MODULE-1

1. (a). Distinguish between the following terms.

i. Multiprogramming and Multitasking

ii. Multiprocessor System and Clustered System.

Multitasking System and Multiprogramming: Time sharing (or multitasking) is a logical extension of multiprogramming. In time-sharing systems, the CPU executes multiple jobs by switching among them, but the switches occur so frequently that the users can interact with each program while it is running. A time-shared operating system uses CPU scheduling and multiprogramming to provide each user with a small portion of a time-shared computer. Each user has at least one separate program in memory. A program loaded into memory and executing is called a process.

Time-sharing and multiprogramming require several jobs to be kept simultaneously in memory. Since in general main memory is too small to accommodate all jobs, the jobs are kept initially on the disk in the job pool.

Multiprocessor systems and clustered systems: Multiprocessor systems (also known as parallel systems or tightly coupled systems) are growing in importance. Such systems have two or more processors in close communication, sharing the computer bus and sometimes the clock, memory, and peripheral devices.

Clustered systems differ from multiprocessor systems, however, in that they are composed of two or more individual systems coupled together. Clustering is usually used to provide high-availability service; that is, service will continue even if one or more systems in the cluster fail. High availability is generally obtained by adding a level of redundancy in the system.

(b). Define operating Systems. Explain the dual-mode operating system with a neat diagram.

A program that acts as an intermediary between a user of a computer and the computer hardware. An operating System is a collection of system programs that together control the operations of a computer system. Some examples of operating systems are UNIX, Mach, MS-DOS, MS-Windows, Windows/NT, Chicago, OS/2, MacOS, VMS, MVS, and VM. The **Dual-Mode** taken by most computer systems is to provide hardware support that allows us to differentiate among various modes of execution. At the very least, we need two separate modes of operation: **user mode and kernel mode** (also called supervisor

mode, system mode, or privileged mode). A bit, called the mode bit is added to the hardware of the computer to indicate the current mode: kernel (0) or user (1). With the mode bit, we are able to distinguish between a task that is executed on behalf of the operating system and one that is executed on behalf of the user. When the computer system is executing on behalf of a user application, the system is in user mode. However, when a user application requests a service from the operating system (via a system call), it must transition from user to kernel mode to fulfil the request. This is shown in Figure below. As we shall see, this architectural enhancement is useful for many other aspects of system operation as well.

At system boot time, the hardware starts in kernel mode.

The operating system is then loaded and starts user applications in user mode. Whenever a trap or interrupt occurs, the hardware switches from user mode to kernel mode (that is, changes the state of the mode bit to 0). Thus, whenever the operating system gains control of the computer, it is in kernel mode. The system always switches to user mode (by setting the mode bit to 1) before passing control to a user program. The dual mode of operation provides us with the means for protecting the operating system from errant users-and errant users from one another.

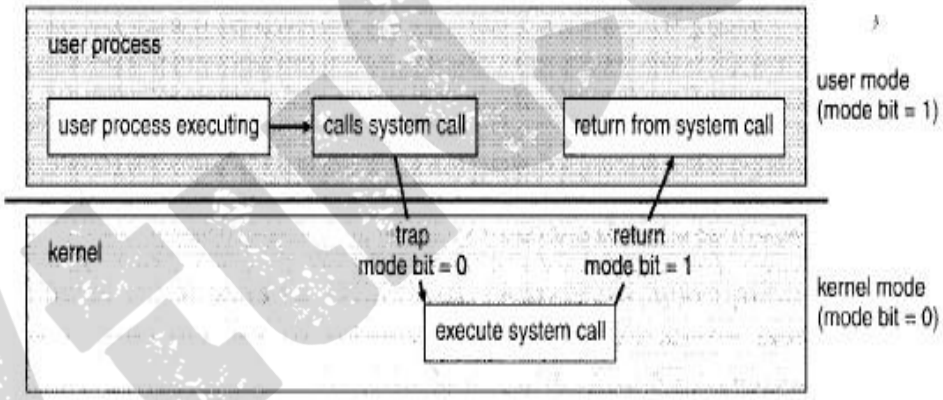


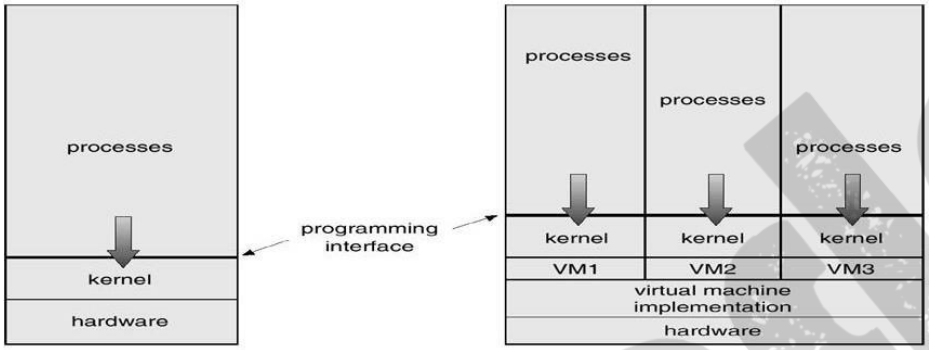
Figure: Transition from user to kernel mode.

(c). With a neat diagram, explain the concept of the virtual machine.

A virtual machine takes the layered approach to its logical conclusion. It treats hardware and the operating system kernel as though they were all hardware. A virtual machine provides an interface identical to the underlying bare hardware. The operating system creates the illusion of multiple processes, each executing on its own processor with its own (virtual) memory. The resources of the physical computer are shared to create the virtual machines.

- ❖ CPU scheduling can create the appearance that users have their own processor.
- ❖ Spooling and a file system can provide virtual card readers and virtual line printers.
- ❖ A normal user time-sharing terminal serves as the virtual machine operator's console.

System Models



Non-virtual Machine Virtual Machine

Advantages/Disadvantages of Virtual Machines:

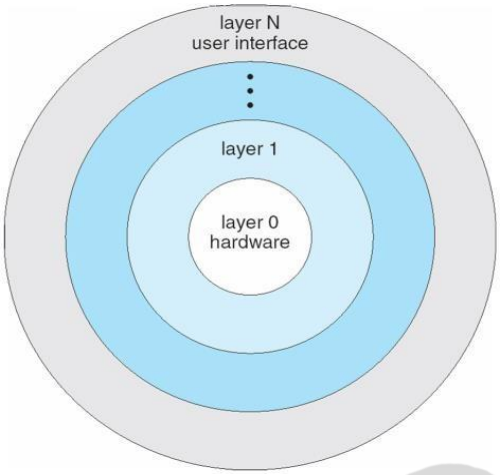
- ❖ The virtual-machine concept provides complete protection of system resources since each virtual Machine is isolated from all other virtual machines. This isolation, however, permits no direct sharing of resources.
- ❖ A virtual-machine system is a perfect vehicle for operating-systems research and development. System development is done on the virtual machine, instead of on a physical machine and so does not disrupt normal system operation.
- ❖ The virtual machine concept is difficult to implement due to the effort required to provide an *exact* duplicate to the underlying machine.

OR

2. (a). Explain the layered approach of operating system structure with a supporting diagram.

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

The main advantage of the layered approach is simplicity of construction and debugging. The layers are selected so that each uses functions (operations) and services of only lower-level layers. This approach simplifies debugging and system verification.



(b). What are system calls? Briefly point out its types with illustrations.

- System calls provide an interface between the process and the operating system.
- System calls allow user-level processes to request some services from the operating system which process itself is not allowed to do.
- For example, for I/O a process involves a system call telling the operating system to read or write particular area and this request is satisfied by the operating system.
- System calls can be grouped roughly into five major categories: process control, file manipulation, device manipulation, information maintenance, and communications.

Process control: A running program needs to be able to halt its execution either normally (end) or abnormally (abort). If a system call is made to terminate the currently running program abnormally, or if the program runs into a problem and causes an error trap, a dump of memory is sometimes taken and an error message generated.

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

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

Information Maintenance: Many system calls exist simply for the purpose of transferring information between the user program and the operating system. For example, most systems have a system call to return the current time and date. Other system calls may return information about the system, such as the number of current users, the version number of the operating system, the amount of free memory or disk space, and so on.

Communication: There are two common models of inter-process communication: the message passing model and the shared-memory model. In the message-passing model, the communicating processes exchange messages with one another to transfer information. Messages can be exchanged between the processes either directly or indirectly through a common mailbox.

(c). Explain the services of the operating system that are helpful for the user and the system.

Operating systems can be explored from two viewpoints, the user and the system:

- **User View:** The user's view of the computer varies according to the interface being used. Most computer users sit in front of a PC, consisting of a monitor, keyboard, mouse, and system unit. Such a system is designed for one user to monopolize its resources. The goal is to maximize the work (or play) that the user is performing. In this case, the operating system is designed mostly for ease of use, with some attention paid to performance and none paid to resource utilization-how various hardware and software resources are shared. Performance is, of course, important to the user; but rather than resource utilization, such systems are optimized for the single-user experience.
- **System View:** From the computer's point of view, the operation system is the program most intimately involved with the hardware. In this context, we can view an operating system as a resource allocator. A computer system has many resources that may be required to solve a problem: CPU time, memory space, file-storage space, I/O devices, and so on. The operating system acts as the manager of these resources.

