

**B. E.**

**Fifth Semester Examination, May-2009**

## **PRINCIPLES OF OPERATING SYSTEM**

**Note : Attempt any five questions.**

**Q. 1. (a) What are the advantages of multiprocessor system? Describe the difference between symmetric and asymmetric multiprocessing.**

**Ans.** Multiprocessor systems have more than one processor in close communicating, sharing the computer bus, the clock and sometimes memory and peripheral devices. These systems are referred to as tightly coupled systems. Advantages of multiprocessor system are :

**1. Increased Thoughtput :**

By increasing the number of processors we hope to get more work done in a short period of time. For a heavy work, we use multiprocessor system. The speedup ratio for a processor system less than  $n$ .

**2. Increased Reliability :**

If function can be distributed properly among several processors, then the failure of one processor will not halt the system, but rather will only slow it down. This ability to continue providing service proportional to the level of surviving hardware is called graceful degradation.

**3. Resource Sharing :**

Many resources are shared by all the processors available. Many processes make use of a resource by sharing it.

**4. Money Saving :**

Multiprocesses can also save money compared to multiple single systems because the processors can share peripherals, cabinets, power supplies.

There are two types of multiprocessing : Symmetric & asymmetric. The most common multiple-processor systems use the symmetric-multiprocessing model, in which each processor runs an identical copy of the operating system and these copies communicate with one another as needed.

Some systems use asymmetric multiprocessing, in which each process 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. This scheme defines a master-slave relationship. The master processors schedule & allocate work to the slave processors.

An example: if the symmetric multiprocessor system is Encore's version of UNIX for the multimax computer. This computer can be configured to employ dozens of processors, all running a copy of UNIX.

A symmetric multiprocessing is more common in extremely large systems, where one of the most time-consuming activities is simply processing input/output.

**Q. 1. (b) Differentiate between the following :**

- (i) **Buffering and Spooling**
- (ii) **System programs and System calls**
- (iii) **Client-Server systems and Peer-to-Peer systems.**

**Ans. (i) Buffering and spooling :**

Buffering is to make available a memory to store data temporarily, so that the computer is freed up quickly & doesn't have to wait for a slower input/output device. e.g., for printing pages we can first store pages in a buffer memory & then print pages easily from the buffer.

Spooling, in essence, uses the disk as a huge buffer, for reading as far ahead as possible on input devices and for storing output files until the output devices are able to accept them. Spooling is also used for processing data at remote sites. The CPU sends the data via communications paths to a remote printer. The remote processing is done at its own speed, with no CPU intervention. The CPU just needs to be notified when the processing is completed, so that it can spool the next batch of data.

Buffering & spooling both are used to keep both the CPU and the input/output devices working at much higher rates.

**(ii) System programs & system calls :**

System calls provide the interface between a process & the operating system. These calls are generally available as assembly-language instructions, and are usually listed in the manuals used by assembly-language programmers. Some systems may allow system calls to be made directly from a higher-level language program, in which case the calls normally resemble pre-defined function or subroutine calls.

A simple program is system make heavy use of system calls. System calls are needed for every thing done in a program. Some of the system calls for file manipulation are : Create file, delete file, open, close, read, write, reposition, get file attributes, set file attributes.

System programs provide a more convenient environment for program development & execution. Some of them are simply user interfaces to system calls, whereas others are considerably more complex. They can be divided into several categories :

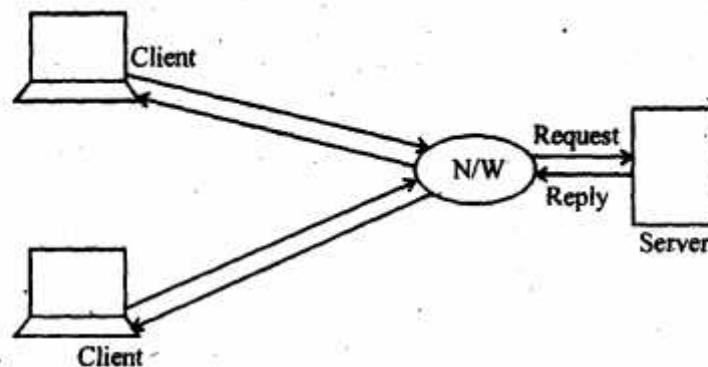
1. File manipulation.
2. Status information.
3. File modification.
4. Programming-language support.
5. Program loading & execution
6. Communications
7. Application programs.

Most important system program for an operating system is the command interpreter, the main function of which is to get and execute the next user-specified command.

A system program may make use of many system calls. System call is the smallest identity of any system program running in the system.

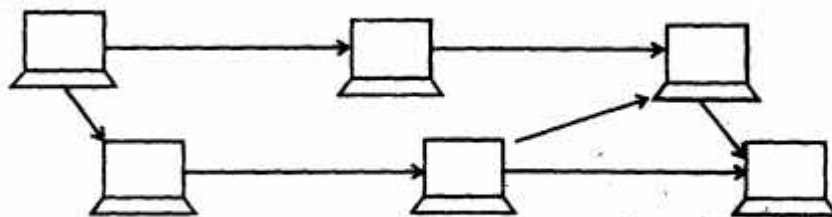
**(iii) Client-server systems & peer-to-peer systems :**

In a client server model the data are stored on powerful computers called servers. The employees have simpler machine called clients. The client sends the request to the server and the server fulfill the request of the client.



