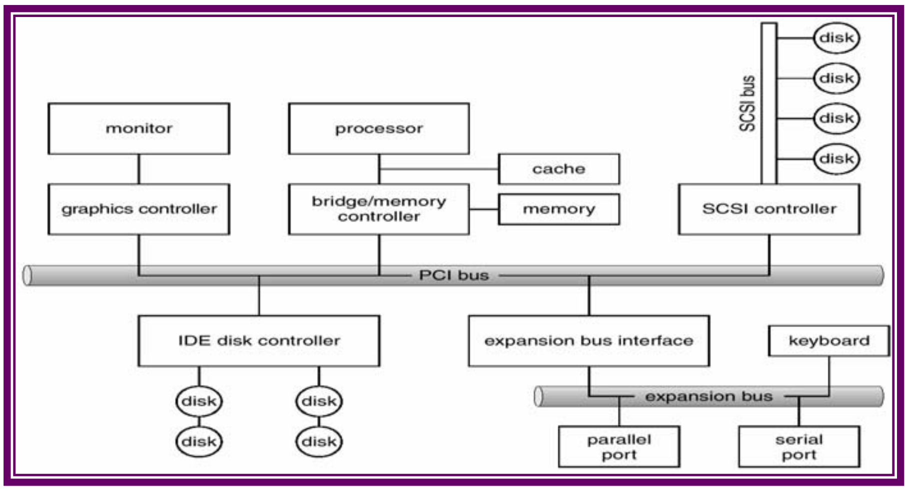
The two main jobs of a computer are I/O and processing. The role of the operating system in computer I/O is to manage and control I/O operations and I/O devices.

1 Overview:

The control of devices connected to the computer is a major concern of operating-system designers. Because I/O devices vary so widely in their function and speed, variety of methods are needed to control them. These methods form the I/O sub-system of the kernel, which separates the rest of the kernel from the complexity of managing I/O devices. I/O-device technology exhibits two conflicting trends. On one hand, we see increasing standardization of software and hardware interfaces. On the other hand, we see an increasingly broad variety of I/O devices. Some new devices are so unlike previous devices that it is a challenge to incorporate them into our computers and operating systems. To encapsulate the details and oddities of different devices, the kernel of an operating system is structured to use device driver modules. The device drivers present a uniform device access interface to the I/O subsystem, much as system calls provide a standard interface between the application and the operating system.

2 I/O Hardware:

Computers operate a great many kinds of devices. General types include storage devices (disks, tapes), transmission devices (network cards, modems), and human-interface devices (screen, keyboard, mouse). Other devices are more specialized. A device communicates with a computer system by sending signals over a cable or even through the air. The device communicates with the machine via a connection point termed a port. If one or more devices use a common set of wires, the connection is called a bus. When device A has a cable that plugs into device B, and device B has a cable that plugs into device C, and device C plugs into a port on the computer, this arrangement is called a daisy chain. It usually operates as a bus. PCI bus (the common PC system bus) that connects the processor-memory subsystem to the fast devices, and an expansion bus that connects relatively slow devices such as the keyboard and serial and parallel ports. Controller is a collection of electronics that can operate a port, a bus, or a device. A serial-port controller is an example of a simple device controller. Because the SCSI protocol is complex, the SCSI bus controller is often implemented as a separate circuit board (a host adapter) that plugs into the computer. The controller has one or more registers for data and control signals. The processor communicates with the controller by reading and writing bit patterns in these registers. One way that, this communication can occur is through the use of special I/O instructions that specify the transfer of a byte or word to an I/O port address..



An I/O port typically consists of four registers, called **the status, control, data-in, and data-out registers.** The status register contains bits that can be read by the host. These bits indicate states such as whether the current command has completed, whether a byte is available to be read from the data-in register, and whether there has been a device error. The control register can be written by the host to start a command or to change the mode of a device. For instance, a certain bit in the control register of a serial port chooses between full-duplex and half-duplex communication, another enables parity checking, a. third bit sets the word length to 7 or 8 bits, and other bits select one of the speeds supported by the serial port. The data-in register is read by the host to get input, and the data-out register is written by the host to send output.

2.1 Polling:

The complete protocol for interaction between the host and a controller can be intricate, but the basic handshaking notion is simple. We assume that 2 bits are used to coordinate the producer consumer relationship between the controller and the host. The controller indicates its state through the busy bit in the status register.