A control program manages the execution of user programs to prevent errors and improper use of the computer. It is especially concerned with the operation and control of I/O devices. Following are the six services provided by operating systems to the convenience of the users.

- **User interface:** Almost all operating systems have a user interface (UI). This interface can take several forms. One is a command-line interface (CLI) and other is a graphical user interface (GUI) is used.

- **Program Execution:** The purpose of computer systems is to allow the user to execute programs. So, the operating system provides an environment where the user can conveniently run programs.
- **I/O Operations:** Each program requires an input and produces output. This involves the use of I/O. So, the operating systems are providing I/O makes it convenient for the users to run programs.
- **File System Manipulation:** The output of a program may need to be written into new files or input taken from some files. The operating system provides this service. Finally, some programs include permissions management to allow or deny access to files or directories based on file ownership.
- **Communications:** The processes need to communicate with each other to exchange information during execution. It may be between processes running on the same computer or running on the different computers. Communications can be occurred in two ways: (i) shared memory or (ii) message passing
- **Error Detection:** An error is one part of the system may cause malfunctioning of the complete system. To avoid such a situation operating system constantly monitors the system for detecting the errors. This relieves the user of the worry of errors propagating to various part of the system and causing malfunctioning.

Following are the three services provided by operating systems for ensuring the efficient operation of the system itself.

- Resource allocation
- Accounting
- Protection

MODULE-2

3. (a). With a neat diagram, explain the states of a process with a transition diagram and process control block.

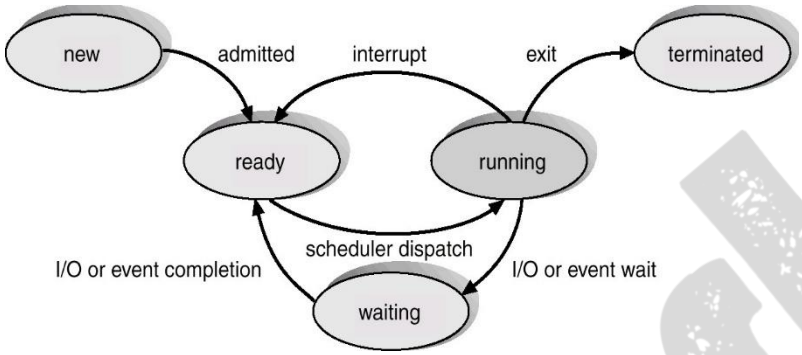


Figure: Diagram of process states

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

- **New State:** The process is being created.
- **Running State:** A process is said to be running if it has the CPU, that is, process actually using the CPU at that particular instant.
- **Blocked (or waiting) State:** A process is said to be blocked if it is waiting for some event to happen such that as an I/O completion before it can proceed. Note that a process is unable to run until some external event happens.
- **Ready State:** A process is said to be ready if it needs a CPU to execute. A ready state process is runnable but temporarily stopped running to let another process run.
- **Terminated state:** The process has finished execution.

Process Control Block: Each process is represented in the operating system by a process control block also called a task control block. A process control block is shown in Figure below. It contains many pieces of information associated with a specific process, including these:

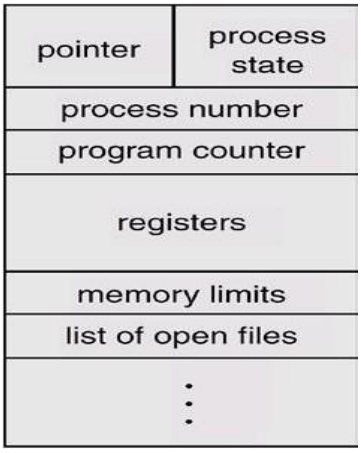


Figure: Process control block (PCB).

i. Process state
ii. Program counter
iii. CPU registers
iv. CPU scheduling information
v. Memory-management information
vi. Accounting information
vii. I/O status information

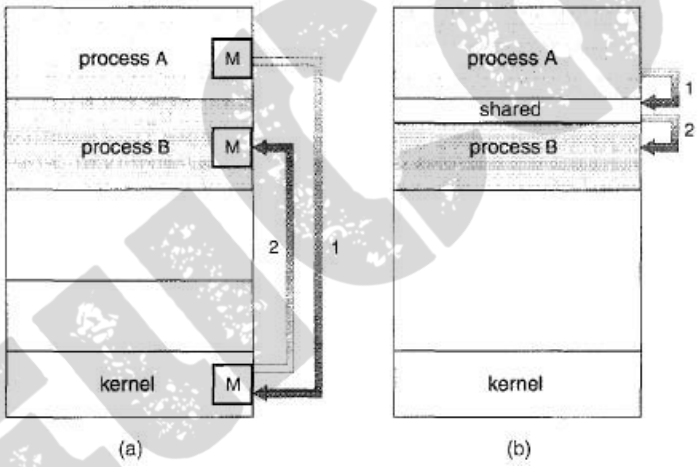
- **Process state:** The state may be new, ready, running, waiting, halted, and SO on. Program counter: The counter indicates the address of the next instruction to be executed for this process.
- **CPU registers:** The registers vary in number and type, depending on the computer architecture. They include accumulators, index registers, stack pointers, and general-purpose registers, plus any condition-code information.
- **CPU-scheduling information:** This information includes a process priority, pointers to scheduling queues, and any other scheduling parameters.
- **Memory-management information**: This information may include such information as the value of the base and limit registers, the page tables, or the segment tables, depending on the memory system used by the operating system.
- **Accounting information:** This information includes the amount of CPU and real time used, time limits, account numbers, job or process numbers, and so on.
- **Status information:** The information includes the list of I/O devices allocated to this process, a list of open files, and so on.

(b). What is inter-process communication? Discuss message passing and the shared memory concept of IPC.

Concurrent execution of cooperating processes requires mechanisms that allow processes to communicate with one another and to synchronize their actions. Cooperating processes require an inter-process communication (IPC) mechanism that will allow them to exchange data and information. There are two fundamental models of inter-process communication:

- shared memory
- message passing

In the shared-memory model, a region of memory that is shared by cooperating processes is established. Processes can then exchange information by reading and writing data to the shared region. In the message-passing model, communication takes place by means of messages exchanged between the cooperating processes. The two communications models are contrasted in Figure below.



Message passing Shared memory

Shared-Memory Systems: Inter-process communication using shared memory requires communicating processes to establish a region of shared memory. Typically, a shared-memory region resides in the address space of the process creating the shared-memory segment. Other processes that wish to communicate using this shared-memory segment must attach it to their address space. Shared memory requires that two or more processes agree to remove this restriction. They can then exchange information by reading and writing data in the shared areas. Two types of buffers can be used. The unbounded buffer places no practical limit on the size of the buffer. The consumer may have to wait for new items, but the

producer can always produce new items. The bounded buffer assumes a fixed buffer size. In this case, the consumer must wait if the buffer is empty, and the producer must wait if the buffer is full.

Message-Passing Systems: Message passing provides a mechanism to allow processes to communicate and to synchronize their actions without sharing the same address space and is particularly useful in a distributed environment, where the communicating processes may reside on different computers connected by a network. A message-passing facility provides at least two operations: send (message) and receive (message). Messages sent by a process can be of either fixed or variable size. If only fixed-sized messages can be sent, the system-level implementation is straightforward. This restriction, however, makes the task of programming more difficult. If processes P and Q want to communicate, they must send messages to and receive messages from each other; a communication link must exist between them. Here are several methods for logically implementing a link and the send () / receive () operations:

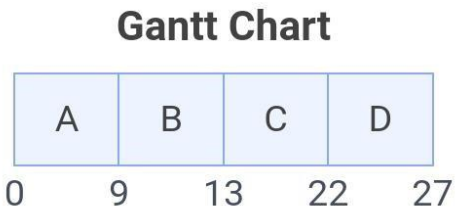
- Direct or indirect communication
- Synchronous or asynchronous communication
- Automatic or explicit buffering

(c). Calculate average waiting and turnaround times by drawing the Gantt chart using FCFS and RR (q=2ms).

