TED (10)-3071 Reg. No. …….

(REVISION-2010) Signature ……..……

FORTH SEMESTER DIPLOMA EXAMINATION IN ENGINEERING/ TECHNOLIGY- OCTOBER, 2012

OPERATING SYSTEM

(Common to CT and IF)

[*Time:* 3 hours

(Maximum marks: 100)

Marks

PART –A
(Maximum marks: 10)

I. Answer all questions in a sentence

1. Write the goals of OS.
    - Execute user programs and make solving user problems easier.
    - Make the computer system convenient to use
2. Write the necessary conditions for the occurrence of deadlock
    - Mutual exclusion
    - Hold and wait
    - No pre-emption
    - Circular wait
3. define thread

    The process model discussed so far has implied that a process is a program that performs a single **thread** of execution.

4. give two differences between logical and physical address

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.

5. define seek time

The positioning time, sometimes called the random-access time, consists of the time to move the disk arm to the desired cylinder, called the seek time

PART – B

II. Answer *any five* questions. Each question carries 6 marks

1. write the advantages and disadvantages of multiprocessing system

Multiprocessing is the simultaneous execution of two or more processes by a computer system having more than one CPU.

Advantages:

- Improves the performance of the system
- Efficient utilization of CPU and all other resources of the system
- It provides a built-in backup. If one of the CPUs breaks down, the other CPU(s) automatically takes over the complete workload

Disadvantages:

- Requires complex operating system to schedule, balance and coordinate the input, output and processing activities of multiple CPUs.
- Large main memory is required for accommodating the OS.
- Expensive

2. describe the general structure of PCB

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

- **Process state.** The state may be new, ready, running, and waiting, halted, and so on.

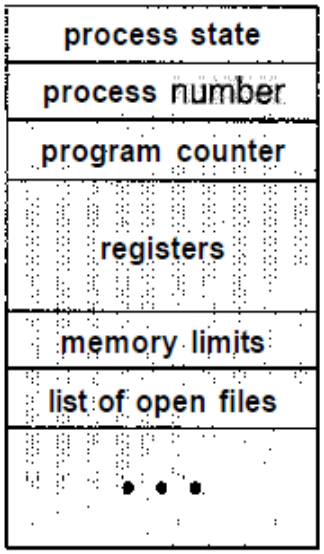
- **Program counter.** The counter indicates the address of the next instruction to be executed for this process.

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

- **CPU-scheduling information.** This information includes a process priority, pointers to scheduling queues, and any other scheduling parameters.

- **Accounting information.** This information includes the amount of CPU nd real time used, time limits, account members, job or process numbers, and so on.

- **I/O status information.** This information includes the list of I/O devices allocated to the process, a list of open files, and so on.

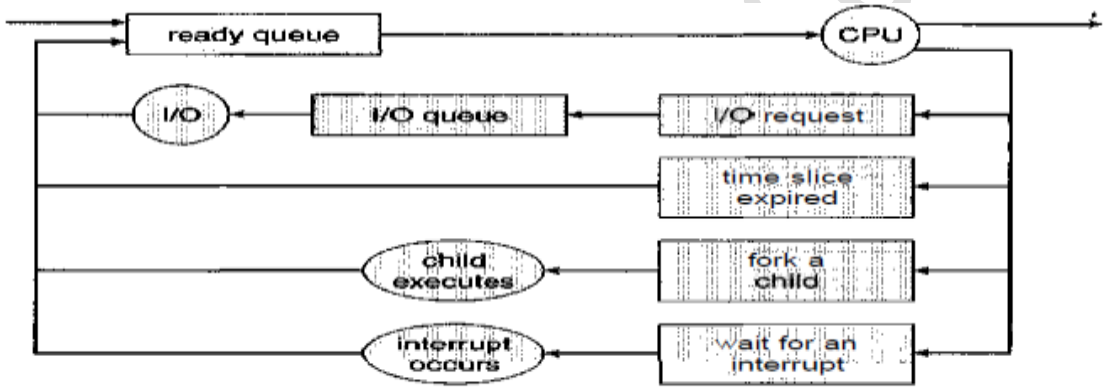


Process control block (PCB).

3. explain the long term, short term and medium term scheduler with the help of queuing diagram

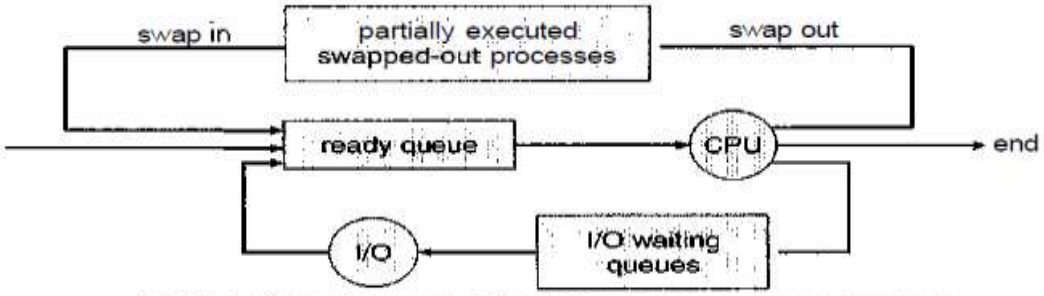
The **long-term scheduler,** or **job scheduler,** selects processes from this pool and loads them into memory for execution. The **short-term scheduler, or** CPU **scheduler,** selects from among the processes that are ready to execute and allocates the CPU to one of them.

The primary distinction between these two schedulers lies in frequency of execution. The short-term scheduler must select a new process for the CPU frequently. A process may execute for only a few milliseconds before waiting for an I/O request. Often, the short-term scheduler executes at least once every 100 milliseconds. Because of the short time between executions, the short-term scheduler must be fast.



Queueing-diagram representation of process scheduling.