1. The host repeatedly reads the busy bit until that bit becomes clear.
2. The host sets the write bit in the command register and writes a byte into the data-out register.
3. The host sets the command-ready bit.
4. When the controller notices that the command-ready bit is set, it sets the busy bit.
5. The controller reads the command register and sees the write command. It reads the data-out register to get the byte, and does the I/O to the device.6. The controller dears the command-ready bit, clears the error bit in the status register to indicate that the device I/O succeeded, and clears the busy bit to indicate that it is finished.

This loop is repeated for each byte.

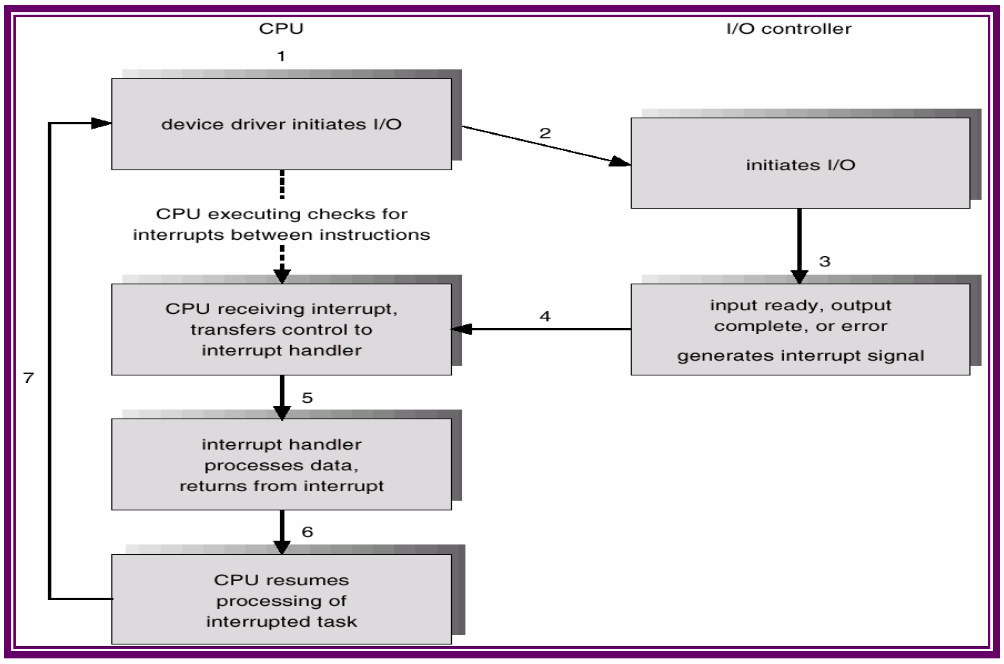
In step 1, the host is busy-waiting or polling: It is in a loop, reading the status register over and over until the busy bit becomes clear.3 CPU-instruction cycles are sufficient to poll a device: read a device register, logical-and to extract a status bit, and branch if not zero. The hardware mechanism that enables a device to notify the CPU is called an interrupt.

2.2 Interrupts:

The CPU hardware has a wire called the interrupt request line that the CPU senses after executing every instruction. When the CPU detects that a controller has asserted a signal on the interrupt request line, the CPU saves a small amount of state, such as the current value of the instruction pointer, and jumps to the interrupt-handler routine at a fixed address in memory. The interrupt handler determines the cause of interrupt, perform the necessary processing and execute a return from interrupt instruction to return the CPU to the execution state prior to the interrupt. We say that the device controller raises an interrupt by asserting a signal on the interrupt request line, the CPU catches the interrupt and dispatches to the interrupt handler, and the handler clears the interrupt by servicing the device.

We need more sophisticated interrupt-handling features:

1) We need the ability to defer interrupt handling during critical processing.
2) We need an efficient way to dispatch to the proper interrupt handler for a device, without first polling all the devices to see which one raised the interrupt.
3) We need multi level interrupts, so that operating system can distinguish between high and low-priority interrupts, and can respond with the appropriate degree of urgency.