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

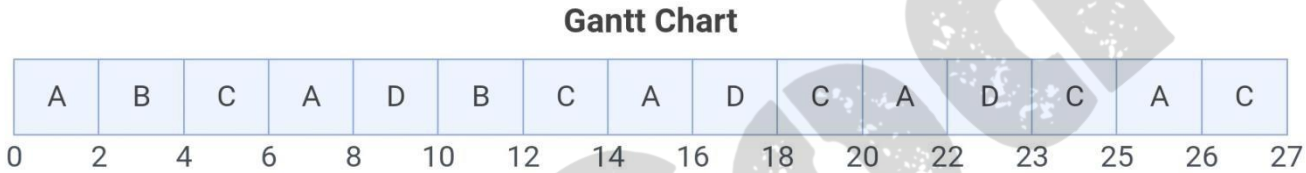
Solution:

FCFS Scheduling:



Job	Arrival Time	Burst Time	Finish Time	Turnaround Time	Waiting Time
A	0	9	9	9	0
B	1	4	13	12	8
C	2	9	22	20	11
D	3	5	27	24	19
			Average	65 / 4 = 16.25	38 / 4 = 9.5

RR Scheduling:



Job	Arrival Time	Burst Time	Finish Time	Turnaround Time	Waiting Time
A	0	9	26	26	17
B	1	4	12	11	7
C	2	9	27	25	16
D	3	5	23	20	15
			Average	82 / 4 = 20.5	55 / 4 = 13.75

OR

4. (a). Discuss in detail the multithreading model, its advantages and disadvantages with suitable illustration.

Many-to-One Model: The many-to-one model (Figure below) maps many user-level threads to one kernel thread. Thread management is done by the thread library in user space, so it is efficient; but the entire process will block if a thread makes a blocking system call. Also, because only one thread can access the kernel at a time, multiple threads are unable to run in parallel on multiprocessors. Green threads-a thread library available for Solaris-uses this model, as does GNU Portable Threads.

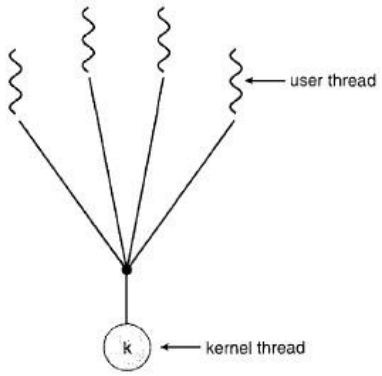


Figure: Many-to-one model.

One-to-One Model: The one-to-one model (Figure below) maps each user thread to a kernel thread. It provides more concurrency than the many-to-one model by allowing another thread to run when a thread makes a blocking system call; it also allows multiple threads to run in parallel on multiprocessors. The only drawback to this model is that creating a user thread requires creating the corresponding kernel thread. Because the overhead of creating kernel threads can burden the performance of an application, most implementations of this model restrict the number of threads supported by the system. Linux, along with the family of Windows operating systems including Windows 95, 98, NT, 2000, and XP implement the one-to-one model.

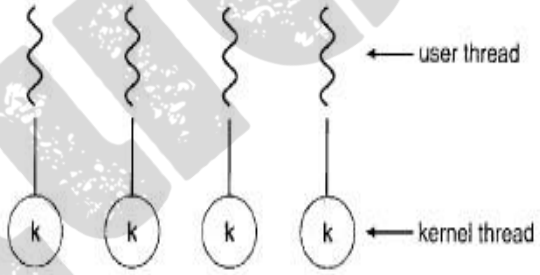


Figure: One-to-one model.

Many-to-Many Model: The many-to-many model (Figure below) multiplexes many user-level threads to a smaller or equal number of kernel threads. The number of kernel threads may be specific to either a particular application or a particular machine (an application may be allocated more kernel threads on a multiprocessor than on a uniprocessor).

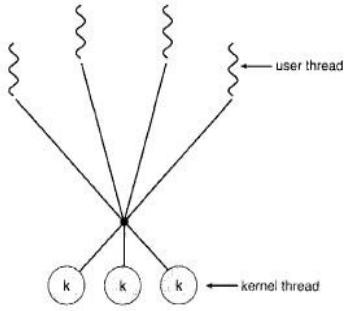


Figure: Many-to-Many model.

One popular variation on the many-to-many model still multiplexes many user level threads to a smaller or equal number of kernel threads but also allows a user-level thread to be bound to a kernel thread. This variation, sometimes referred to as the *two-level model* (Figure below), is supported by operating systems such as IRIX, HP-UX, and Tru64 UNIX.

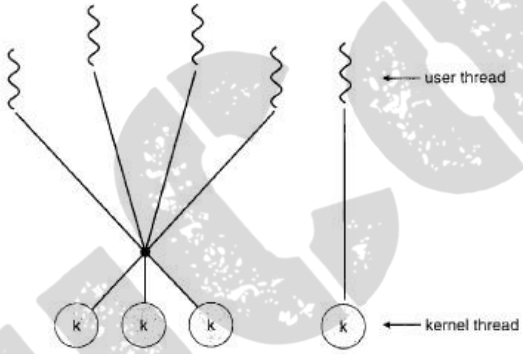


Figure: Two-level model.

(b). Explain five different scheduling criteria used in the computing scheduling mechanism.

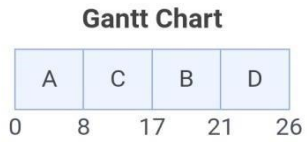
Many criteria have been suggested for comparing CPU scheduling algorithms. The criteria include the following:

- **CPU utilization**. We want to keep the CPU as busy as possible. Conceptually, CPU utilization can range from 0 to 100 percent.
- **Throughput**. If the CPU is busy executing processes, then work is being done. One measure of work is the number of processes that are completed per time unit, called ***throughput***.
- **Turnaround time**. From the point of view of a particular process, the important criterion is how long it takes to execute that process. The interval from the time of submission of a process to the time of completion is the turnaround time.
- **Waiting time**. The CPU scheduling algorithm does not affect the amount of time during which a process executes or does I/O; it affects only the amount of time that a process spends waiting in the ready queue.
- **Response time**. In an interactive system, turnaround time may not be the best criterion. Often, a process can produce some output fairly early and can continue computing new results while previous results are being output to the user. Thus, another measure is the time from the submission of a request until the first response is produced. This measure, called response time, is the time it takes to start responding, not the time it takes to output the response. The turnaround time is generally limited by the speed of the output device.

c) Calculate the average waiting time and the average turnaround time by drawing the Gantt chart using SRTF and the Priority scheduling algorithm.

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

SRTF Algorithm



Job	Arrival Time	Burst Time	Finish Time	Turnaround Time	Waiting Time
A	0	8	8	8	0
B	1	4	21	20	16
C	2	9	17	15	6
D	3	5	26	23	18
			Average	66 / 4 = 16.5	40 / 4 = 10

Priority Based Scheduling

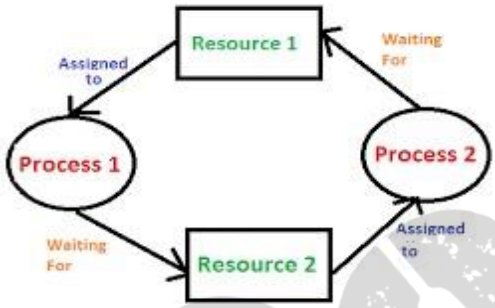
Job	Arrival Time	Burst Time	Finish Time	Turnaround Time	Waiting Time
A	0	8	17	17	9
B	1	4	5	4	0
C	2	9	26	24	15
D	3	5	10	7	2
			Average	52 / 4 = 13	26 / 4 = 6.5

Note: A = P1, B=P2, C=P3 and D=P4

MODULE-3

5.(a). Define deadlock. What are the necessary conditions for deadlock to occur?

In a multiprogramming environment, several processes may compete for a finite number of resources. A process requests resources; and if the resources are not available at that time, the process enters a waiting state. Sometimes, a waiting process is never again able to change state, because the resources it has requested are held by other waiting processes. This situation is called a deadlock.



In a deadlock, processes never finish executing, and system resources are tied up, preventing other jobs from starting necessary conditions. A deadlock situation can arise if the following four conditions hold simultaneously in a system:

- **Mutual Exclusion Condition**: At least one resource must be held in a non-shareable mode, that is, only one process at a time claims exclusive control of the resource. If another process requests that resource, the requesting process must be delayed until the resource has been released.
- **Hold and Wait Condition**: Requesting process hold already the resources while waiting for requested resources. There must exist a process that is holding a resource already allocated to it while waiting for additional resource that are currently being held by other processes.
- **No-Premptive Condition**: Resources already allocated to a process cannot be prempted. Resources cannot be removed from the processes are used to completion or released voluntarily by the process holding it.
- **Circular Wait Condition**: The processes in the system form a circular list or chain where each process in the list is waiting for a resource held by the next process in the list. There exists a set {P0, P1, …, P0} 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 P0 is waiting for a resource that is held by P0.

(b). Illustrate Peterson's solution for the critical section problem.

A classic software-based solution to the critical-section problem known as Peterson's solution. Peterson's solution is restricted to two processes that alternate execution between their critical sections and remainder sections. The processes are numbered P_0 and P_1. For convenience, when presenting *Pi,* we use *Pj* to denote the other process; that is, j equals 1 - i. Peterson's solution requires two data items to be shared between the two processes: int turn; boolean flag[2]; The variable turn indicates whose turn it is to enter its critical section. That is, if turn == i, then process P_i is allowed to execute in its critical section. The flag array is used to indicate if a process *is ready* to enter its critical section.