Some operating systems, such as time-sharing systems, may introduce an additional, intermediate level of scheduling (**medium-term scheduler**). The key idea behind a medium-term scheduler is that sometimes it can be advantageous to remove processes from memory (and from active contention for the CPU) and thus reduce the degree of multiprogramming.



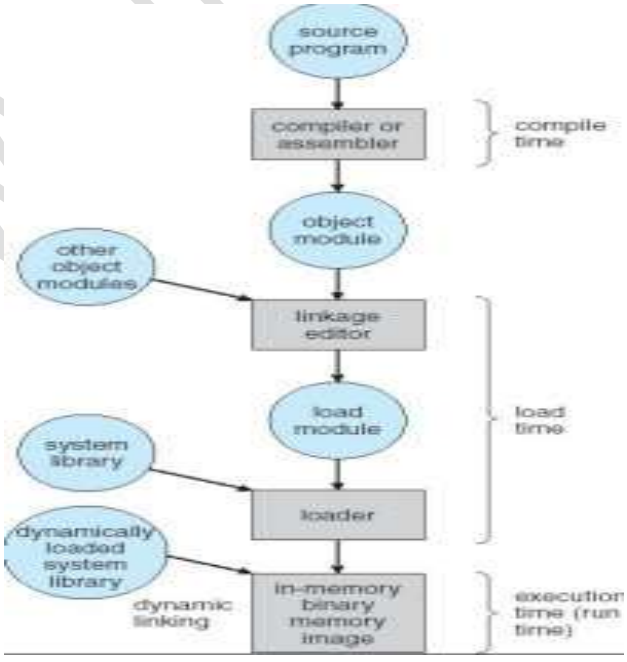
Addition of medium-term scheduling to the queueing diagram.

4. classify address bindings

- The process of associating program instructions and data to physical memory addresses is called address binding
- 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**. The compiler translates symbolic addresses to re-locatable addresses. If it is not known at compile time where the process will reside in memory, then the compiler must generate re-locatable code. If it is known at compile time where the process will reside in memory, then absolute **code** can be generated.
- **Load time**. The loader translates the re-locatable address generated by the compiler to absolute addresses.
- **Execution time**. Binding delayed until run time if the process can be moved during its execution from one memory segment to another. Execution time binding requires hardware support for address maps (e.g., *base* and *limit registers*). Most general-purpose OSs uses this method (Dynamic).



5. differentiate internal and external fragmentation

As processes are loaded and removed from memory, the free memory space is broken into little pieces. **External fragmentation** exists when there is enough total memory space to satisfy a request, but the available spaces are not contiguous; storage is fragmented into a large number of small holes. This fragmentation problem can be severe.

Memory fragmentation can be internal as well as external. Consider a multiple-partition allocation scheme with a hole of 18,464 bytes. Suppose that the next process requests 18,462 bytes. If we allocate exactly the requested block, we are left with a hole of 2 bytes. The overhead to keep track of this hole will be substantially larger than the hole itself. The general approach to avoiding this problem is to break the physical memory into fixed-sized blocks and allocate memory in units based on block size. With this approach, the memory allocated to a process may be slightly larger than the requested memory. The difference between these two numbers is **internal fragmentation**

6. Write any 3 services provided by the kernel I/O subsystem

The kernel's I/O subsystem provides numerous services. Among these is I/O scheduling, buffering, caching, spooling, device reservation, and error handling.

- **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.

- **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. One reason is to cope with a speed mismatch between the producer and consumer of a data stream

- **Caching:** A cache is a region of fast memory that holds copies of data. Access to the cached copy is more efficient than access to the original. For instance, the

instructions of the currently running process are stored on disk, cached in physical memory, and copied again in the CPU's secondary and primary caches

7. Describe the different file operations

- **Creating a file.** Two steps are necessary to create a file. First, space in the file system must be found for the file. We discuss how to allocate space for the file in Chapter 11. Second, an entry for the new file must be made in the directory.

- **Writing a file.** To write a file, we make a system call specifying both the name of the file and the information to be written to the file. Given the name of the file, the system searches the directory to find the file's location. The system must keep a *write* pointer to the location in the file where the next write is to take place. 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 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. Because a process is usually either reading from or writing to a file, the current operation location can be kept as a per-process current file- position pointer**.** Both the read and write operations use this same pointer, saving space and reducing system complexity.

- **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 *seeks.*

- **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 tile be reset to length zero and its file space released.

PART – C

(Answer *one* full question from each unit. Each question carries 15 marks)

UNIT – I

III. Explain any5 operating system components 15

Process Management

A *process* is a program in execution. A process needs certain resources, including CPU time, memory, files, and I/O devices, to accomplish its task.

- The operating system is responsible for the following activities in connection with process management.
    - Process creation and deletion.
    - Process suspension and resumption.
    - Provision of mechanisms for:
        - Process synchronization
        - Process communication

Main-Memory Management

Memory is a large array of words or bytes, each with its own address. It is a repository of quickly accessible data shared by the CPU and I/O devices.

- Main memory is a volatile storage device. It loses its contents in the case of system failure.
- The operating system is responsible for the following activities in connections with memory management:
    - Keep track of which parts of memory are currently being used and by whom.
    - Decide which processes to load when memory space becomes available.
    - Allocate and de-allocate memory space as needed.

File Management

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

- The operating system is responsible for the following activities in connections with file management:
  – File creation and deletion.
  – Directory creation and deletion.
  – Support of primitives for manipulating files and directories.
  – Mapping files onto secondary storage.
  – File backup on stable (nonvolatile) storage media.

I/O System Management

- The I/O system consists of:
  – A buffer-caching system
  – A general device-driver interface
  – Drivers for specific hardware devices

Secondary-Storage Management

Since main memory (*primary storage*) is volatile and too small to accommodate all data and programs permanently, the computer system must provide *secondary storage* to back up main memory.

