

B. E.

Fifth Semester Examination, Dec-2008

PRINCIPLES OF OPERATING SYSTEM

Note : Attempt any five questions.

Q. 1. (a) What are the advantages of multiprogramming system? Describe the difference between multiprogramming and multitasking.

Ans. Advantages of Multiprogramming :

Multiprogramming is the rapid switching of the CPU between multiple processes in memory. It is done only when the currently running process requests I/O, or terminates. It was commonly used to keep the CPU busy while one or more processes are doing I/O. It is now mostly superseded by multitasking, in which processes also lose the CPU when their time quantum expires. Multiprogramming makes efficient use of the CPU by overlapping the demands for the CPU and its I/O devices from various users. It attempts to increase CPU utilization, by always having something for the CPU to execute.

The prime reason for multiprogramming is to give the CPU something to do while waiting for I/O to complete. If there is no I/O, the CPU is fully occupied doing I/O, so there is nothing to be gained (at least in terms of CPU utilization) by multiprogramming. No matter how much I/O a program does, the CPU will be 100% busy. This of course assumes the major delay is the wait while data is copied. A CPU could do other work if the I/O were slow for other reasons (arriving on a serial line, for instance).

I/O operations are exceedingly slow (compared to instruction execution)

A program containing even a very small number of I/O ops, will spend most of its time waiting for them

Hence: poor CPU usage when only one program is present in memory.

Multitasking :

Multitasking is a method by which multiple tasks, also known as processes, share common processing resources such as a CPU. In the case of a computer with a single CPU, only one task is said to be running at any point in time, meaning that the CPU is actively executing instructions for that task. Multitasking solves the problem by scheduling which task may be the one running at any given time, and when another waiting task gets a turn. The act of reassigning a CPU from one task to another one is called a context switch. When context switches occur frequently enough the illusion of parallelism is achieved. Even on computers with more than one CPU (called multiprocessor machines), multitasking allows many more tasks to be run than there are CPUs. Operating systems may adopt one of many different scheduling strategies, which generally fall into the following categories :

- In multiprogramming systems, the running task keeps running until it performs an operation that requires waiting for an external event (e.g. reading from a tape) or until the computer's scheduler forcibly swaps the running task out of the CPU. Multiprogramming systems are designed to maximize CPU usage.

- In time-sharing systems, the running task is required to relinquish the CPU, either voluntarily or by an external event such as a hardware interrupt. Time sharing systems are designed to allow several programs to

execute apparently simultaneously.

In real-time systems, some waiting tasks are guaranteed to be given the CPU when an external event occurs. Real time systems are designed to control mechanical devices such as industrial robots, which require timely processing.

Advantages :

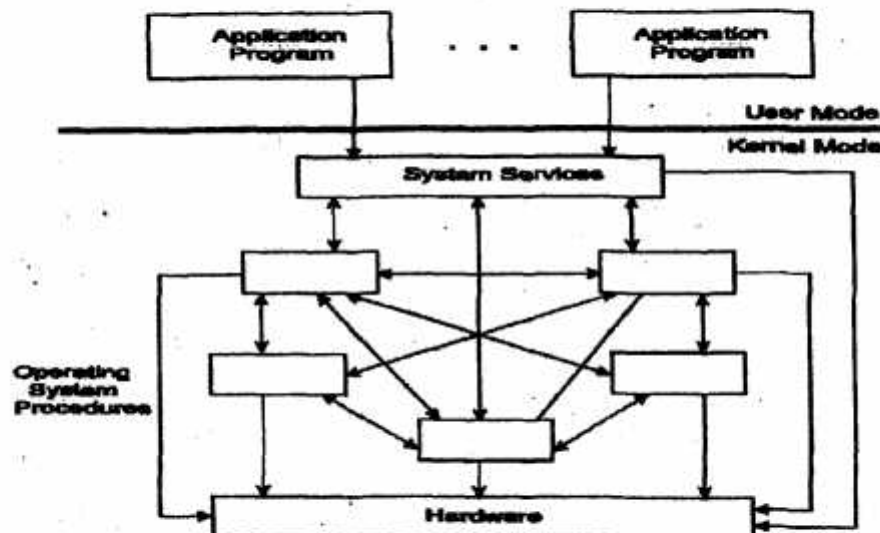
Multiprogramming makes efficient use of the CPU by overlapping the demands for the CPU and its I/O devices from various users. It attempts to increase CPU utilization by always having something for the CPU to execute.

Q. 1. (b) Discuss the following types of Operating system structures :

- (i) Monolithic systems
- (ii) Layered systems
- (iii) Virtual machine
- (iv) Client-server model.