For example, if flag [i] is true, this value indicates that P_i is ready to enter its critical section. With an explanation of these data structures complete, we are now ready to describe the algorithm shown in Figure below.

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

        critical section

    flag[i] = FALSE;

        remainder section

} while (TRUE);
```

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

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

To prove property ***Mutual exclusion is preserved***, we note that each P_i 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 [i] ==flag [j] == true. These two observations imply that P_0 and P_1 could not have successfully executed their while statements at about the same time, since the value of turn can be either 0 or 1, but cannot be both.

(c). Consider the following snapshot of a system:

	Allocation				Max				Available			
	A	B	C	D	A	B	C	D	A	B	C	D
P_0	0	0	1	2	0	0	1	2	1	5	2	0
P_1	1	0	0	0	1	7	5	0				
P_2	1	3	5	4	2	3	5	6				
P_3	0	6	3	2	0	6	5	2				
P_4	0	0	1	4	0	6	5	6				

Answer the following questions using the banker's algorithm:

a) What is the content of the matrix *Need?*

b) Is the system in a safe state?

c) If a request from process PI arrives for (0, 4, 2, 0), can the request be granted immediately?

Solution:

a) Since, Need = Max − Allocation, the content of Need is

A	B	C	D
0	0	0	0
0	7	5	0
1	0	0	2
0	0	2	0
0	6	4	2

b) Yes, the sequence <*P*0, *P*2, *P*1, *P*3, *P*4> satisfies the safety requirement.

c) Yes. Since

i. (0,4,2,0) _ *Available* = (1,5,2,0)
ii. (0,4,2,0) _ *Maxi* = (1,7,5,0)
iii. The new system state after the allocation is made is

	Allocation	***Max***	***Need***	***Available***
	A B C D	***A B C D***	***A B C D***	***A B C D***
*P*0	0 0 1 2	0 0 1 2	0 0 0 0	1 1 0 0
*P*1	1 4 2 0	1 7 5 0	0 3 3 0	
*P*2	1 3 5 4	2 3 5 6	1 0 0 2	
*P*3	0 6 3 2	0 6 5 2	0 0 2 0	
*P*4	0 0 1 4	0 6 5 6	0 6 4 2	

and the sequence < *P*0, *P*2, *P*1, *P*3, *P*4 > satisfies the safety requirement.

OR

6.(a). Explain different methods to recover from deadlocks.

- When a detection algorithm determines that a deadlock exists, several alternatives are available.
- One possibility is to inform the operator that a deadlock has occurred and to let the operator deal with the deadlock manually.
- Another possibility is to let the system *recover* from the deadlock automatically.
- There are two options for breaking a deadlock. One is simply to abort one or more processes to break the circular wait.
- The other is to pre-empt some resources from one or more of the deadlocked processes.

Process Termination: To eliminate deadlocks by aborting a process, we use one of two methods. 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 great expense; the deadlocked processes may have computed for a long time, and the results of these partial computations must be discarded and probably will have to 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.

Aborting a process may not be easy. If the process was in the midst of updating a filet terminating it will leave that file in an incorrect state.

Similarly, if the process was in the midst of printing data on a printer, the system must reset the printer to a correct state before printing the next job. If the partial termination method is used, then we must determine which deadlocked process (or processes) should be terminated. This determination is a policy decision, similar to CPU-scheduling decisions. The question is basically an economic one; we should abort those processes whose termination will incur the minimum cost.

Resource Preemption: To eliminate deadlocks using resource preemption, we successively preempt some resources from processes and give these resources to other processes until the deadlock cycle is broken. If preemption is required to deal with deadlocks, then three issues need to be addressed:

- **Selecting a victim**. Which resources and which processes are to be preempted? As in process termination, we must determine the order of preemption to minimize cost.

- **Rollback**. If we preempt a resource from a process, 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.
- **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?

In a system where victim selection is based primarily on cost factors, it may happen that the same process is always picked as a victim. As a result, this process never completes its designated task, a starvation situation that must be dealt with in any practical system.

(b). What is a resource allocation graph? Consider an example to explain how it is very useful in describing a deadly embrace.

System resource-allocation graph consists of a set of vertices *V* and a set of edges *E*.

V is partitioned into two types:

$P = \{P_1, P_2, \ldots, P_n\}$, the set consisting of all the processes in the system

$R = \{R_1, R_2, \ldots, R_m\}$, the set consisting of all resource types in the system

A directed edge from process *Pi* to resource type *Rj* is denoted by

$$P_i \longrightarrow R_j$$

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

$$R_j \longrightarrow P_i$$

it signifies that an instance of resource type *Rj* has been allocated to process *Pi*. A directed edge ***Pi*** $\longrightarrow$ ***Rj*** is called a request edge; a directed edge $R_j \longrightarrow P_i$ is called an **assignment edge**. The resource-allocation graph shown in below depicts the following situation

- The sets P, R, and E:
  - $P = \{P_1, P_2, P_3\}$
  - $R = \{R_1, R_2, R_3, R_4\}$
  - $E = \{P_1 \rightarrow R_1, P_2 \rightarrow R_3, R_1 \rightarrow P_2, R_2 \rightarrow P_2, R_2 \rightarrow P_1, R_3 \rightarrow P_3\}$
- Resource instances:
  - One instance of resource type R_1
  - Two instances of resource type R_2
  - One instance of resource type R_3
  - Three instances of resource type R_4
- Process states:
  - Process P_1 is holding an instance of resource type R_2 and is waiting for an instance of resource type R_1.
  - Process P_2 is holding an instance of R_1 and an instance of R_2 and is waiting for an instance of R_3.
  - Process P_3 is holding an instance of R_3.

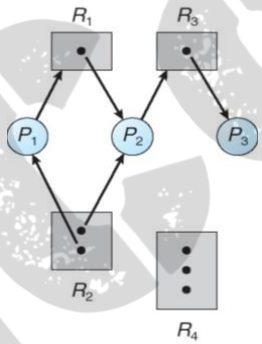


Figure: Resource-allocation graph.

- ❖ Given the definition of a resource-allocation graph, it can be shown that, if the graph contains no cycles, then no process in the system is deadlocked.
- ❖ If the graph does contain a cycle, then a deadlock may exist.
- ❖ If each resource type has exactly one instance, then a cycle implies that a deadlock has occurred.
- ❖ 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.
- ❖ In this case, a cycle in the graph is both a necessary and a sufficient condition for the existence of deadlock.
- ❖ If each resource type has several instances, then a cycle does not necessarily imply that a deadlock has occurred. In this case, a cycle in the graph is a necessary but not a sufficient condition for the existence of deadlock.

(c). What is a semaphore? State a Dining Philosopher problem gives a solution using 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 definition of wait in pseudo code is

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

The classical definitions of signal in pseudo code is