- Most modern computer systems use disks as the principle on-line storage medium, for both programs and data.
- The operating system is responsible for the following activities in connection with disk management:
  – Free space management
  – Storage allocation
  – Disk scheduling

OR

IV.

a) Compare compiler and interpreter 3

The compiler produces an architecture-neutral byte-code output (.class) file that will run on any implementation of the JVM. A compiler may produce assembly code, which is consumed by an assemble

An interpreter is a computer program that directly executes, i.e. performs, instructions written in a programming or scripting language, without previously compiling them into a machine language program.

b) Differentiate symmetric and asymmetric multiprocessing 3

The multiple-processor systems in use today are of two types. Some systems use **asymmetric multiprocessing,** in which each processor is assigned a specific task. A master processor controls the system; the other processors either look to the master for instruction or have predefined tasks.

The most common systems use **symmetric multiprocessing (SMP),** in which each processor performs all tasks within the operating system. SMP means that all processors are peers; no master-slave relationship exists between processors.

c) Write notes on

i. Batch processing system
ii. Multiprogramming system
iii. Time sharing system 9

Batch processing system: In a batch system, more processes are submitted than can be executed immediately. These processes are spooled to a mass-storage device (typically a disk), where they are kept for later execution.

Multiprogramming system: Multiprogramming increases CPU utilization by organizing jobs (code and data) so that the CPU always has one to execute In a multi-programmed system, the operating system simply switches to, and executes, another job. When *that* job needs to wait, the CPU is switched to *another* job, and so on. Eventually, the first job finishes waiting and gets the CPU back. As long as at least one job needs to execute, the CPU is never idle

Time sharing system: In time-sharing systems, the CPU executes multiple jobs by switching among them, but the switches occur so frequently that the users can interact with each program while it is running. Time sharing requires an interactive (or hands-on) computer system, which provides direct communication between the user and the system

UNIT – II

V.

a) Explain any three process scheduling algorithm with an example 9

1. Shortest-Job-First (SJF) Scheduling

- In fact it is shortest next CPU burst
- Assume CPU burst length for each process in ready queue are known
- Two schemes:
  - *Non-preemptive* – once CPU assigned, process not preempted until its CPU burst completes
  - *Can be preemptive* – if a new process with CPU burst less than remaining time of current, preempt ***Shortest-Remaining-Time-First (SRTF)***
- SJF is optimal – gives minimum average waiting time for a given set of processes
- Example of Non-Preemptive SJF

Process	Burst Time
P_1	16
P_2	3
P_3	4

SJF (non-preemptive) The Gantt Chart for the schedule is:

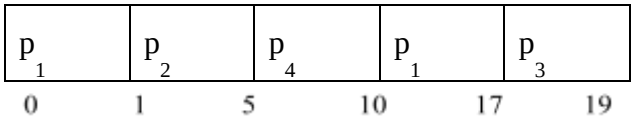
Here, the waiting time for P_1 is 7ms, P_2 is 0ms, P_3 is 3ms.

Average waiting time = (7 + 0 + 3)/3 =10/3=3.33 ms.

- Example of Preemptive SJF

Process	Arrival time	Burst Time
P_1	0	8
P_2	1	4
P_3	2	9
P_4	3	5

SJF (preemptive) The Gantt Chart for the schedule is:

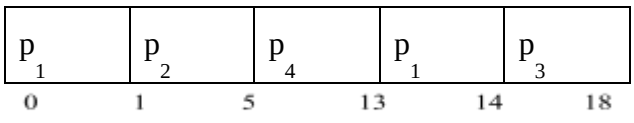


Average waiting time = (10-1) +(1-1) +(17-2)+(5-3)/4 = 26/4 =6.5ms.

2. Priority Scheduling

- Priority number (integer) associated with process
- SJF and SRTF are special cases of general priority scheduling
- Larger the burst time lower is the priority
- CPU allocated to process with highest priority
- Can be preemptive or non-preemptive
- Preemptive priority scheduling will preempt the CPU if a high priority job arrives to ready queue
- Problem: Starvation / indefinite blocking → low priority processes may never execute
- Solution: Aging →increasing gradually the priority of processes which are waiting in the system for CPU for a long time
- Ex: If priority is 127(low) decrement by 1 for every 15 minutes of wait – takes 32 hours to get the priority 0.
- Example of priority scheduling

Process	Burst Time (ms)	priority
P_1	8	3
P_2	2	1
P_3	1	3
P_4	3	2
P_5	4	4



Average waiting time = wait times of $(p_1+p_2+p_3+p_4+p_5)/5$

= (5+0+13+2+14)/5= 34/5

= 6.8 ms

3. Round robin scheduling

- Designed for time-sharing systems
- Jobs get the CPU for a fixed time (quantum time or time slice)
- Similar to FCFS, but with preemption
    - CPU interrupted at regular intervals
- Needs hardware timer
- Ready queue treated as a circular buffer
- Process may use less than a full time slice
- They terminate and scheduling take place
- If process is incomplete at the end of time slice, they join end of ready queue
- With n processes and quantum = q, each process waits for at most (n-1)*q

b) Illustrate resource allocation graph 6

Deadlocks can be described more precisely in terms of a directed graph called a system resource-allocation graph. This graph consists of a set of vertices V and a set of edges E. The set of vertices V is partitioned into two different types of nodes: P -{Pi, Pi,,.., P,,\, the set consisting of all the active processes in the system, and R = {R[, R2, •••/ Rm}, the set consisting of all resource types in the system.

The resource-allocation graph shown in Figure depicts the following situation.