Ans.

(i) Monolithic Systems :



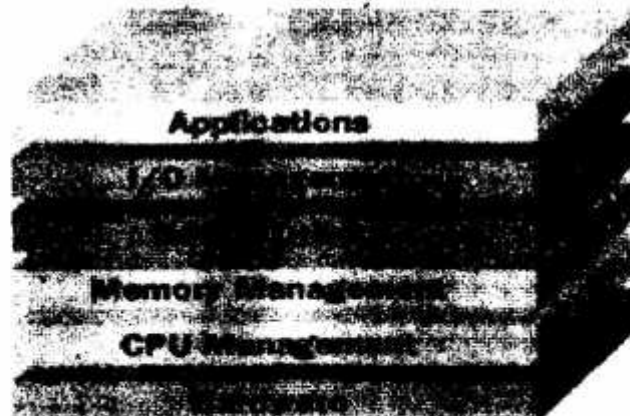
- Application programs that invokes the requested system services.
- A set of system services that carry out the operating system procedures/calls.
- A set of utility procedures that help the system services.

- MS-DOS - written to provide the most functionality in the least space :
- not divided into modules (monolithic).
- Although MS-DOS has some structure, its interfaces and levels of functionality are not well separated.

(ii) Layered approach :

The operating system is divided into a number of layers (levels), each built on top of lower layers.

The bottom layer (layer 0) is the hardware; the highest (layer N) is the user interface. With modularity, layers are selected such that each uses functions (operations) and services of only lower-level layers.



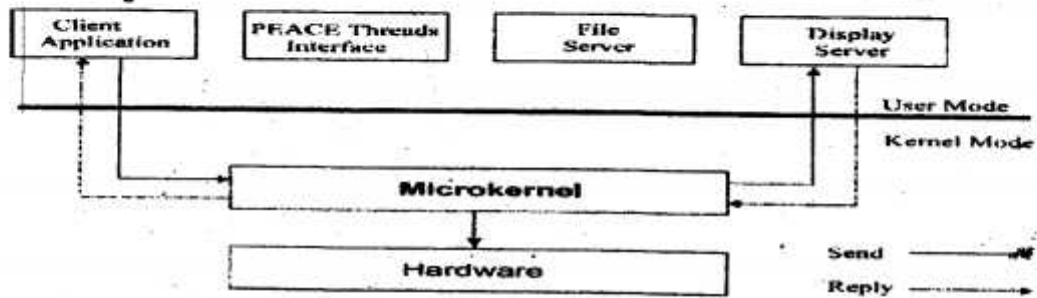
(iii) Virtual Machine :

A Virtual Machine (VM) takes the layered and microkernel 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 host creates the illusion that a process has its own processor and (virtual memory).
- Each guest provided with a (virtual) copy of underlying computer
- 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.

(iv) Client Server Model :

The advent of new concepts in operating system design, microkernel, is aimed at migrating traditional services of an operating system out of the monolithic kernel into the user-level process. The idea is to divide the operating system into several processes, each of which implements a single set of services. For example, I/O servers, memory server, process server, threads interface system. Each server runs in user mode, provides services to the requested client. The client, which can be either another operating system component or application program, requests a service by sending a message to the server. An OS kernel (or microkernel) running in kernel mode delivers the message to the appropriate server; the server performs the operation and microkernel delivers the result to the client in another message, as shown in figure.



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

| Process | Arrival Time | Burst Time |
|---------|--------------|------------|
| P1 | 0.0 | 10 |
| P2 | 3.0 | 2 |
| P3 | 4.0 | 1 |
| P4 | 5.0 | 4 |

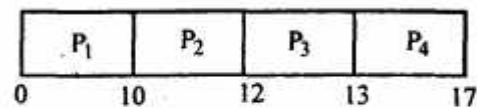
Calculate Average Waiting Time and Average Turn around Time using :

- FCFS
- SJF preemptive
- Round Robin (Time quantum = 1)

Ans.

| Process | Arrival Time | Burst Time |
|---------|--------------|------------|
| P1 | 0.0 | 10 |
| P2 | 3.0 | 2 |
| P3 | 4.0 | 1 |
| P4 | 5.0 | 4 |

FCFS:



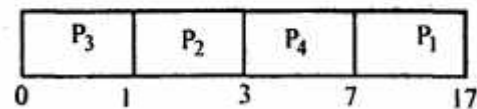
The waiting time for P₁ = 0

$$P_2 = 10, P_3 = 12, P_4 = 13$$

∴ Total average time

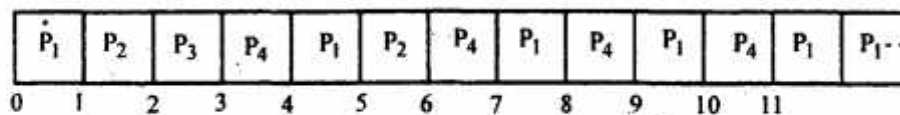
$$= \frac{0+10+12+13}{4} = \frac{35}{4}$$

SJF:



The Total average time for these tasks will be.