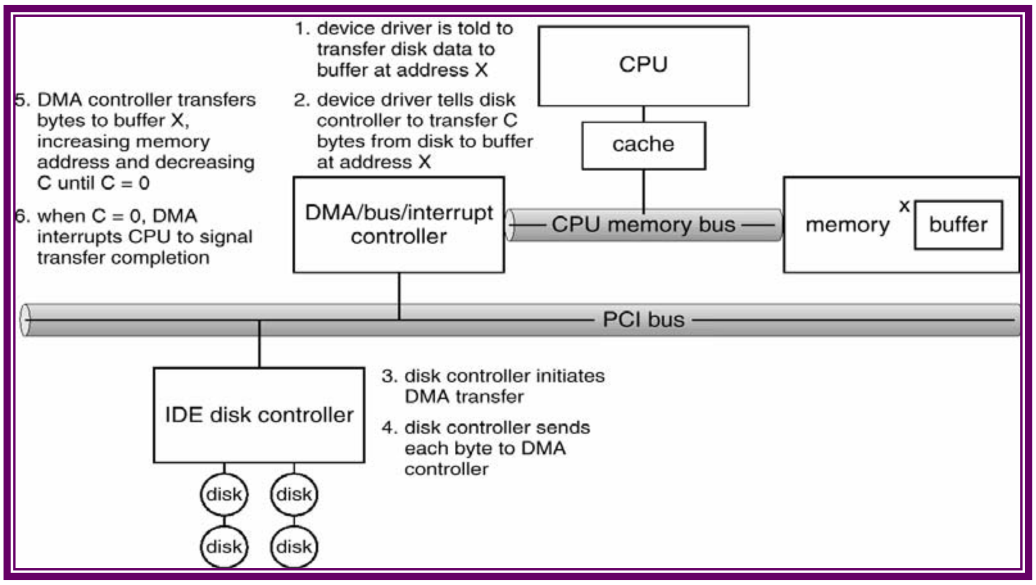

CPU
I/O controller
1
device driver initiates I/O
2
initiates I/O
CPU executing checks for interrupts between instructions
3
CPU receiving interrupt, transfers control to interrupt handler
4
input ready, output complete, or error generates interrupt signal
7
5
interrupt handler processes data, returns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

In modern computer hardware, these three features are provided by the CPU and by the interrupt-controller hardware. Most CPUs have two interrupt request lines.

1) Non-maskable interrupt: which is reserved for events such as unrecoverable memory errors.
2) Maskable interrupt: It can be turned off by the CPU before the execution of critical instruction sequences that must not be interrupted.

DMA

For a device that does large transfers, such as a disk drive, it seems wasteful to use an expensive general-purpose processor to watch status bits and to feed data into a controller register 1 byte at a time—a process termed programmed I/O (PIO).some of this work to a special-purpose processor called a direct-memory-access (DMA) controller. To initiate a DMA transfer, the host writes a DMA command block into memory. This block contains a pointer to the source of a transfer, a pointer to the destination of the transfer, and a count of the number of bytes to transferred. The CPU writes the address of this command block to DMA controller, then goes on with other work. The DMA controller then proceeds to operate the memory bus directly, placing addresses on the bus to perform transfers without the help of the main CPU. A simple DMA controller is a standard component in PCs, and bus-mastering I/O boards for the PC usually contain their own high-speed DMA hardware.


1. device driver is told to transfer disk data to buffer at address X
2. device driver tells disk controller to transfer C bytes from disk to buffer at address X
5. DMA controller transfers bytes to buffer X, increasing memory address and decreasing C until C = 0
6. when C = 0, DMA interrupts CPU to signal transfer completion
CPU
cache
DMA/bus/interrupt controller
CPU memory bus
memory
X
buffer
PCI bus
IDE disk controller
3. disk controller initiates DMA transfer
4. disk controller sends each byte to DMA controller
disk
disk
disk
disk

Handshaking between the DMA controller and the device controller is performed via a pair of wires called DMA-request and DMA-acknowledge. The device controller places a signal on the DMA-request wire when a word of data is available for transfer. This signal causes the DMA controller to seize the memory bus, to place the desired address on the memory-address wires, and to place a signal on the DMA-acknowledge wire. When the device controller receives the DMA-acknowledge signal, it transfers the word of data to memory, and removes the DMA-request signal. When the entire transfer is finished, the DMA controller interrupts the CPU. Although this cycle stealing can slow down the CPU computation, offloading the data-transfer work to a DMA controller generally improves the total system performance.

Some computer architectures use physical memory addresses for DMA, but others perform direct virtual memory access (DVMA), using virtual addresses that undergo virtual to physical memory address translation.