Two processes are involved, one on the client machine & one on the server machine. Communication takes the form of the client process sending a message over the network to the server process. The client then wants for a reply message. When the server process gets the request, it performs the work or look up the requested data & sends a reply.

Another type of person-to-person communication often goes by the name of peer-to-peer communication, to distinguish it from the client-server model. In this form, individuals who form a loose group, can communicate with other in the group. Every person can, in principle, communicate with one or more other people, there is no fixed division into clients & servers.



Peer-to-peer communication really hit the big time around 2000 with a service called Napster.

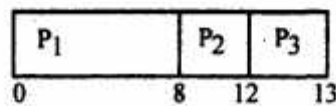
Q. 2. (a) Consider the following processes with arrival time and execution time as indicated :

Process	Arrival Time	Burst Time
P1	0.0	8
P2	0.4	4
P3	1.0	1

Calculate Average Waiting Time and Average Turn around Time using :

- (i) FCFS
- (ii) SJF non-preemptive
- (iii) SJF preemptive.

Ans. (i) Using FCFS :



Gantt chart for FCFS.

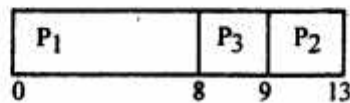
Average waiting time  $= \frac{0 + (8 - 0) + (12 - 8)}{3}$

$$= \frac{7.6 + 11}{3} = \frac{18.6}{3} = 6.2$$

Average turn around time  $= \frac{8 + (12 - 0) + (13 - 8)}{3}$

$$= \frac{8 + 11.6 + 12}{3} = \frac{31.6}{3} = 10.53$$

(ii) SJF non-preemptive :



Gantt chart for SJF non-preemptive



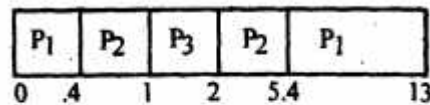
$$\text{Average waiting time} = \frac{0 + (9 - 4) + (8 - 1)}{3}$$

$$= \frac{8.6 + 7}{3} = \frac{15.6}{3} = 5.2$$

$$\text{Average turn around time} = \frac{8 + (13 - 4) + (9 - 1)}{3}$$

$$= \frac{8 + 12.6 + 8}{3} = \frac{28.6}{3} = 9.53$$

(iii) SJF Preemptive :



Gantt chart for SJF preemptive

$$\text{Average waiting time} = \frac{(0 + (5.4 - 4)) + (.4) + 0}{3} = \frac{5 + 1.4}{3}$$

$$= \frac{5 + 1.4}{3} = \frac{6.4}{3} = 2.133$$

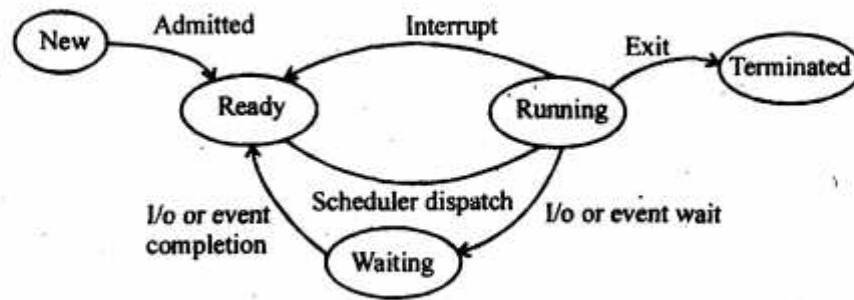
$$\text{Average turnaround time} = \frac{(0 + 13) + (5.4 - 4) + (2 - 1)}{3} = \frac{13 + 5 + 1}{3} = \frac{19}{3} = 6.33$$

**Q. 2. (6) What is process? Describe seven state model for process.**

**Ans.** A process is a program in execution. The execution of a process must progress in a sequential fashion. That is, at any time, at most one instruction is executed on behalf of the process.

A process is more than the program code. It also includes the current activity, as represented by the value of the program counter and the contents of the processor's registers. A process generally also includes the process stack, containing temporary data and a data section containing global variables.

Diagram of process states is given as :



**Diagram of process state**

As 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 :

**1. New :**

The process is being created.

**2. Running :**

Instructions are being executed.

**3. Waiting :**

The process is waiting for some event to occur (such as an input/output completion or reception of a signal).

**4. Ready :**

The process is waiting to be assigned to a processor.

**5. Terminated :**

The process has finished execution.

**Q. 3. (a) Discuss the similarities and differences between paging and segmentation. What are advantages of combining paging with segmentations?**

**Ans.** A possible solution to the external fragmentation problem is to permit the logical address space of a processes to be non-contiguous, thus allowing a process to be allocated physical memory wherever the latter is available. One way of implementing this solution is through the use of paging avoids the considerable

problem of fitting the varying-sized memory chunks onto the backing store, from which most of the previous memory-management schemes suffered.