(iii) Round Robin : Time quantum = 1



The total average time will be.

$$\frac{0+4+2+7+11}{4} = \frac{24}{4} = 6 \text{ sec.}$$

Q. 2. (b) What is process? Describe seven state modes for process.

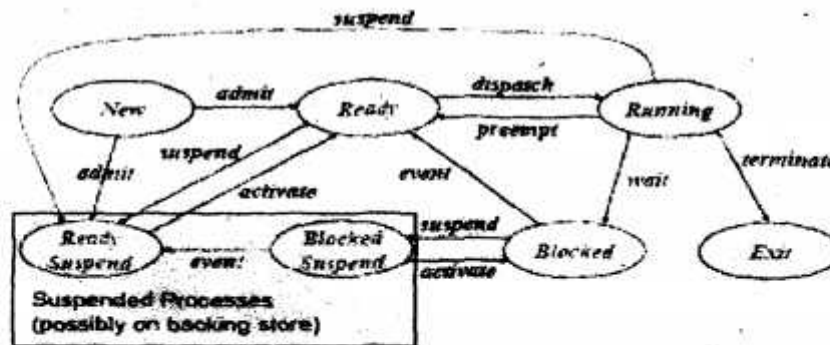
Ans. Process:

A program in execution; process execution must progress in sequential fashion.

A process includes :

- Program counter.
- Stack
- Data section.

Seven state model of a process :



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

Ans. Paging : Paging (non-contiguous allocation) :

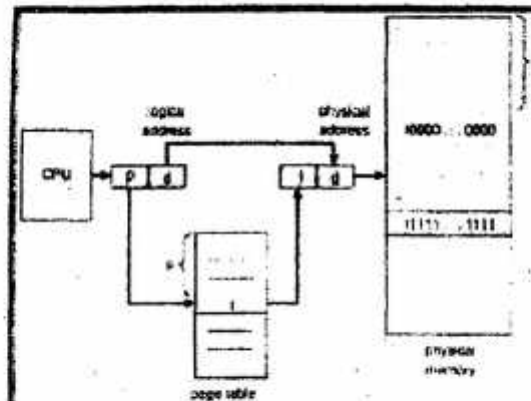
- Physical address space of a process can be non-contiguous; process is allocated physical memory whenever the latter is available.
- Divide physical memory into fixed-sized blocks called frames (typically between 512 bytes and 8192 bytes).
- Divide logical memory into blocks of same size called pages.
- To run a program of size n pages, need to find n free frames and load program.
- Internal fragmentation.

Address translation Scheme :

A logical address is divided into :

- Page number (p)—used as an index into a page table which contains base address of each page in physical memory.

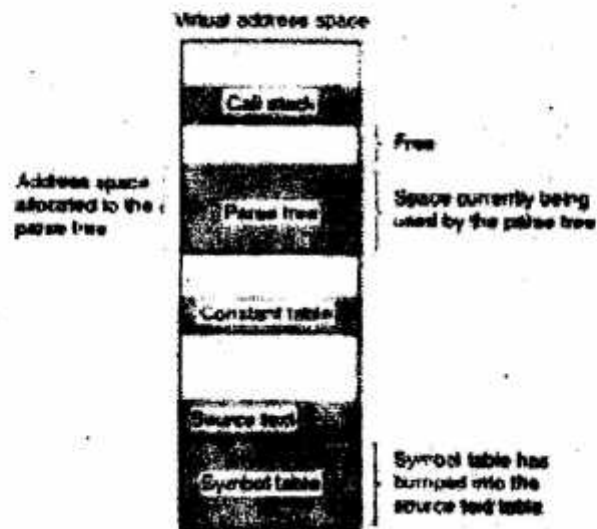
- Page offset (d)—combined with base address to define the physical memory address that is sent to the memory.



Segmentation :

One-dimensional address space with growing pieces.

- At compile time, one table may bump into another.
- Segmentation :
- Generate segmented logical address at compile time.
- Segmented logical address is translated into physical address at execution time q by software or hardware?