```
Signal(S)
{
    S++;
}
```

The dining-philosophers problem is considered a classic synchronization problem neither because of its practical importance nor because computer scientists dislike philosophers but because it is an example of a large class of concurrency-control problems. It is a simple representation of the need to allocate several resources among several processes in a deadlock-free and starvation-free manner.

One simple solution is to represent each chopstick with a semaphore. A philosopher tries to grab a chopstick by executing a wait () operation on that semaphore; she releases her chopsticks by executing the signal () operation on the appropriate semaphores.

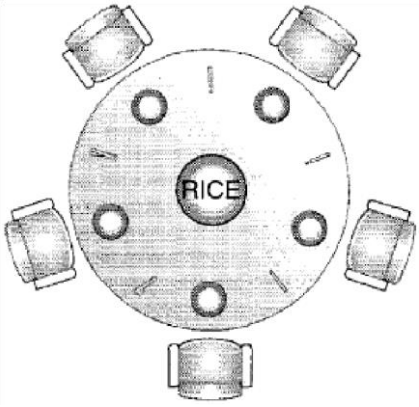


Figure: The situation of the dining philosophers.

Thus, the shared data are

semaphore chopstick [5] ;

where all the elements of chopstick are initialized to 1. The structure of philosopher i is shown in Figure 2. Although this solution guarantees that no two neighbours are eating simultaneously, it nevertheless must be rejected because it could create a deadlock. Suppose that all five philosophers become hungry simultaneously and each grabs her left chopstick. All the elements of chopstick will now be equal to O. When each philosopher tries to grab her right chopstick, she will be delayed forever.

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

Figure 2 The structure of philosopher i.

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 (to do this she must pick them up in a critical section).
- 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.

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

MODULE 4

7. (a). What is TLB? Explain TLB in detail with a paging system with a neat diagram.

The standard solution to this problem is to use a special, small, fast lookup hardware cache, called a translation look-aside buffer (TLB).

- The TLB is associative, high-speed memory. Each entry in the TLB consists of two parts: a key (or tag) and a value.
- When the associative memory is presented with an item, the item is compared with all keys simultaneously.
- The TLB is used with page tables in the following way. The TLB contains only a few of the page-table entries. When a logical address is generated by the CPU, its page number is presented to the TLB.
- If the page number is found, its frame number is immediately available and is used to access memory.
- The whole task may take less than 10 percent longer than it would if an unmapped memory reference were used.
- If the page number is not in the TLB (known as a TLB miss), a memory reference to the page table must be made. When the frame number is obtained, we can use it to access memory (Figure below).

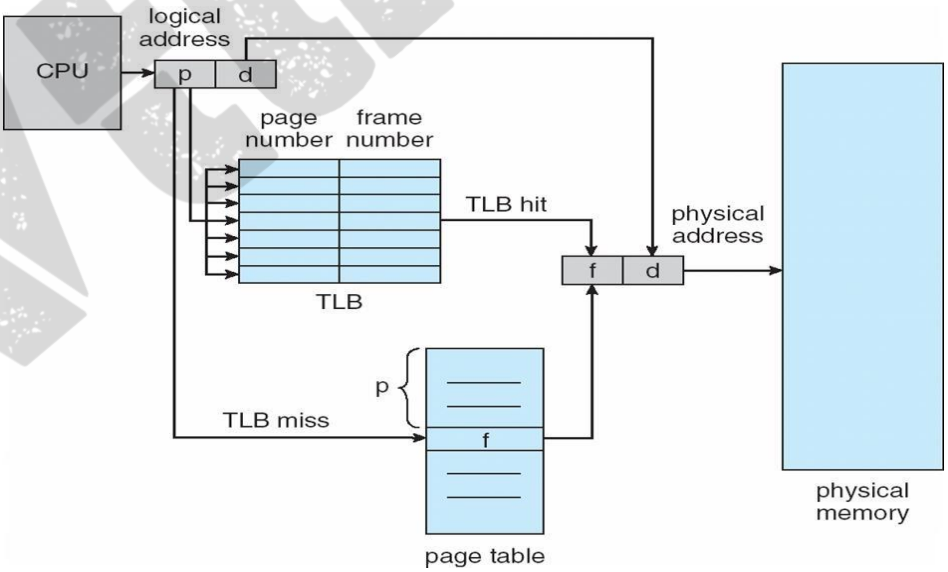


Figure: TLB

(b). With the help of a neat diagram, explain the various steps of address binding.

Address binding of instructions and data to memory addresses can happen at three different stages.

- **Compile time:** The compile time is the time taken to compile the program or source code. During compilation, if memory location known a priori, then it generates absolute codes.
- **Load time:** It is the time taken to link all related program file and load into the main memory. It must generate relocatable code if memory location is not known at compile time.
- **Execution time:** It is the time taken to execute the program in main memory by processor. Binding delayed until run time if the process can be moved during its execution from one memory segment to another. Need hardware support for address maps (e.g., base and limit registers).

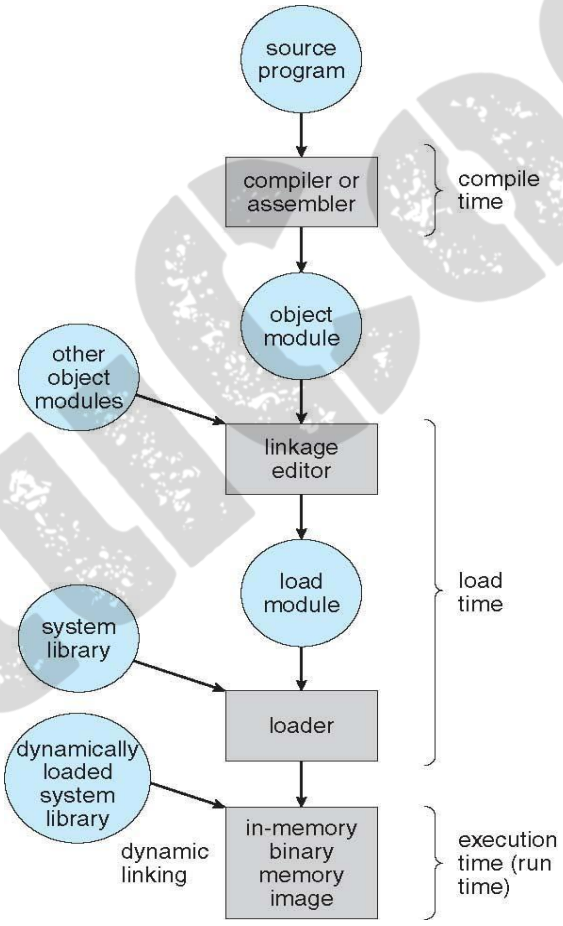


Figure: Address Binding

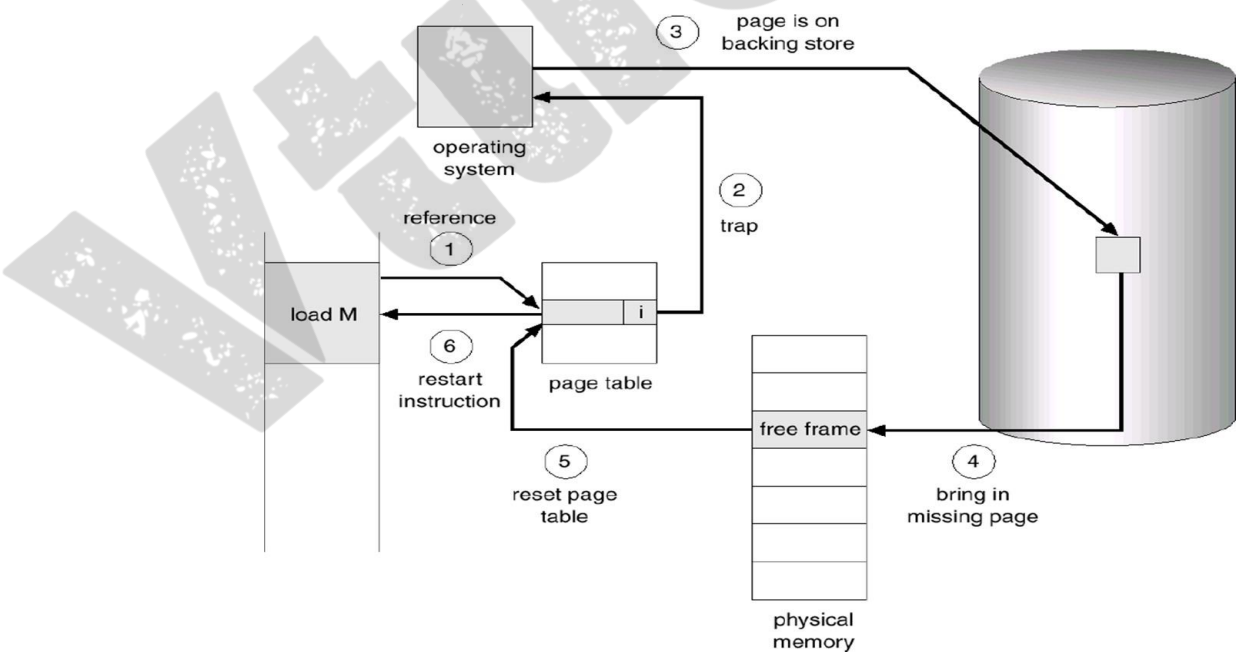
(c). Consider the page reference string: 1,0,7,1,0,2,1,2,3,0,3,2,4,0,3,6,2,1 for a memory with three frames. Determine the number of page faults using the FIFO, Optimal, and LRU replacement algorithms. Which algorithm is most efficient?

Solve by yourself

OR

8. (a). What is demand paging? Explain the steps in handling page faults using the appropriate diagram.

- Consider how an executable program might be loaded from disk into memory.
- Loading the entire program into memory results in loading the executable code for *all* options, regardless of whether an option is ultimately selected by the user or not.
- An alternative strategy is to initially load pages only as they are needed.
- This technique is known as demand paging and is commonly used in virtual memory systems.
- With demand-paged virtual memory, pages are only loaded when they are demanded during program execution; pages that are never accessed are thus never loaded into physical memory.
- The paging hardware, in translating the address through the page table, will notice that the invalid bit is set, causing a trap to the operating system.
- 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 straightforward (Figure below): a valid or an invalid memory access.
- If the reference was invalid, we terminate the process. If it was valid, but we have not yet brought in that page, we now page it in.
- We find a free frame (by taking one from the free-frame list, for example).
- We schedule a disk operation to read the desired page into the newly allocated frame.
- 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.
- We restart the instruction that was interrupted by the trap. The process can now access the page as though it had always been in memory.

A page fault causes the following sequence to occur:

a) **Trap** to the operating system.

b) Save the **user registers and process state**.

c) Determine that the **interrupt was a page fault**.

d) Check that the **page reference was legal and determine the location of the page on the disk**.

e) Issue a read from the **disk to a free frame**:

f) Wait in a queue for this device **until the read request is serviced**.

g) Wait for the device **seek and / or latency time**.

h) Begin the **transfer of the page to a free frame**.

i) While waiting, allocate the CPU to some other user (CPU scheduling, optional).

(b). What is segmentation? Explain the basic method of segmentation with an example.