In paging physical memory is broken into fixed-sized blocks called frames. Logical memory is also broken into blocks of the same size-called pages. When a process is to be executed, its pages are loaded into any available memory frames from the backing store. The backing store is divided into fixed-sized blocks that use of the same sizes as the memory frames.

When we use a paging schemes, we have no external fragmentation. Any free frame can be allocated to a process that needs it. However, we have some internal fragmentation.

An important aspect of memory management that became unavoidable with paging is the separation of the user's view of memory & 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. The mapping allows differentiation between logical memory & physical memory.

The user prefers to view memory as a collection of variable-sized segments, with no necessary ordering among segments. 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 & length. The address specifies both the segment name and the offset within the segment. The user therefore specifies each address by two quantities a segment name & an offset.

As occur with paging, this mapping requires two memory references per logical address, effectively showing the computer system by a factor of 2. The normal solution is to use a set of associative registers to hold the most recently used segment-table entries.

Segmentation may cause external fragmentation, which is not in paging.

Both paging & segmentation have their advantages & disadvantages. In fact, of the two most popular microprocessors now being used, the motorola 68000 line is designed based on a flat address space, where is the Intel 80 × 86 family in based on segmentation. Both are merging memory models toward a mixture of paging, segmentation. It is possible to combine these two schemes to improve on each. This combination is best illustrated by two different architectures—the innovative but not widely used MULTICS system & the Intel 386.

**Q. 3. (b) Consider the following pages reference string :**

1, 2, 3, 1, 4, 5, 6, 2, 1, 3, 2, 7, 6, 3, 4, 1, 2, 6

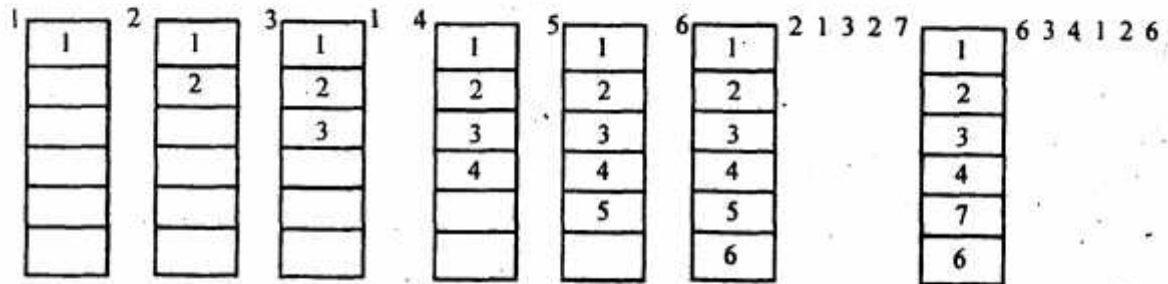
**How many page faults would occur for the following replacement algorithms assuming six frames? All frames are initially empty :**

(i) LRU

(ii) FIFO

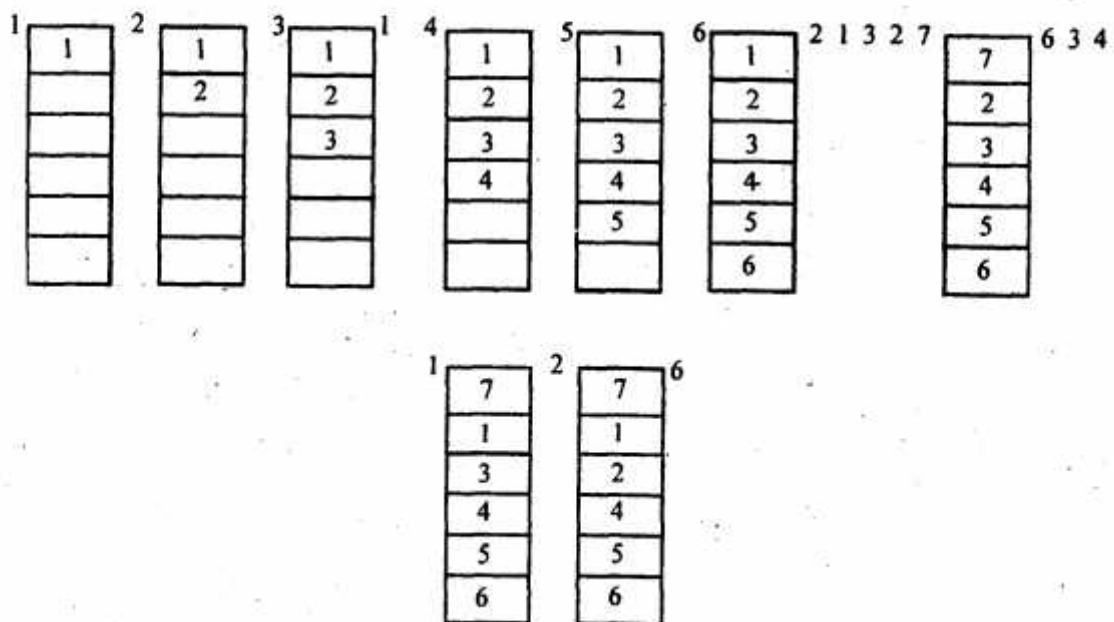
(iii) Optimal

Ans. (i) LRU :



There are total seven page faults.