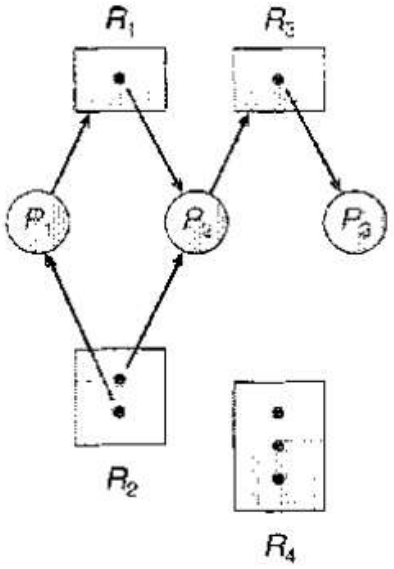
The sets P, *R,* and £:

- *P={PhP2/P?,}*
- *R*= {/?!, *RZ,R3,* R;}
- £ = {p, _>*Ru P2* _> R3/ R, _>*p2f R2* _> P2/ *R2* _> p.,, *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*



Resource allocation graph

Process states:

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

OR

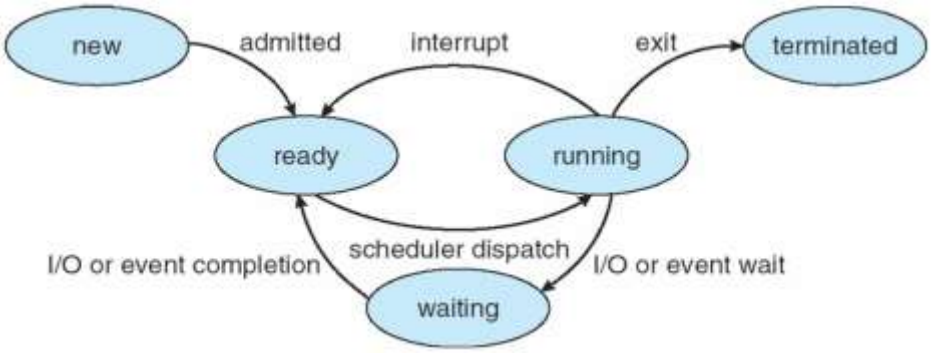
VI.

a) Outline the criteria for evaluating scheduling algorithms 6

- *CPU utilization* – To what percentage CPU is utilized
  - 40% - lightly loaded and 90% - heavily loaded
- *Throughput* – No. of processes that complete their execution per unit time (Degree of multiprogramming)
  - For long processes it could be 1 in an hour and for short processes it could be 10 per sec

- *Turnaround time* –Time interval between the time of submission and completion (Execution time)
    - Includes also waiting times for CPU as well as I/O devices
- *Waiting time* – sum of all the time waiting in *Ready* queue
    - Does not take into account wait time for I/O and I/O operation time
- *Response time* – amount of time it takes from the time of submission of a job until the first response is produced
    - In time shared systems small turn-around time may not be enough
    - Response time should be small. It is the time taken to start responding. Does not include time to output that response

b) Draw the process state diagram 3



c) Write three requirements of any solution to the critical section problem 6

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

1. **Mutual exclusion.** If process P; is executing in its critical section, then no other processes can be executing in their critical sections.
2. **Progress.** If no process is executing in its critical section and some processes wish to enter their critical sections, then only those processes that are not executing in their remainder sections can participate in the decision on which will enter its critical section next, and this selection cannot be postponed indefinitely.

3. **Bounded waiting.** There exists a bound, or limit, 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.

UNIT – III

VII.

a) With neat diagram explain segmentation 6

Consider how you think of a program when you are writing it. You think of it as a main program with a set of methods, procedures, or functions. It may also include various data structures: objects, arrays, stacks, variables, and so on. Each of these modules or data elements is referred to by name. You talk about "the stack," "the math library," "the main program," without caring what addresses in memory these elements occupy You are not concerned with whether the stack is stored before or after the Sqrt() function. Each of these segments is of variable length; the length is intrinsically defined by the purpose of the segment in the program. Elements within a segment are identified by their offset from the beginning of the segment: the first statement of the program, the seventh stack frame entry in the stack, the fifth instruction of the Sqrt (), and so on

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 tithe programmers.)

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 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

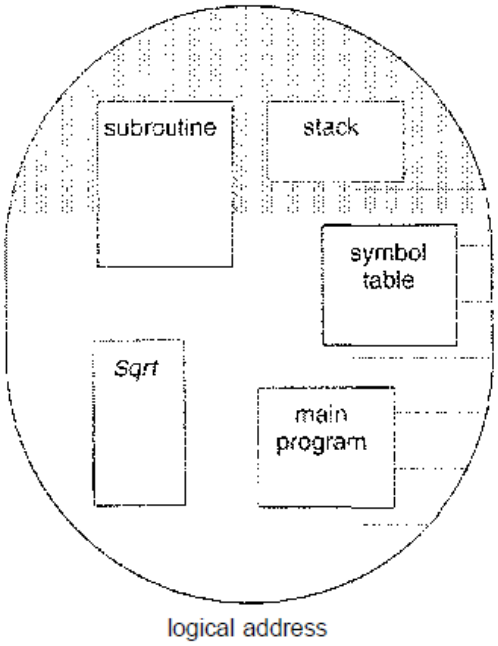


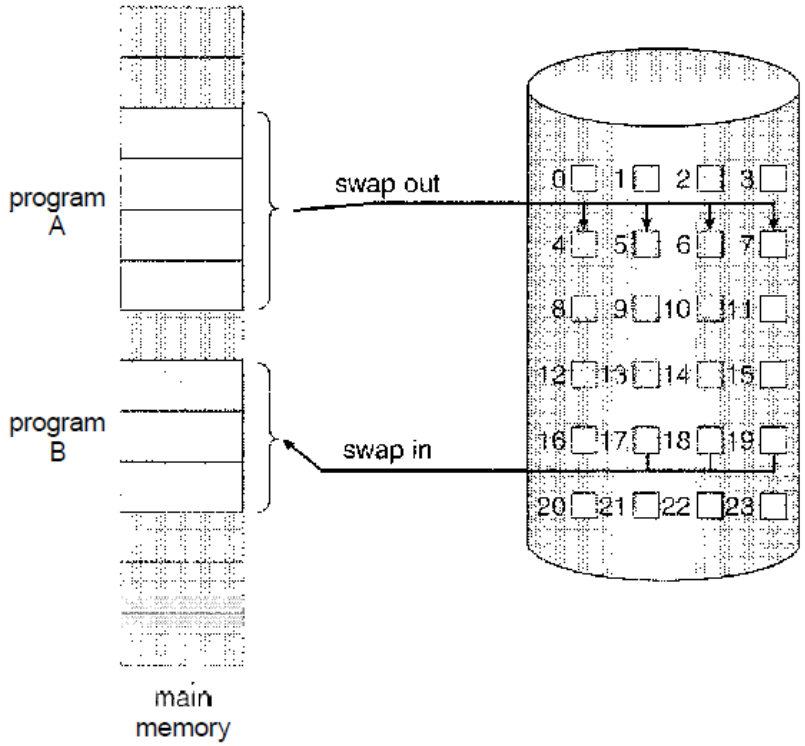
Figure 8.18 User's view of a program.