An important aspect of memory management that became unavoidable with paging is the separation of the user's view of memory and the actual physical memory. The user's view of memory is not the same as the actual physical memory. The user's view is mapped onto physical memory. This mapping allows differentiation between logical memory and physical memory.

Basic Method:

Do users think of memory as a linear array of bytes, some containing instructions and others containing data? Most people would say no. Rather users prefer to view memory as a collection of **variable-sized segments**, with **no necessary ordering among segments** (Figure 4.15).

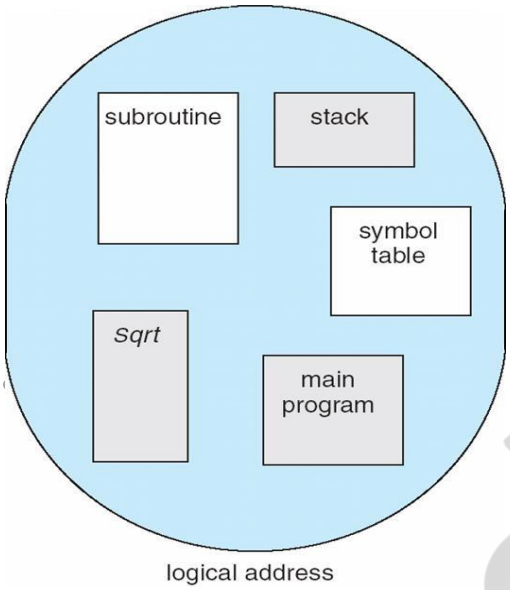


Figure: User's view of a program.

- 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.
- (Contrast this scheme with the paging scheme, in which the user specifies only a single address, which is partitioned by the hardware into a page number and an offset, all invisible to the programmer.)

For simplicity of implementation, segments are numbered and are referred to by a segment number, rather than 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.

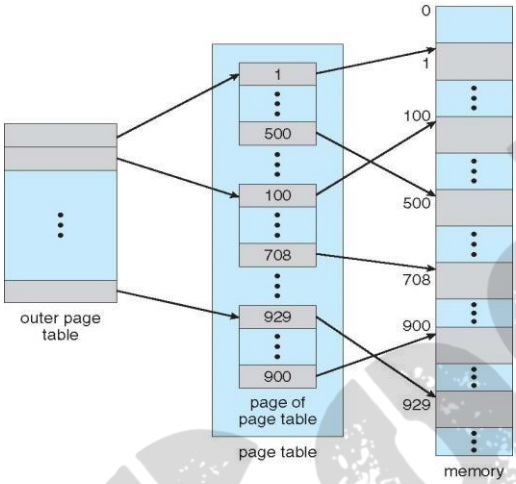
A C compiler might create **separate segments for the following**:

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

(c). Discuss the structure of the page table with a suitable diagram.

Hierarchical Page Tables:

- ✦ Break up the logical address space into multiple page tables
- ✦ A simple technique is a two-level page table
- ✦ We then page the page table



A logical address (on 32-bit machine with 1K page size) is divided into:

- ✦ a page number consisting of 22 bits
- ✦ a page offset consisting of 10 bits

Since the page table is paged, the page number is further divided into:

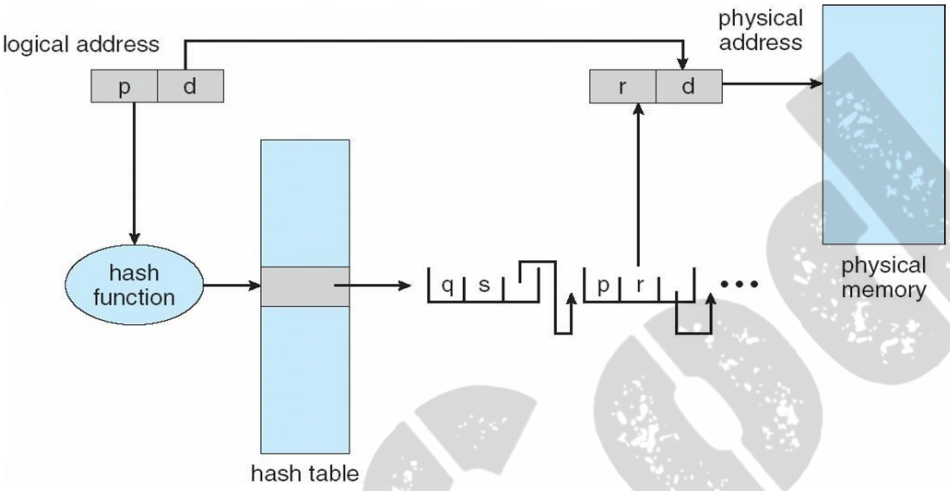
- ✦ a 12-bit page number
- ✦ a 10-bit page offset

Thus, a logical address is as follows:

page number		page offset
p_1	p_2	d
12	10	10

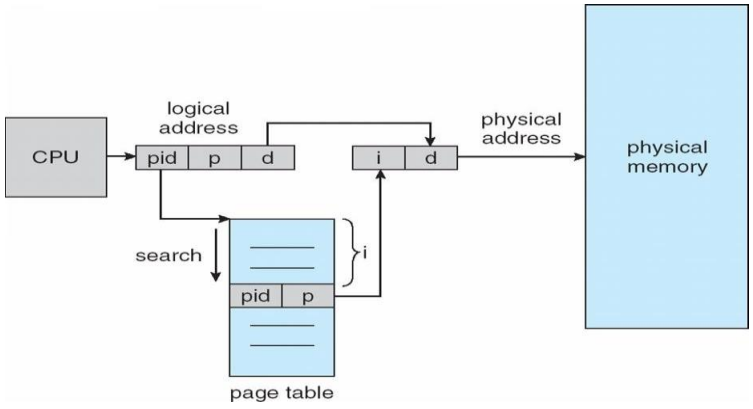
Hashed Page Tables: A common approach for handling address spaces larger than 32 bits is to use a hashed page table, with the hash value being the virtual page number. Each entry in the hash table contains a linked list of elements that hash to the same location (to handle collisions).

Each element consists of three fields: (1) the virtual page number, (2) the value of the mapped page frame, and (3) a pointer to the next element in the linked list. The algorithm works as follows: The virtual page number in the virtual address is hashed into the hash table. The virtual page number is compared with field 1 in the first element in the linked list. If there is a match, the corresponding page frame (field 2) is used to form the desired physical address. If there is no match, subsequent entries in the linked list are searched for a matching virtual page number.



Inverted Page Table

- Rather than each process having a page table and keeping track of all possible logical pages, track all physical pages o One entry for each real page of memory.
- Entry consists of the virtual address of the page stored in that real memory location, with information about the process that owns that page.
- Decreases memory needed to store each page table, but increases time needed to search the table when a page reference occurs o Use hash table to limit the search to one — or at most a few page-table entries.



MODULE-5

9. (a). What is a file? What are its attributes? Explain file operations.

A file is a named collection of related information that is recorded on secondary storage. The information in a file is defined by its creator. 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.

File Attributes: A file is named, for the convenience of its human users, and is referred to by its name. A file's attributes vary from one operating system to another but typically consist of these:

- **Name**. The symbolic file name is the only information kept in human readable form.
- **Identifier.** This unique tag, usually a number identifies the file within the file system; it is the non-human-readable name for the file.
- **Type**. This information is needed for systems that support different types of files.
- **Location**. This information is a pointer to a device and to the location of the file on that device.
- **Size.** The current size of the file (in bytes, words, or blocks) and possibly the maximum allowed size are included in this attribute.
- **Protection.** Access-control information determines who can do reading, writing, executing, and so on.
- **Time, date, and user identification**: This information may be kept for creation, last modification, and last use. These data can be useful for protection, security, and usage monitoring.

File operations: The operating system can provide system calls to create, write, read, reposition, delete, and truncate files.

- **Creating a file.** Two steps are necessary to create a file. First, space in the file system must be found for the file. Second, an entry for the new file must be made in the directory.
- **Writing a file.** To write a file, we make a system call specifying both the name of the file and the information to be written to the file.
- **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.

- **Repositioning within a file**. The directory is searched for the appropriate entry, and the current-file-position pointer is repositioned to a given value, Repositioning within a file need not 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, so that it can be reused by other files, and erase the directory entry.
- **Truncating a file**. The user may want to erase the contents of a file but keep its attributes. Rather than forcing the user to delete the file and then recreate it, this function allows all attributes to remain unchanged-except for file length-but lets the file be reset to length zero and its file space released.

(b). Explain in detail about various file operations in a file system.

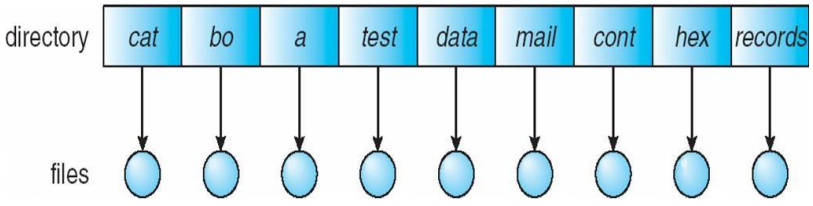
Basic file system operations include file creation, file reading, file writing, file deletion, and directory traversal.

- **File creation:** This is the act of making a new file in the file system.