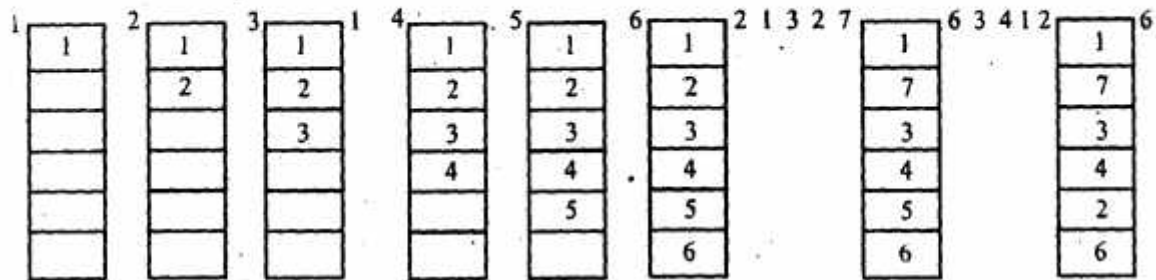
(ii) FIFO :





There are nine page faults.

(iii) Optimal :



There are eight page faults.

**Q. 4. (a) Discuss various file allocation and access methods. Compare their advantages and disadvantages.**

**Ans. Different allocation methods for the file system are :**

#### **1. Contiguous Allocation :**

The contiguous allocation method requires each file to occupy a set of contiguous blocks on the disk. Disk addresses define a linear ordering on the disk.

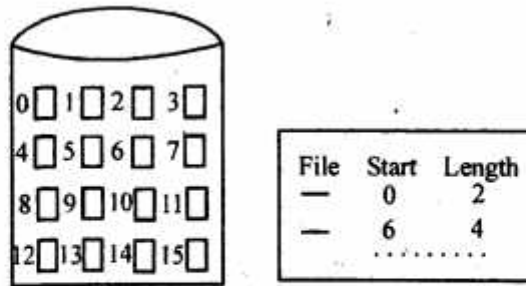
Contiguous allocation of a file is defined by the disk address and length of the first block. If the file is  $n$  blocks long, and starts at location  $b$ , then it occupies  $b, b+1, \dots, b+n-1$ . The directory entry for each file indicates the addresses of the starting block & the length of the area allocated for this file.

#### **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, of a file that starts of block  $b$ , we can immediately access block  $b+i$ . Thus, both sequential and direct access can be supported by contiguous allocation.

One difficulty with contiguous allocation is finding space for a new file.

The contiguous disk-space-allocation problem can be seen to be a particular application of the general dynamic storage-allocation problem, which is how to satisfy a request of size  $n$  from a list of free holes. This will create a problem of external fragmentation in contiguous allocation.



### Contiguous allocation of disk space

There are other problems with contiguous allocation. A major problem is determining how much space is needed for a file. When the file is created, the total amount of space it will need must be found & allocated.

To avoid several of these drawbacks, some operating systems use a modified contiguous allocation scheme, in which a contiguous chunk of space is allocated initially, then, when that amount is not large enough, another chunk of contiguous space, an extent, is added to the initial allocation. The location of a file's block is then recorded as a location and a block count, plus a link to the first block of the next extent.

### 2. Linked Allocation :

Linked allocation solves all problems of contiguous allocation. With linked allocation, each file is a linked list of disk blocks; the disk blocks may be scattered anywhere on the disk. The directory contains a pointer to the first & last blocks of the file. For e.g., a file of five blocks might start at block 9, continue at block 16, then block 1, block 10 & finally block 25. Each block contains a pointer to next block.

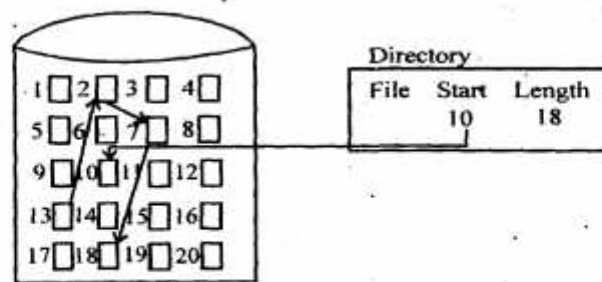
To create a new file, we simply create a new entry in the directory with linked allocation, each directory entry has a pointer to the first disk block of the file.

There is no external fragmentation with linked allocation and any free block on the free-space list can be used to satisfy a request.

Linked allocation also has disadvantages. The major problem is that it can be used effectively for only sequential-file access. To find the  $i^{\text{th}}$  block of a file, we must start at the beginning of that file and follow the pointers until we get the  $i^{\text{th}}$  block. Each access to a pointer requires a disk read, and sometimes a disk seek. Consequently, it is sufficient to support a direct-access capability for linked allocation files.