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.

b) Illustrate demand paging 9

Consider a program that starts with a list of available options from which the user is to select. 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.

A demand-paging system is similar to a paging system with swapping where processes reside in secondary memory (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.** Alay swapper never swaps a page into memory unless that page will be needed. Since we are now viewing a process as a sequence of pages, rather than as one large contiguous address space, use of the term *swapper* is technically incorrect. A swapper manipulates entire processes, whereas a **pager** is concerned with the individual pages of a process. We thus use *pager,* rather than *swapper,* in connection with demand paging.



Transfer of a paged memory to contiguous disk space.

OR

VIII.

a) Describe any three page replacement algorithms 9

In doing so, we use the reference string

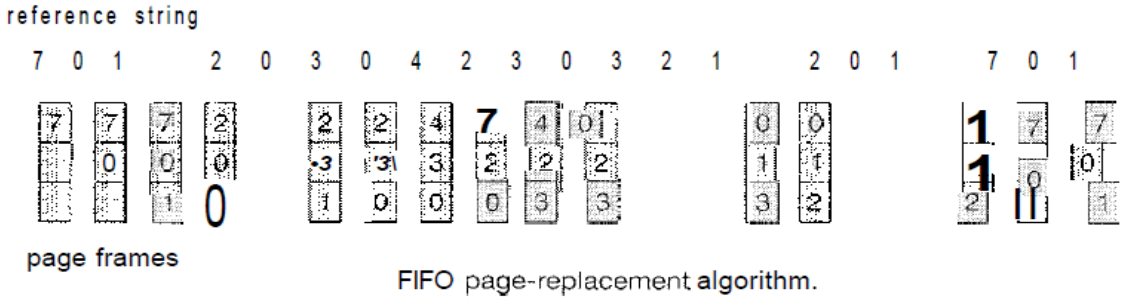
7, 0,1, 2, 0, 3, 0, 4, 2, 3, 0, 3, 2,1, 2, 0, 1, 7, 0,1

For a memory with three frames,

FIFO Page Replacement

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

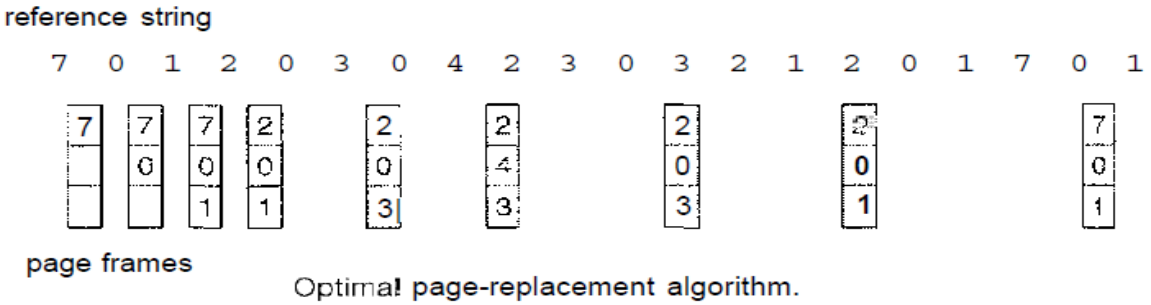
The FIFO page-replacement algorithm is easy to understand and program. However, its performance is not always good. On the one hand, the page replaced may be an initialization module that was used a long time ago and is no longer needed. On the other hand, it could contain a heavily used variable that was initialized early and is in constant use.



FIFO page-replacement algorithm.

Optimal Page Replacement

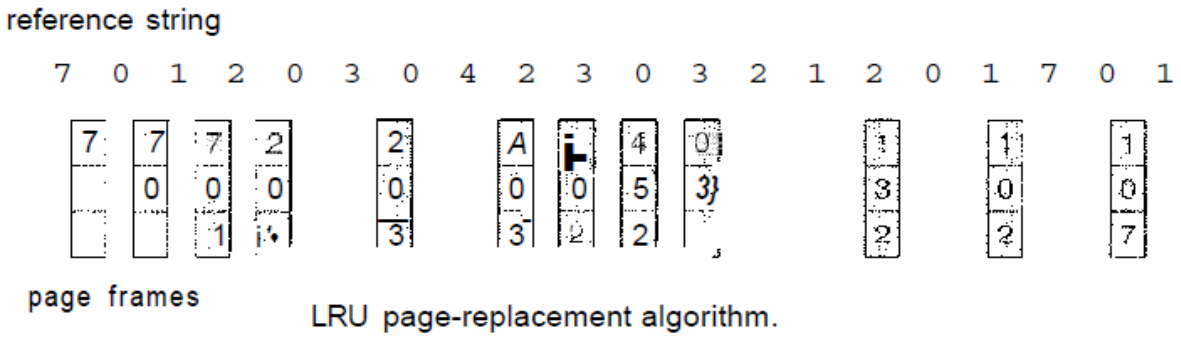
The first three references cause faults that fill the three empty frames. The reference to page2 replaces page 7, because 7 will not be used until reference 18, whereas page0 will be used at 5, and page 1 at 14. The reference to page 3 replaces page1, as page 1 will be the last of the three pages in memory to be referenced again. With only nine page faults, optimal replacement is much better than affix algorithm, which resulted in fifteen faults. (If we ignore the first three, which all algorithms must suffer, then optimal replacement is twice as good affix replacement.) In fact, no replacement algorithm can process this reference string in three frames with fewer than nine faults.