- **File reading:** In this operation, the file system provides access to the contents of a file.
- **File writing:** Here, existing content in a file can be updated or new content can be added.
- **File deletion:** Deleting a file removes it from the file system.
- **Directory traversal:** This is the ability to navigate through directories or folders in the file system**.**

(c). Discuss various directory structures with neat diagrams.

Following are the most common schemes for defining the logical structure of a directory.

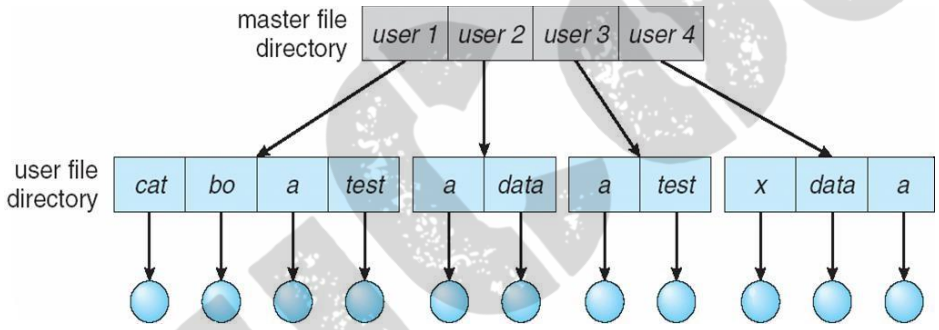
Single Level: The simplest directory structure is the single-level directory. All files are contained in the same directory, which is easy to support and understand (Figure below).



A single-level directory has significant limitations, however when the number of files increases or when the system has more than one user. Since all files are in the same directory, they must have unique names. If two users call their data file *test,* then the unique-name rule is violated.

Two-Level Directory

- As *we* have seen, a single-level directory often leads to confusion of file names among different users.
- The standard solution is to create a *separate* directory for each user.
- In the two-level directory structure, each user has his own user file directory (UFD). The UFDs have similar structures, but each list only the files of a single user.
- When a user job starts or a user log in, the system's master file directory (MFD) is searched.
- The MFD is indexed by user name or account number, and each entry points to the UFD for that user (Figure below).

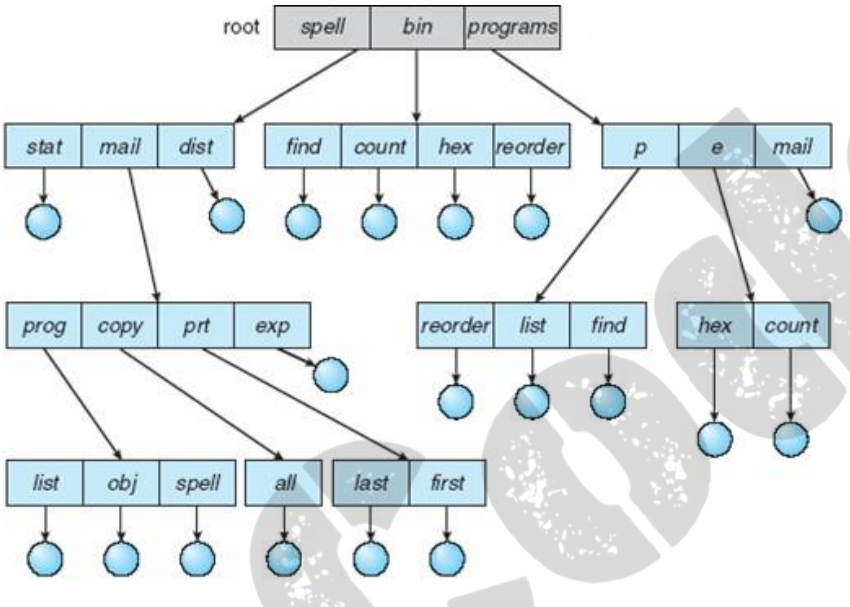


Although the two-level directory structure solves the name-collision problem, it still has disadvantages. This structure effectively isolates one user from another. Isolation is an advantage when the users are completely independent but is a disadvantage when the users *want* to cooperate on some task and to access one another's files.

Tree-Structured Directories

- Once we have seen how to view a two-level directory as a two-level tree, the natural generalization is to extend the directory structure to a tree of arbitrary height (Figure below).
- This generalization allows users to create their own subdirectories and to organize their files accordingly.
- A tree is the most common directory structure. The tree has a root directory, and every file in the system has a unique path name.

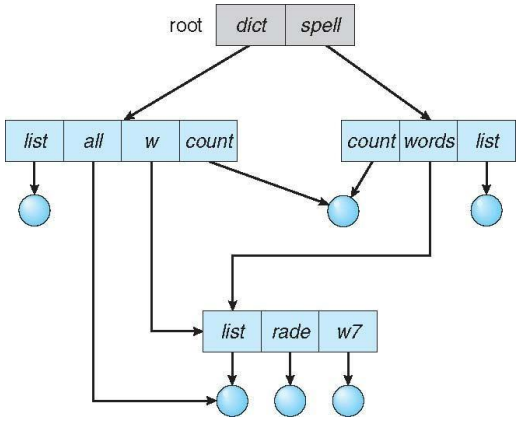
- A directory (or subdirectory) contains a set of files or subdirectories.
- A directory is simply another file, but it is treated in a special way. All directories have the same internal format. One bit in each directory entry defines the entry as a file (0) or as a subdirectory (1). Special system calls are used to create and delete directories.



- In normal use, each process has a current directory. The current directory should contain most of the files that are of current interest to the process.
- When reference is made to a file, the current directory is searched.
- If a file is needed that is not in the current directory, then the user usually must either specify a path name or change the current directory to be the directory holding that file.
- To change directories, a system call is provided that takes a directory name as a parameter and uses it to redefine the current directory.

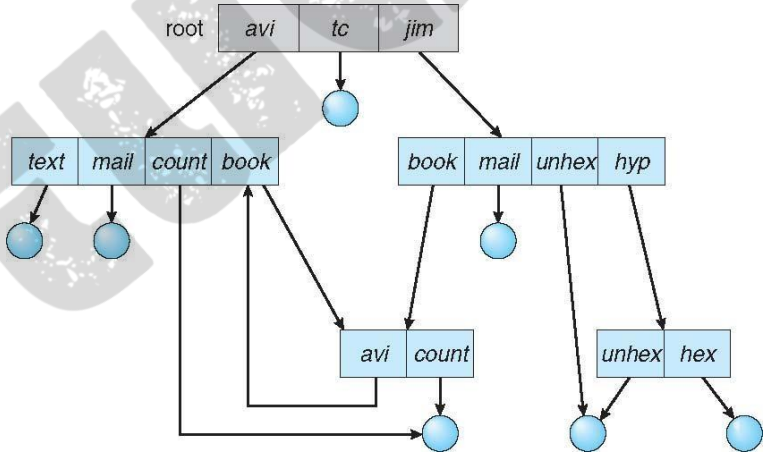
Acyclic-Graph Directories

- A tree structure prohibits the sharing of files or directories. An acyclic graph that is, a graph with no cycles-allows directories to share subdirectories and files (Figure below).
- The *same* file or subdirectory may be in two different directories. The acyclic graph is a natural generalization of the tree-structured directory scheme.



General Graph Directory

- A serious problem with using an acyclic-graph structure is ensuring that there are no cycles.
- If we start with a two-level directory and allow users to create subdirectories, a tree-structured directory result.
- It should be fairly easy to see that simply adding new files and subdirectories to an existing tree-structured directory preserves the tree-structured nature. However, when we add links to an existing tree-structured directory, the tree structure is destroyed, resulting in a simple graph structure (Figure below).

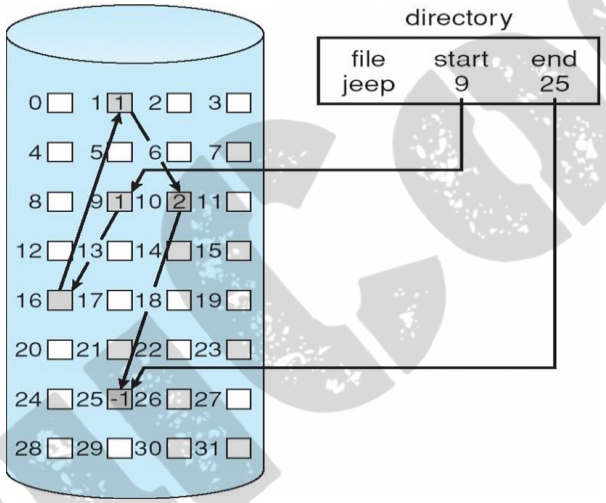


OR

10. (a). Explain contiguous and linked disk space allocation methods.

Linked Allocation

- Linked allocation solves *all problems of contiguous allocation*. With linked allocation, each file is a linked list of disk blocks; the disk blocks may be scattered anywhere on the disk.
- The directory contains a pointer to the first and last blocks of the file.
- For example, a file of five blocks might start at block 9 and continue at block 16, then block I, then block 10, and finally block 25 (Figure below).
- Each block contains a pointer to the next block. These pointers are not made available to the user.
- Thus, if each block is 512 bytes in size, and a disk address (the pointer) requires 4 bytes, then the user sees blocks of 508 bytes.



- To create a new file, we simply create a new entry in the directory. With linked allocation, each directory entry has a pointer to the first disk block of the file.
- This pointer is initialized to *nil* (the end-of-list pointer value) to signify an empty file. The size field is also set to O.
- A write to the file causes the free-space management system to find a free block, and this new block is written to and is linked to the end of the file.
- To read a file, we simply read blocks by following the pointers from block to block.
- There is no external fragmentation with linked allocation, and any free block on the freespace list can be used to satisfy a request.
- The size of a file need not be declared when that file is created.

Contiguous Allocation:

- Contiguous allocation requires that each file occupy a set of contiguous blocks on the disk.

- Disk addresses **define a linear ordering on the disk**. With this ordering, assuming that only one job is accessing the disk, accessing block ***b* + 1 after block *b* normally requires** no head movement.