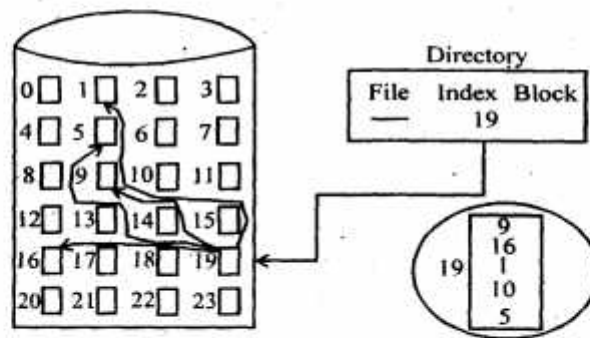
Another disadvantage is the space required for the pointers. Each file requires some more space than it is of one another problem is reliability.

An important variation on the linked allocation method is the use of a file-allocation table (FAT).



### 3. Indexed Allocation :

Linked allocation solves the external fragmentation and size-declaration problem of contiguous allocation. However in the absence of a FAT, linked allocation can't support with the blocks themselves all over the disk & need to 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 disks-block addresses. The  $i^{th}$  entry in the index block points to the  $i^{th}$  block of the file. The directory contains the address of the index block. To read the  $i^{th}$  block, we use the pointer in the  $i^{th}$  index block entry to find and read the desired block. Indexed allocation supports direct access, without suffering from external fragmentations, because any free block on the disk satisfy a request for more space.



### Indexed allocation of a file

Indexed allocation does suffer from wasted space. The pointer overhead of the index block is generally greater than the pointer overhead of linked allocation.



**Q. 4. (b) Give advantages of Distributed system over Centralized system.**

**Ans.** In distributed system, the various processors communicate with one another through various communication uses such as high-speed buses or telephone lines. These system are also called loosely coupled system. Advantages of distributed system as :

**1. Resource Sharing :**

If a number of different sites are connected to one another, then a user at one site may be able to use the resources available at another. In general, resource sharing in a distributed system provides mechanisms for sharing files at remote sites-processing information in a distributed database, printing files at remote sites, using remote specialized h/w devices, performing other operations.

**2. Computation Speed up :**

If a particular computation can be partitioned into a number of sub-computations that can run concurrently, then a distributed system may allow us to distribute the computation among the various sites. This will do the computation in a much less time.

**3. Reliability :**

If one site fails in a distributed system, the remaining sites can potentially continue operating. If the system is composed of a number of large autonomous installations, the failure of one of them should not affect the rest. The system can continue with its operation even if some of its sites have failed.

**4. Communication :**

There are many instances in which programs need to exchange data with one another on the system. When many sites are connected to one another by a communication network, the processes at diff. sites have the opportunity to exchange information user may initiate file transfers or communicate with one another via electric mail. A user can send mail to another users at the same site or at a different site.

**Q. 5. (a) Discuss the following :**

- (i) **Critical section problem**
- (ii) **Race condition**
- (iii) **Priority inversion problem**
- (iv) **Monitors.**

**Ans. (i) Critical section problem :**

Consider a system consisting of  $n$  processes  $(P_0, P_1, P_2, \dots, P_{n-1})$ . Each process has a segment of code, called a critical section, in which the process may be changing common. Variables, updating a table, writing a file & so on. The important feature of the system is that, when one process is executing in its critical section, no other process is to be allowed to execute in its critical section. Thus, the execution of critical sections by the process is mutually exclusive in time. The critical section may be followed by an exit section. The remaining code is the remainder section.



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

**1. Mutual Exclusion :**

If process  $P_i$  is executing in its critical section, no other process can be executing in their critical sections.

**2. Progress :**

If no process is executing in its critical sections there exist some process that wish to enter their critical section & there exist some process that wish to enter their critical sections, then only those processes that are not executing in their remainder section can participate in the decision of which will enter the critical section.

**(ii) Race Condition :**

There exist a bound on the number of times that other processes are allowed to enter their critical section after a process has made a request to enter its critical section & before that request is granted.

A situation, where several processes access and manipulate the same data concurrently and the outcome of the execution depends on the particular order in which the access takes place, is called a race condition. To guard against the race condition, we need to ensure that only one process at a time can be manipulating the variable to make such a guarantee, we require some form of synchronization of the processes. Such situations occur frequently in operating systems as different parts of the system manipulate resources and are want the changes not to interfere with one another.

**(iii) Priority Inversion Problem :**

For the case where we use preemption, we have to preempt one process by another. When a higher priority process comes, we have to replace it by a lower priority process. But what happens if the higher-priority process needs to read or modify kernel data that are currently being accessed by another, lower-priority one to finish. This situation is known as priority inversion problem. In fact, there could be a chain of processes, all accessing resources that the high-priority process needs. This problem can be solved via the priority-inheritance protocol in which all these processes (the processes that are accessing resources that the high-priority process needs) inherit the priority until they are done with the resources. When they are finished, their priority reverts to its natural value.

**(iv) Monitors :**

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

The representation of a monitor type cannot be used directly by the various processes. Thus, a procedure defined within a monitor can access only those variables declared locally within the monitor and the formal parameters. Similarly, the local variables of a monitor can be accessed by only the local procedures.