Optimal page-replacement algorithm.

LRU Page Replacement

The result of applying LRU replacement to our example reference string is shown in Figure. The LRU algorithm produces 12 faults. Notice that the first 5 faults are the same as those for optimal replacement. When the reference to page 4 occurs, however, LRU replacement sees that, of the three frames in memory, page 2 was used least recently. Thus, the LRU algorithm replaces page2, not knowing that page 2 is about to be used. When it then faults for page2, the LRU algorithm replaces page 3, since it is now the least recently used of the three pages in memory. Despite these problems, LRU replacement with 12faults is much better than FIFO replacement with 15.

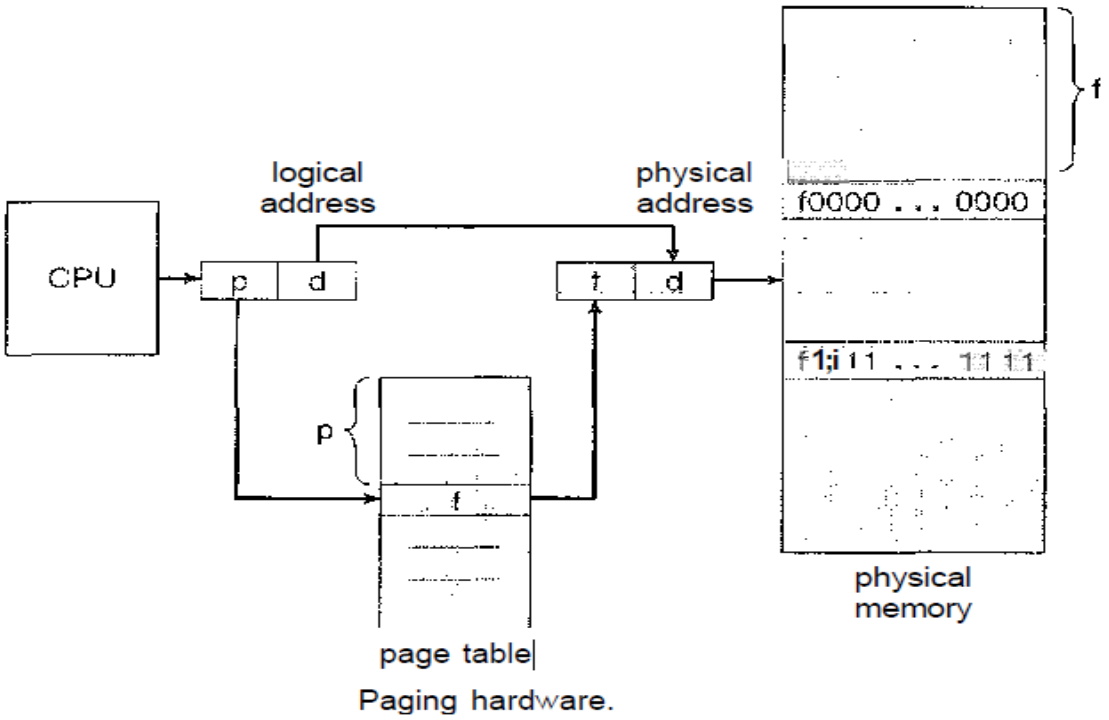


LRU page-replacement algorithm.

b) Explain paging scheme with paging hardware diagram 6

Paging is a memory-management scheme that permits the physical address space of a process to be noncontiguous. Paging avoids the considerable problem of fitting memory chunks of varying sizes onto the backing store; most memory-management schemes used before the introduction of paging suffered from this problem. The problem arises because, when some code fragments or data residing in main memory need to be swapped out, space must be found on the backing store. The backing store also has the fragmentation problems discussed in connection with main memory; except that access is much slower, so compaction is

impossible. Because of its advantages over earlier methods, paging in its various forms is commonly used in most operating systems.



Paging hardware.

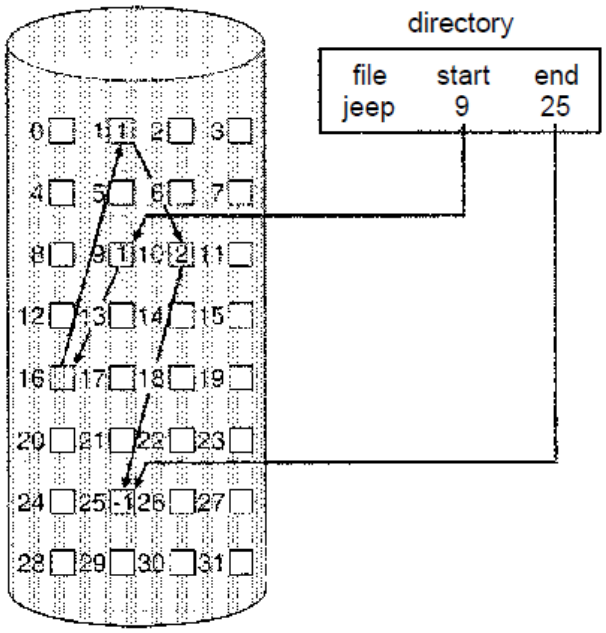
UNIT – IV

IX.