- When **head movement is needed (from the last sector of one cylinder to the first sector of the next cylinder),** the head need only move from one track to the next.
- Contiguous allocation of a file is defined by the **disk address and length (in block units) of the first block**. If the file is n blocks long and starts at location 17, then it **occupies blocks b, b + 1, b + 2, ..., 17 + n - 1**.
- The directory entry for each file indicates the address of the starting block and the length of the area allocated for this file (Figure below 5.16).

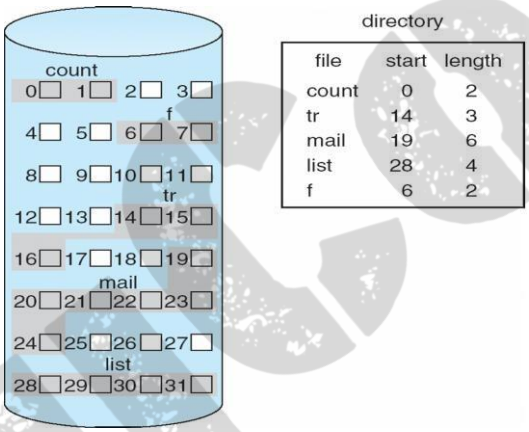


Figure: Contiguous allocation of disk space.

Accessing a file that has been allocated contiguously is easy. For sequential access, the file system remembers the disk address of the last block referenced and, when necessary, reads the next block.

For direct access to block *i* of a file that starts at block b, we can immediately access block *b* + *i*. Thus, both sequential and direct access can be supported by contiguous allocation. Contiguous allocation has some problems, however. One difficulty is finding space for a new file. The system chosen to manage free space determines how this task is accomplished.

(b). Explain the access matrix method of system protection with the domain as objects and its implementation.

Access Matrix

- The model of protection can be viewed abstractly as a matrix, called an access matrix.
- The rows of the access matrix represent domains, and the columns represent objects.
- Each entry in the matrix consists of a set of access rights.
- The entry access(i,j) defines the set of operations that a process executing in domain D_i can invoke on object O_j.
- To illustrate these concepts, we consider the access matrix shown in Figure below.
- There are four domains and four objects-three files (F_1, F_2, F_3) and one laser printer. A process executing in domain D_1 can read files F_1 and F_3.
- A process executing in domain D_4 has the same privileges as one executing in domain D_1; but in addition, it can also write onto files F_1 and F_3.
- Note that the laser printer can be accessed only by a process executing in domain D_2.

object / domain	F_1	F_2	F_3	printer
D_1	read		read	
D_2				print
D_3		read	execute	
D_4	read write		read write	

- The access-matrix scheme provides us with the mechanism for specifying a variety of policies.
- The mechanism consists of implementing the access matrix and ensuring that the semantic properties *we* have outlined indeed hold.
- More specifically, we must ensure that a process executing in domain D_i can access only those objects specified in row *i,* and then only as allowed by the access-matrix entries.

- Implementation of Access Matrix.

How can the access matrix be implemented effectively? In general the matrix will be sparse; that is, most of the entries will be empty. Although data structure techniques are available for representing sparse matrices, they are not particularly useful for this application, because of the way in which the protection facility is used.

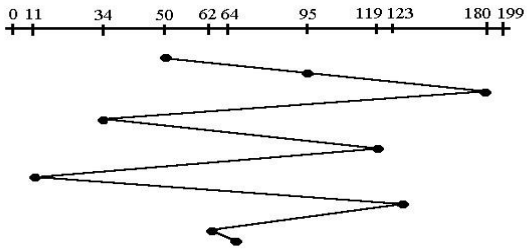
Methods:

- Global Table
- Access Lists for Objects
- Capability Lists for Domains
- A Lock-Key Mechanism

(c). Given the following sequences 95,180,34,119,11,123,62,64 with the track 50 and ending track 199. What is the total disk travelled by the disk arm using FCFS, SSTF, LOOK and CLOOK algorithm.

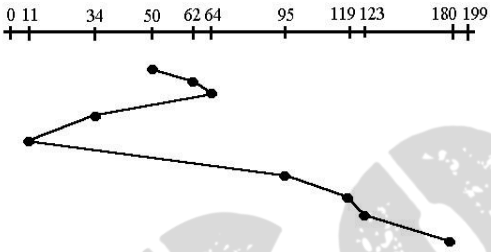
Given the following queue -- 95, 180, 34, 119, 11, 123, 62, 64 with the Read-write head initially at the track 50 and the tail track being at 199 let us now discuss the different algorithms.

First Come -First Serve (FCFS): All incoming requests are placed at the end of the queue. Whatever number that is next in the queue will be the next number served. Using this algorithm doesn't provide the best results. To determine the number of head movements you would simply find the number of tracks it took to move from one request to the next. For this case it went from 50 to 95 to 180 and so on. From 50 to 95 it moved 45 tracks. If you tally up the total number of tracks you will find how many tracks it had to go through before finishing the entire request. In this example, it had a total head movement of 640 tracks. The disadvantage of this algorithm is noted by the oscillation from track 50 to track 180 and then back to track 11 to 123 then to 64. As you will soon see, this is the worse algorithm that one can use.



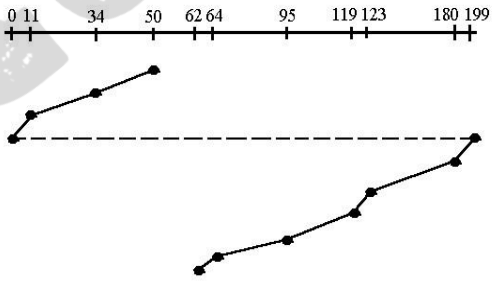
Shortest Seek Time First (SSTF):

In this case request is serviced according to next shortest distance. Starting at 50, the next shortest distance would be 62 instead of 34 since it is only 12 tracks away from 62 and 16 tracks away from 34. The process would continue until all the process are taken care of. For example the next case would be to move from 62 to 64 instead of 34 since there are only 2 tracks between them and not 18 if it were to go the other way. Although this seems to be a better service being that it moved a total of 236 tracks, this is not an optimal one. There is a great chance that starvation would take place. The reason for this is if there were a lot of requests close to each other the other requests will never be handled since the distance will always be greater.



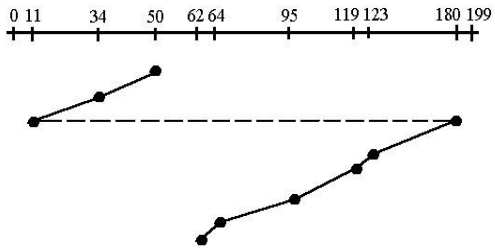
Circular Scan (C-SCAN):

Circular scanning works just like the elevator to some extent. It begins its scan toward the nearest end and works it way all the way to the end of the system. Once it hits the bottom or top it jumps to the other end and moves in the same direction. Keep in mind that the huge jump doesn't count as a head movement. The total head movement for this algorithm is only 187 track, but still this isn't the more sufficient.



C-LOOK

This is just an enhanced version of C-SCAN. In this the scanning doesn't go past the last request in the direction that it is moving. It too jumps to the other end but not all the way to the end. Just to the furthest request. C-SCAN had a total movement of 187 but this scan (C-LOOK) reduced it down to 157 tracks.



BEST OF LUCK

VTUCODE FAMILY😊😊😊