The monitor construct ensures that only one process at a time can be active within the monitor. Consequently, the programmer doesn't need to code this synchronization constraint explicitly. These mechanisms are

provided by the condition construct.

**Q. 5. (b) What is binary semaphore? Implement wait() and signal() without busy wait for binary semaphores. With the help of binary semaphore implement the counting semaphore.**

**Ans.** The semaphore is a binary semaphore with an integer value that can range only b/w 0 and 1. A binary semaphore can be simpler to implement than a counting semaphore, depending on the underlying hardware architecture. We will now show how a counting semaphore is implemented using binary semaphores.

Let S be a writing semaphore. To implement it in terms of binary semaphore we need the following :

data structures :

var S1 : binary\_semaphore;

S2 : binary\_semaphore;

S3 : binary\_semaphore;

C : integer;

Initially S1 = S3 = 1, S2 = 0, & the value of integer C is set to the initial value of the counting semaphore S.

The wait operation on the counting semaphores can be implemented as follows :

wait (S3);

wait (S1);

C := C - 1;

if C < 0

then begin

signal (S1);

wait (S2);

end

else signal (S1);

signal (S3);

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

wait (S1);

C := C + 1;

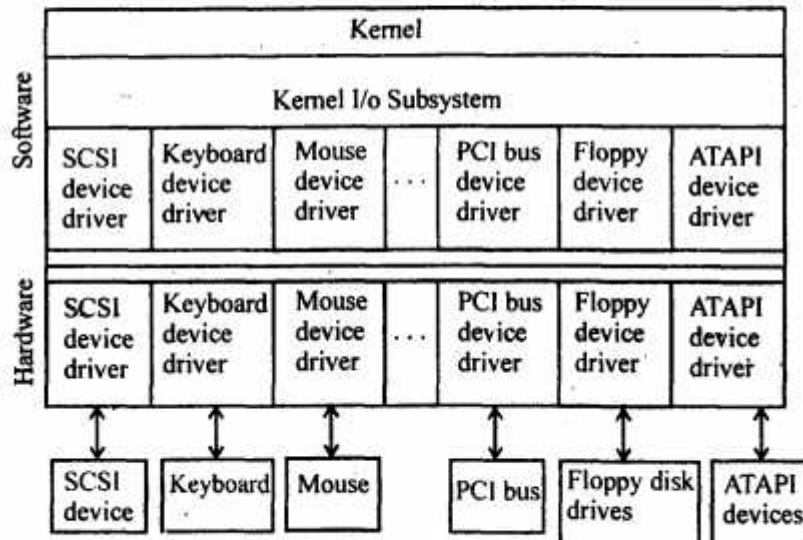
if C ≤ 0 then signal (S2);

signal (S1);

The S3 semaphore has no effect on signal (S), it merely serializes the wait (S) operations.

**Q. 6. Explain input/output device organization? Describe input/output interrupts and input/output buffering.**

**Ans.** The operating system organizes the input/output devices into categories to form a general application input/output interface. Fig. illustrates how the input/output related portions of the kernel are structured in software layers.



The purpose of the device-driver layer is to hide the differences among device controllers from the input/output subsystem of the kernel, much as the input/output system calls encapsulate the behaviour of devices in a few generic classes that hide hardware differences from applications. Making the input/output sub-system independent of the hardware simplifies the job of the operating system developer. It also benefits the h/w manufacturers. They either design new devices to be compatible with an existing host controllers interface, or they write device drivers to interface the new hardware to popular operating systems.

**Devices may vary in many dimensions :**

**(i) Character-stream or block :**

A character-stream device transfers byte one by one, whereas a block device transfers a block of bytes as a unit.

**(ii) Sequential or random access :**

A sequential device transfers data in a fixed order that is determined by the device, whereas the user of a random-access device can instruct the device to seek to any of the available data storage locations.

**(iii) Synchronous or asynchronous :**

A synchronous device is one that performs data transfers with predicate response times. An asynchronous device exhibits irregular or unpredictable response times.

**(iv) Sharable or dedicated :**

A sharable device can be used concurrently by several processes or threads; a dedicated device cannot.