a) Explain linked and indexed file allocation methods 8

Linked file allocation

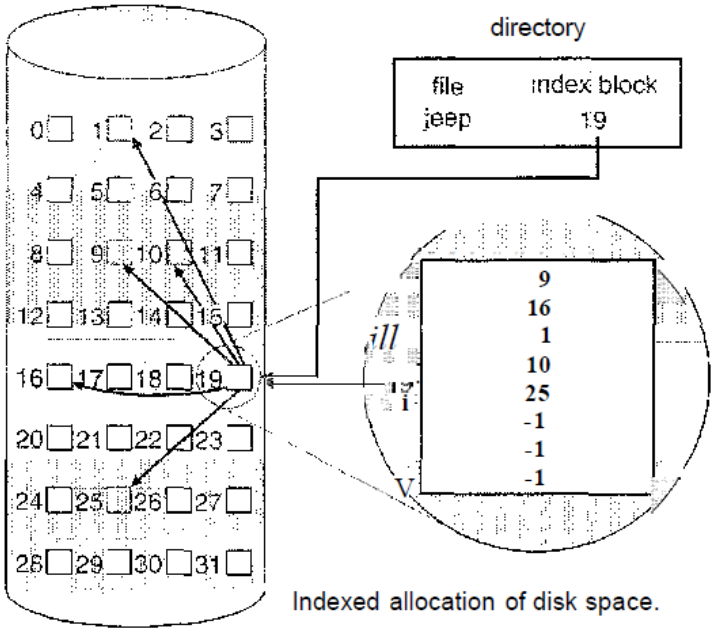
With linked allocation, each file is a linked list of disk blocks; the disk blocks may be scattered anywhere on the disk. The directory contains a pointer to the firsthand last blocks of the file. For example, a file of five blocks might start at block9 and continue at block 16, then block 1, then block 10, and finally block 25. 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 disk address (the pointer) requires 4 bytes, then the user sees blocks of 508bytes



Linked allocation of disk space.

Indexed file allocation

Linked allocation solves the external-fragmentation and size-declaration problems of contiguous allocation. However, in the absence of a FAT, linked allocation cannot support efficient direct access, since the pointers to the blocks are scattered with the blocks themselves all over the disk and must be retrieved in order. **Indexed allocation** solves this problem by bringing all the pointers together into one location: the **index block.** Each file has its own index block, which is an array of disk-block addresses. The i^{th} entry in the index block points to the /'" block of the file. The directory contains the address of the index block (Figure 11.8). To find and read the i^{t} block, we use the pointer in $teet^{h}$ index-block entry.



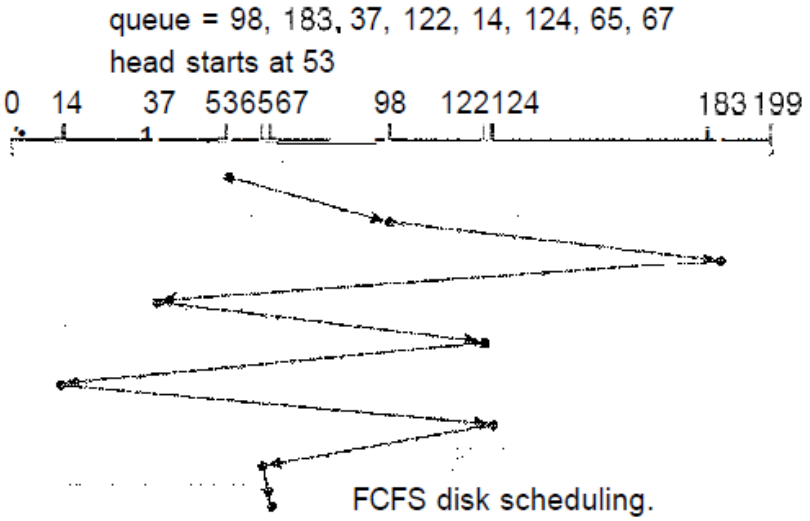
Indexed allocation of disk space.

b) Illustrate FCFS and SSTF disc scheduling algorithms 7

FCFS disc Scheduling

The simplest form of disk scheduling is, of course, the first-come, first-served (FCFS) algorithm. This algorithm is intrinsically fair, but it generally does not provide the fastest service. Consider, for example, a disk queue with requests for I/O to blocks on cylinders in that order. If the disk head is initially at cylinder 53, it will first move from53 to 98, then to 183, 37, 122, 14, 124/65, and finally to 67, for a total head movement of 640 cylinders. This schedule is diagrammed in Figure. The wild swing from 122 to 14 and then back to 124 illustrates the problem with this schedule. If the requests for cylinders 37 and 14 could be serviced together, before or after the requests at 122 and 124, the total head movement could be decreased substantially, and performance could be thereby improved.

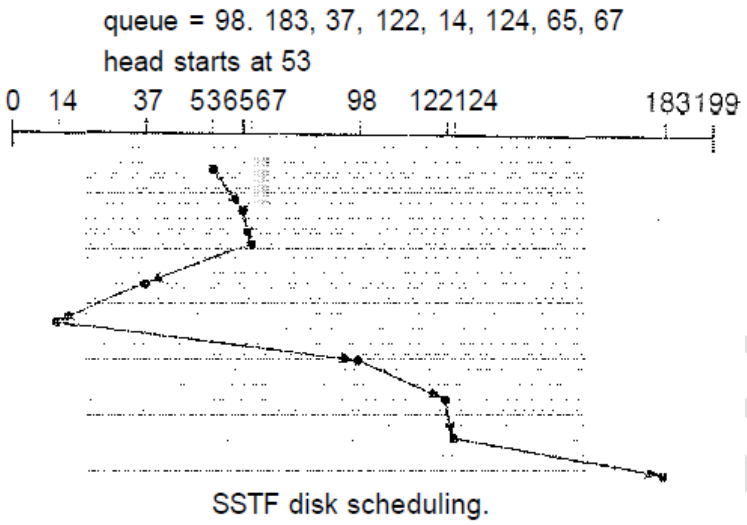
98, 183, 37,122, 14, 124, 65, 67,



FCFS disk scheduling.