3 Application I/O Interface:

Specifically, we can abstract away the detailed differences in I/O devices by identifying a few general kinds. Each of these general kinds is accessed through a standardized set of functions—an interface. The actual differences are encapsulated in kernel modules called device drivers that internally are custom tailored to each device, but that export one of the standard interfaces. The purpose of the device-driver layer is to hide the differences among device controllers from the I/O subsystem of the kernel, much as the I/O system calls encapsulate the behavior of devices in a few generic classes that hide hardware differences from applications. Making the I/O subsystem independent of the hardware simplifies the job of the operating-system developer. It also benefits the hardware manufacturers. New peripherals can be attached to a computer without waiting for the operating-system vendor to develop support code.

Devices vary in many dimensions:

- **Character-stream or block:** A character-stream device transfers bytes one by one, whereas a block device transfers a block of bytes as a unit.
- **Sequential or random-access:** A sequential device transfers data in a fixed order that is determined by the device, whereas the user of a random-access device can instruct the device to seek to any of the available data storage locations.
- **Synchronous or asynchronous:** a synchronous device is one that performs data transfers with predictable response time. An asynchronous device exhibits irregular or unpredictable response time.
- **Sharable or dedicated:** a sharable device can be used concurrently by several processes or threads. A dedicated device cannot.
- **Speed of operation:** device speeds range from a few bytes per second to a few gigabytes per second.
- **Read-Write, Read-only, Write-only:** some devices perform both input and output, but others support only one data direction.

3.1 Block and Character Devices:

The block-device interface captures all the aspects necessary for accessing disk drives and other block oriented devices. The expectation is that the device understands commands such as read and -write, and, if it is a random-access device, it has a seek command to specify which block to transfer next. The operating system itself, and special applications such as database-management systems, may prefer to access a block device as a simple linear array of blocks. This mode of access is sometimes called raw I/O. Memory-mapped file access can be layered on top of block-device drivers. Rather than offering read and write operations, a memory-mapped interface provides access to disk storage via an array of bytes in main memory.

To execute a program, the operating system maps the executable into memory, and then transfers control to the entry address of the executable. The mapping interface is also commonly used for kernel access to swap space on disk. A keyboard is an example of a device that is accessed through a character stream interface. The basic system calls in this interface enable an application to get or put one character. On top of this interface, libraries can be built that offer line-at-a-time access, with buffering and editing services

3.2 Network Devices:

Because the performance and addressing characteristics of network I/O differ significantly from those of disk I/O, most operating systems provide a network I/O interface that is different from the read-write-seek interface used for disks. One interface that is available in many operating systems, including UNIX and Windows NT, is the network socket interface. The system calls in the socket interface enable an application to create a socket, to connect a local socket to a remote address, to listen for any remote application to plug into the local socket, and to send and receive packets over the connection. The use of select eliminates the polling and busy waiting that would otherwise be necessary for network I/O. These functions encapsulate the essential behaviors of networks, greatly facilitating the creation of distributed applications that can use any underlying network hardware and protocol stack. Many other approaches to inter process communication and network communication have been implemented. For instance, Windows NT provides one interface to the network interface card, and a second interface to the network protocols.

3.3 Clocks and Timers:

Most computers have hardware clocks and timers that provide three basic functions:

- Give the current time
- Give the elapsed time
- Set a timer to trigger operation X at time T

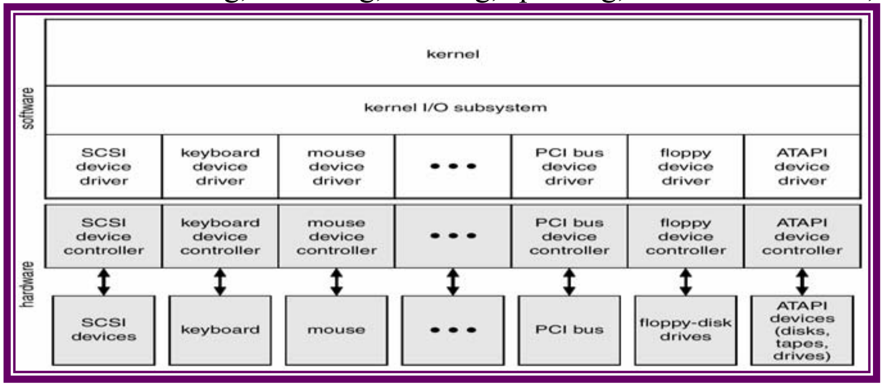
The hardware to measure elapsed time and to trigger operations is called a programmable interval timer. It can be set to wait a certain amount of time and then to generate an interrupt. It can be set to do this operation once, or to repeat the process, to generate periodic interrupts. The operating system may also provide an interface for user processes to use timers. The operating system can support more timer requests than the number of timer hardware channels by simulating virtual clocks.