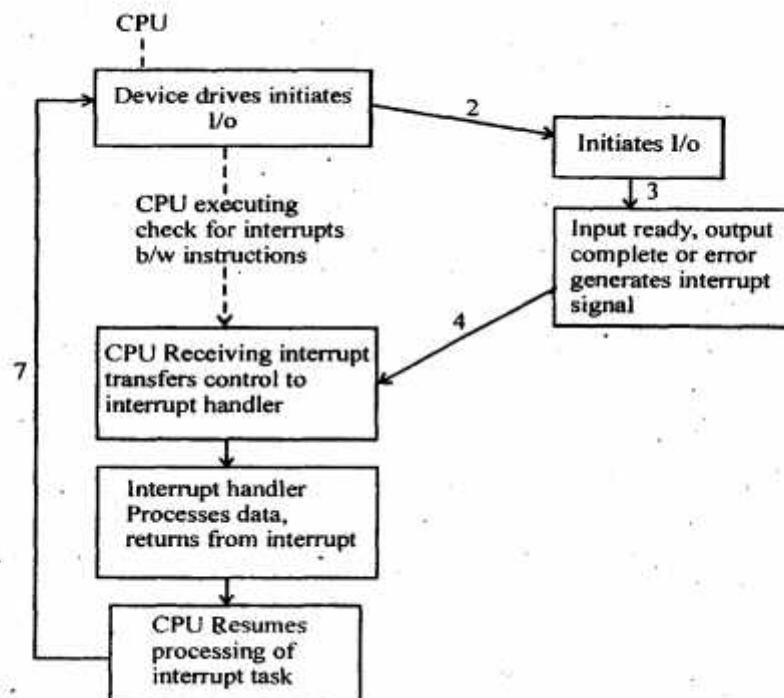
**(v) Speed of Operation :**

Device speeds range from a few bytes per second to a few gigabytes per seconds.

**(vi) Read-write, read only or write only :**

Some devices perform both input & output, but other supports only one data direction.

**Input/output Interrupts :**





The basic concept of interrupt mechanism works as follows. The CPU h/w has a wire called the interrupt request line that the CPU senses after executing every instruction. When the CPU detects that a controller has asserted a signal on the interrupt request line, the CPU saves a small amount of state, such as the current value of the instruction pointer, and jumps to the interrupt handler routine at a fixed address in memory. The interrupt handler determines the cause of the interrupt, perform the necessary processing, and executes a return from interrupt instruction to return the CPU to the execution state prior to the interrupt.

This basic interrupt mechanism enables the CPU to respond to an asynchronous event, such as a device controller becoming ready for service. In a modern operating system, we must have interrupt handling features.

Fig., shows the interrupt driven input/output cycle.

#### **Input/Output 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 & consumer of a data stream. Like for printing pages as the printer is much slower than processor, we can store all pages first & buffer, then print these pages from printer.

A second use of buffering is to adapt between devices that have different data-transfer sizes. Such disparities are especially common in computer networking, where buffers are used widely for fragmentation/reassembly of messages. At the sending side, a large message is fragmented into small n/w packets. The packets are sent over the network, the receiving side places them in a reassembly buffer to form an image of the source data.

A third use of buffering is to support copy buffering semantics, the version of the data written to disk is guaranteed to be the version of the time of the application system call, independent of any subsequent changes in the application buffer.

**Q. 7-(a) What are threads? What are the differences between user level threads and kernel level threads? Under what circumstances is one type better than the others?**

**Ans.** A thread sometimes called a light weight process, it is a basic unit of CPU utilization, and consists of a program counter, a register set, and a stack space. It shares with Peer threads its code section, data section and operating system resources such as open files and signals, collectively known as a task. A traditional or heavy weight process is equal to a task with one thread. A task does nothing if no threads are in it and a thread must be in exactly one task.

Also, some systems implement user-level threads in user-level libraries, rather than via system calls so thread switching doesn't need to call the operating system, and to cause can interrupt to the kernel. User-level threads do have disadvantages. For instance, if the kernel is single-threaded, then any user-level thread executing a system call will cause the entire task to wait until the system call returns.

We can grasp the functionality of threads by comparing multiple-thread control with multiple-process control.

Threads operate, in many respects, in the same manner as processes. Threads can be in one of the several states; ready, blocked, running or terminated. Because all threads can access every address in the task, a thread can read or write over any other thread's stacks.

Consider this system in operation. Any one task may have many user-level threads. These user-level threads may be scheduled and switched among kernel supported lightweight processes without the intervention of the kernel. No context switch is needed for one user-level thread to block & another to start running, so user-level threads are extremely efficient.

These user-level threads are supported by light weight processes. Each LWP is connected to exactly one kernel-level thread, whereas each user-level thread is independent of the kernel.

The kernel threads are scheduled by the kernel's scheduler and execute on the CPU or CPUs in the system. If a kernel thread blocks, the processor is free to run another kernel thread. We conclude for the user-level & kernel level threads that :

1. A kernel thread has only a small data structure and a stack. Switching between kernel threads doesn't require changing memory access information and therefore is relatively fast.
2. An LWP contains a process control block with register data, accounting information and memory information. Switching between LWP's therefore require quite a bit of work and is relatively slow.
3. A user-level thread needs only a stack and a program counter no kernel resources are required. The kernel is not involved in switching these user-level threads; therefore, switching among them is fast. There may be thousands of these user level threads, but all the kernel will ever see is the LWPs in the process that support these user-level threads.

**Q. 7. (b) Differentiate between Interrupt and Trap.**

**Ans.** There are many different types of events that may trigger an interrupt for example, the completion of an input/output operation, division by zero, invalid memory access and a request for some operating-system service. For each such interrupt, a service routine is provided that is responsible for dealing with the interrupt. When the CPU is interrupted, it stops what it is doing & immediately transfers execution to a fixed location. The fixed location usually contains the starting address where the service routine for the interrupt is located. The interrupt service routine executes, on completion, the CPU resumes the interrupted computation.