SSTF disc Scheduling

It seems reasonable to service all the requests close to the current head position before moving the head far away to service other requests. This assumption is the basis for the **shortest-seek-time-first (SSTF) algorithm.** The SSTF algorithm selects the request with the minimum seek time from the current head position. Since seek time increases with the number of cylinders traversed by the head, SSTF chooses the pending request closest to the current head position. For our example request queue, the closest request to the initial head position (53) is at cylinder 65. Once we are at cylinder 65, the next closest request is at cylinder 67. From there, the request at cylinder 37 is closer than the one at 98, so 37 is served next. Continuing, we service the request at cylinder 14, then 98,122, 124, and finally 183. This scheduling method results in a total head movement of only 236 cylinders—little more than one-third of the distance needed for FCFS scheduling of this request queue. This algorithm gives a substantial improvement in performance.



SSTF disk scheduling.

OR

X.

a) Explain the way in which the OS performs disc management 9

The operating system is responsible for several other aspects of disk management, too. Here we discuss disk initialization, booting from disk, and bad-block recovery.

Disk Formatting

To use a disk to hold files, the operating system still needs to record its own data structures on the disk. It does so in two steps. The first step is to partition the disk into one or more groups of cylinders. The operating system can treat each partition as though it were a separate disk. For instance, one partition can hold a copy of the operating system's executable code, while another holds user files. After partitioning, the second step is logical formatting (or creation of a file system). In this step, the operating system stores the initial file-system data structures onto the disk. These data structures may include maps of free and allocated space (a FAT or modes) and an initial empty directory.

Boot Block

Since ROM is read only, it cannot be infected by a computer virus. The problem isthat changing this bootstrap code requires changing the ROM, hardware chips. For this reason, most systems store a tiny bootstrap loader program in the boot ROM whose only job is to bring in a full bootstrap program from disk. The full bootstrap program can be changed easily: A new version is simply written onto the disk. The

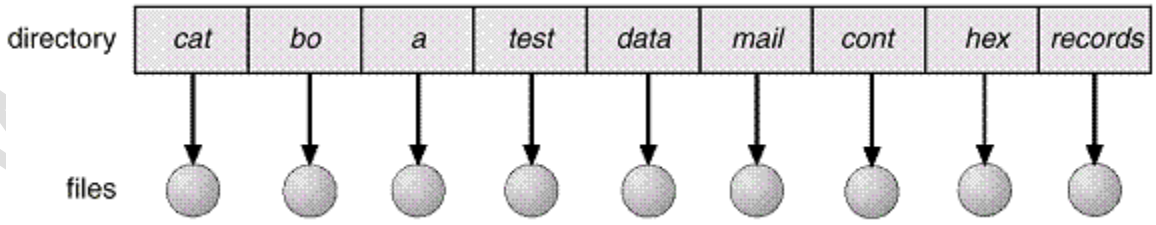
full bootstrap program is stored in "the boot blocks" at a fixed location on the disk. A disk that has a boot partition is called a boot disk or system disk. The code in the boot ROM instructs the disk controller to read the boot blocks into memory (no device drivers are loaded at this point) and then starts executing that code. The full bootstrap program is more sophisticated than the bootstrap loader in the boot ROM; it is able to load the entire operating system from a non-fixed location on disk and to start the operating system running. Even so, the full bootstrap code may be small.

Bad Blocks

Because disks have moving parts and small tolerances (recall that the disk head flies just above the disk surface), they are prone to failure. Sometimes the failure is complete; in this case, the disk needs to be replaced and its contents restored from backup media to the new disk. More frequently, one or more sectors become defective. Most disks even come from the factory with bad blocks. Depending on the disk and controller in use, these blocks are handled in a variety of ways

b) Explain three directory structures 6

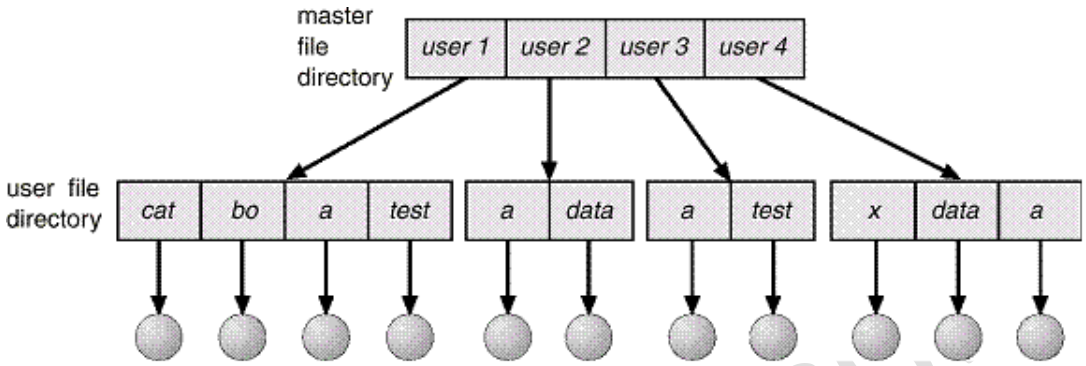
Single Level Directory: in this there is a single directory which contains all the Files into. But this is not the best way because all the Files are Stored into the Single Folder So this is very difficult for a user to search a File. There is Only one Directory Which contains the other files So that Many times this is not used because this will Contains Huge Amount of Files and this will Makes difficult for the users to Locate or Find a File



Two Level Directories: in this a Directory also contains a Another Directory and all the Files are Organized into the Sub Directory.

In the Two Level a directory also Contains Sub Directory and the Files. Or we can say that a Single Directory will be the Container of Many other files and the Directories So

that When a user wants to Access any Directory then he has To Travel all the other Directories and Files from that Directory



Tree structured Directory: 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.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 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),
  - as a subdirectory (1).
- Path names can be of two types: **absolute** and **relative**
  - An absolute path name begins at the root and follows a path down to the specified file, giving the directory names on the path.
  - A relative path name defines a path from the current directory.
- With a tree-structured directory system, users can be allowed to access, in addition to their files, the files of other users.
  - For example, user B can access a file of user A by specifying its path names.
  - User B can specify either an absolute or a relative path name.
  - Alternatively, user B can change her current directory to be user A's directory and access the file by its file names.