3.4 Blocking and Non-blocking I/O:

One remaining aspect of the system-call interface relates to the choice between blocking I/O and non-blocking (asynchronous) I/O. When an application issues blocking system call the application is moved from operating systems run queue to wait queue. After the system call completes, the application is moved back to the run queue, where it is eligible to resume execution, at which time it will receive the values returned by the system call. The physical actions performed by I/O devices are generally asynchronous. Some user-level processes need non-blocking I/O. One example is a user interface that receives keyboard and mouse input while processing and displaying data on the screen. Another example is a video application that reads frames from a file on disk while simultaneously decompressing and displaying the output on the display. A non-blocking call does not halt the execution of the application for an extended time. Instead, it returns quickly, with a return value that indicates how many bytes were transferred. An alternative to a non-blocking system call is an asynchronous system call. An asynchronous call returns immediately, without waiting for the I/O to complete. The application continues to execute its code, and the completion of the I/O at some future time is communicated to the application, either through the setting of some variable in the address space of the application, or through the triggering of a signal or software interrupt or a call-back routine that is executed outside the linear control flow of the application.

4 Kernel I/O Sub-systems:

Kernels provide many services related to I/O. The services that can be provided are I/O scheduling, buffering, caching, spooling, device reservation, and error handling.



4.1 I/O Scheduling:

To schedule a set of I/O requests means to determine a good order in which to execute them. The order in which applications issue system calls rarely is the best choice. Scheduling can improve overall system performance, can share device access fairly among processes, and can reduce the average waiting time for I/O to complete. Suppose that a disk arm is near the beginning of a disk, and that three applications issue blocking read calls to that disk. Application 1 requests a block near the end of the disk, application 2 requests one near the beginning, and application 3's request is in the middle of the disk. It is clear that the operating system can reduce the distance that the disk arm travels by serving the applications in order 2,3,1. The I/O scheduler rearranges the order of the queue to improve the overall system efficiency and the average response time experienced by applications. One way that the I/O subsystem improves the efficiency of the computer is by scheduling I/O operations. Another way is by using storage space in main memory or on disk, via techniques called buffering, caching, and spooling.

4.2. Buffering:

A *buffer* is a memory area that stores data while they are transferred between two devices or between a device and an application. Buffering is done for three reasons.

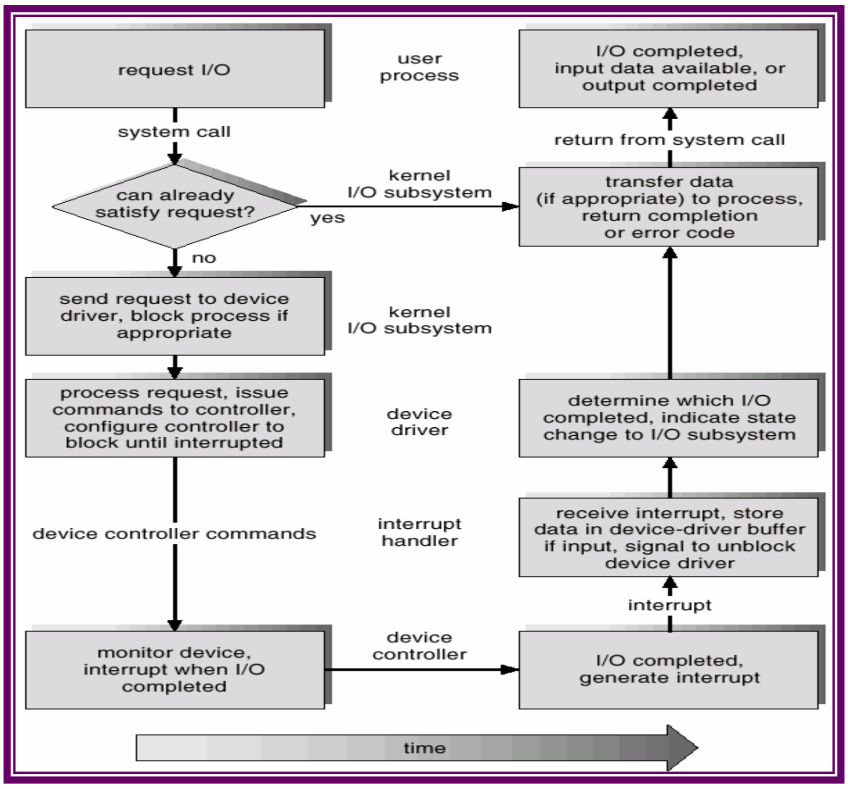
1) To cope with a speed mismatch between the producer and consumer of a data stream. The modem is about a thousand times slower than the hard disk. So a buffer is created in main memory. After the modem fills the first buffer, the disk write is requested. The modem then starts to fill the second buffer while the first buffer is written to disk. By the time that the modem has filled the second buffer, the disk write from the first one should have completed, so the modem can switch back to the first buffer while the disk writes the second one. This double buffering decouples the producer of data from, the consumer, thus relaxing timing requirements between them.
2) To adapt between devices that have different data-transfer sizes. At the sending side, a large message is fragmented into small network packets. The packets are sent over the network, the receiving side places them in a reassembly buffer to form an image of source data.
3) To support copy semantics for application I/O. An example will clarify the meaning of "copy semantics." Suppose that an application has a buffer of data that it wishes to write to disk. It calls the write system call, providing a pointer to the buffer, and an integer specifying the number of bytes to write. With copy semantics, the version of the data written to disk is guaranteed to be the version at the time of the application system call, independent of any subsequent changes in the application's buffer. A simple way that the operating system can guarantee copy semantics is for the write system call to copy the application data into a kernel buffer before returning control to the application.