A higher-priority interrupt will be taken even if a lower-priority interrupt is active, but interrupts at the same or lower levels are masked or selectively disabled, so that lost or unnecessary interrupts are prevented.



Modern operating systems are interrupt driven. If there are no processes to execute, no input/output devices to service and no user to whom to respond, an operating system will sit quietly, waiting for something to happen. Events are almost always signaled by the occurrence of an interrupt, or a trap. A trap (or an execution) is a software generated interrupt caused either by an error (for e.g., division by zero), or by a specific request from a user program that an operating-system service be performed.

When an interrupt (or trap) occurs, the h/w transfers controls to the operating system. First, the operating system preserves the state of the CPU by storing registers & the program counter. Then, it determines which type of interrupt has occurred.

**Q. 8. Write short notes on any four of the following :**

- (i) **Swapping**
- (ii) **Disk Scheduling**
- (iii) **Multiprogramming v/s Multitasking**
- (iv) **Deadlock**
- (v) **Bankers algorithm.**

**Ans. (i) Swapping :**

A process needs to be in memory to be executed. A process, however can be swapped temporarily out of memory to a backing store and then brought back into memory for continued execution. For e.g., assume a multiprogramming environment with a round-robin CPU scheduling algorithm when a quantum expires, the memory manager will start to swap out the process that just finished, and to swapped in another process to the memory space that has been freed. When each process finishes its quantum, it will be swapped with another process.

A variant of this swapping policy is used for priority based scheduling algorithms. If a higher priority process arrives than the CPU swap the lower priority process with the higher one. Swapping requires a backing store. The backing store is commonly a fast disk. It must be large enough to accommodate copies of all memory images for all users and must provide direct access to the memory images.

**(ii) Disk Scheduling :**

One of the responsibilities of the O.S. is to use the h/w efficiently. For the disk drives, this means having a fast access time & disk bandwidth. The access time has two major components. The seek time is the time for the disk arm to move the heads to the cylinder containing the desired sector. The rotational latency is the additional time waiting for the disk to rotate the desired sector to the disk head. The disk to rotate the desired

sector to the disk head. The disk bandwidth is the total number of bytes transferred, divided by the total time between the first request for service & the completion of the last transfer. We can improve both the access time & the bandwidth by scheduling the servicing of disk input/output requests in a good order.

The various algorithm for disk scheduling are :

1. FCFS scheduling (First Come First Serve)
2. SSTF Scheduling (Shortest-Seek Time First)
3. SCAN scheduling
4. C-SCAN scheduling
5. Look scheduling.

**(iii) Multiprogramming v/s Multitasking :**

Multiprogramming operating system allows two or more users to run programs at the same time. The objective of a multiprogramming operating system is to increase the system utilization efficiency. Some of the most popular multiprogramming operating system are :

\* UNIX, VMS, Window NT etc.

Multitasking allows more than one program to run concurrently. The ability to execute more than one task at the same time, a task being a program. The terms multitasking & multiprocessing are often used interchangeably.

In multitasking only one CPU is involved, but it switches from one program to another so quickly that it gives the appearance of executing all the programs at the same time. Time sharing is done between different programs executing concurrently.

**(iv) Deadlock :**

In a multiprogramming environment, several processes may compete for a finite number of resources. A process requests resources; if the resources are not available at that time, the process enters a wait state. It may happen that waiting processes will never again change state, because the resources they have requested are held by other waiting processes. This situation is called a deadlock. A deadlock situation can arise if the following four conditions hold simultaneously in a system.



1. **Mutual exclusion** : At least one resource must be held to a non-sharable mode.

2. **Hold & wait** : There exist a process that is holding one resource & waiting to acquire another.

3. **Nopreemption** : Resources cannot be preempted.

4. **Circular wait** : There must exist a set  $\{P_0, P_1, \dots, P_n\}$ , such that  $P_0$  is waiting for resource held by  $P_1$ ,  $P_1$  is waiting for  $P_2$  &  $P_n$  is waiting for a resource held by  $P_0$ , thus forms a deadlock cycle.

**(v) Bankers algorithm :**

The bankers algorithm is used for deadlock avoidance when a new process enters the system, it must declare the maximum number of instances of each resource type that it may need. This number may not exceed the total number of resources in the system. When a user requests a set of resources, the system must determine whether the allocation of these resources will leave the system in safe state. If it is the resources are allocated, otherwise, the process must wait until some other process releases enough resources.

We need the following data structures :

\* **Available** : A vector of length  $m$  indicates the number of available resources of each type. If  $\text{available}[j] = k$ , these are  $k$  instances of resource type  $P_j$  available.

\* **Max** : An  $n \times m$  matrix defines the maximum demand of each process. If  $\text{max}[i, j] = k$ , then  $P_i$  may request at most  $k$  instances of  $R_j$ .

\* **Allocation** : An  $n \times m$  matrix defines the number of resources of each type currently allocated to each process.

\* **Need** : An  $n \times m$  matrix indicates the remaining resources, need of each process.

Using the data structures we will find whether the following state is safe or not and for making a new request what resources must be allocated first  $P_i$  a new safe state.