4.3 Caching:

A *cache* is region of fast memory that holds copies of data. Access to the cached copy is more efficient than access to the original. The difference between a buffer and a cache is that a buffer may hold the only existing copy of a data item, whereas a cache, by definition, just holds a copy on faster storage of an item that resides elsewhere. To preserve copy semantics and to enable efficient scheduling of disk I/O, the operating system uses buffers in main memory to hold disk data. These buffers are also used as a cache, to improve the I/O efficiency for files that are shared by applications, or that are being written and reread rapidly.

Modern operating systems obtain significant flexibility from the multiple stages of lookup tables in the path between a request and a physical device controller. The mechanisms that pass requests between applications and drivers are general:

1. A process issues a blocking read system call to a file descriptor of a file that has been opened previously.
2. The system-call code in the kernel checks the parameters for correctness In the case of input, if the data are already available in the buffer cache, the data are returned to the process and the I/O request is completed.
3. Otherwise, a physical I/O needs to be performed, so the process is removed from the run queue and is placed on the wait queue for the device, and the I/O request is scheduled. Eventually, the I/O subsystem sends the request to the device driver. Depending on the operating system, the request is sent via a subroutine call or via an in-kernel message.
4. The device driver allocates kernel buffer space to receive the data, and schedules the I/O. Eventually, the driver sends commands to the device controller by writing into the device control registers.
5. The device controller operates the device hardware to perform the Data transfer.
6. The driver may poll for status and data, or it may have set up a DMA transfer into kernel memory. We assume that the transfer is managed by a DMA controller, which generates an interrupt when the transfer completes.
7. The correct interrupt handler receives the interrupt via the interrupt-vector table, stores any necessary data, signals the device driver, and returns from the interrupt.
8. The device driver receives the signal, determines which I/O request completed, determines the request's status, and signals the kernel I/O subsystem that the request has been completed.
9. The kernel transfers data or return codes to the address space of the requesting process, and moves the process from the wait queue back to
the ready queue.
10. moving the process to ready queue unblock the process. When the scheduler assigns the process to the CPU, the process resumes execution at the completion of the system call.


request I/O
user process
I/O completed, input data available, or output completed
system call
return from system call
can already satisfy request?
kernel I/O subsystem
yes
transfer data (if appropriate) to process, return completion or error code
no
send request to device driver, block process if appropriate
kernel I/O subsystem
process request, issue commands to controller, configure controller to block until interrupted
device driver
determine which I/O completed, indicate state change to I/O subsystem
device controller commands
interrupt handler
receive interrupt, store data in device-driver buffer if input, signal to unblock device driver
interrupt
monitor device, interrupt when I/O completed
device controller
I/O completed, generate interrupt
time

I/O Life Cycle

***** All The Best *****