Paging vs. Segmentation :

| Consideration | Paging | Segmentation |
|---|--|--|
| Need the programmer be aware that this technique is being used? | No | No |
| How many total address space spaces are there? | 1 | Many |
| Can producers and data be distinguished and separately protected? | No | Yes |
| Can tables whose size fluctuates be accommodated easily? | No | Yes |
| Is sharing of procedures between users facilitated? | No | Yes |
| Why was this technique invented? | To get a large linear address space without having to buy more physical memory | To allow programs and data to be broken up into logically independent address spaces and to aid sharing and protection |

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

1, 2, 3, 4, 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 :

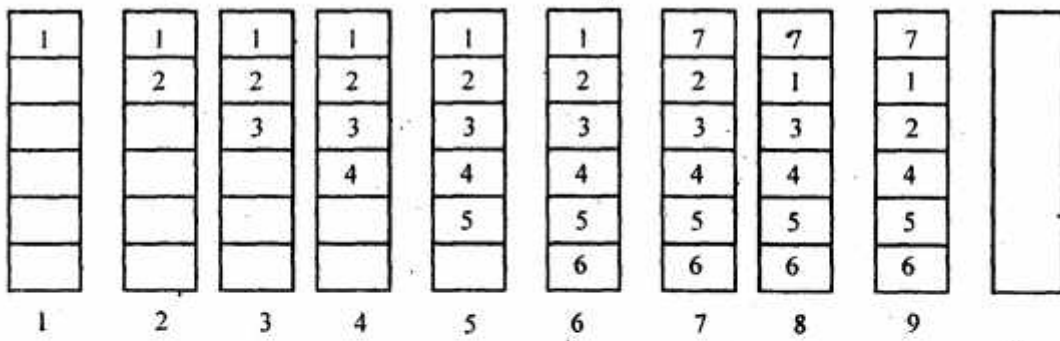
- (i) **LRU**
- (ii) **FIFO**
- (iii) **Optimal**

Ans. The page reference string is

1, 2, 3, 4, 5, 6, 2, 1, 3, 2, 7, 6, 3, 4, 1, 2, 6 using 6 frames.

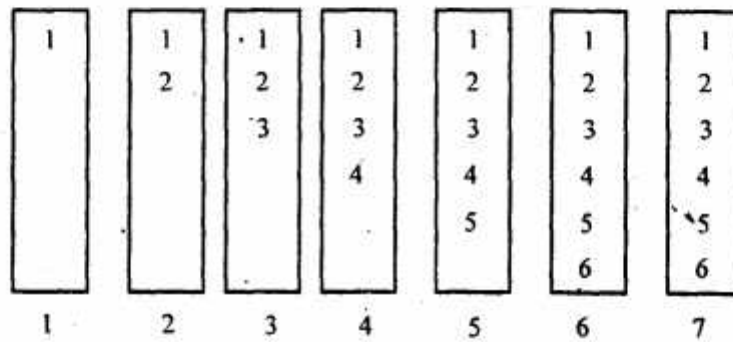
(i) LRU :

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



The total number of page faults will be 7.

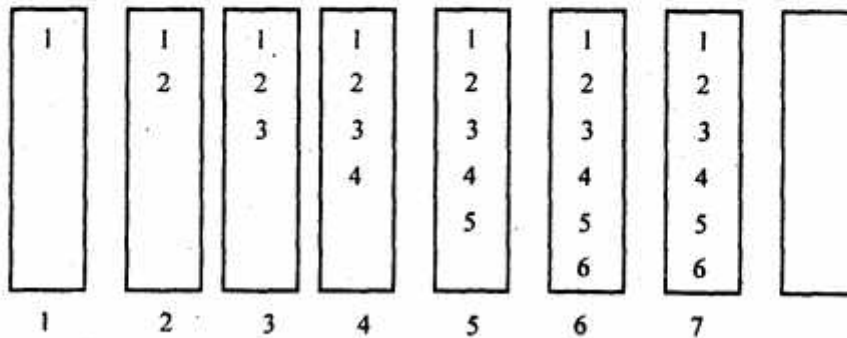
(ii) FIFO:



The total number of page faults will be 9.

(iii) Optimal Algo :

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



The total number of page faults will be 7.

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

Ans. Various file allocation methods :

One main problem in file management is how to allocate space for files so that disk space is utilized effectively and files can be accessed quickly. Three major methods of allocating disk space are contiguous, linked, and indexed. Each method has its advantages and disadvantages. Accordingly, some systems support all three (e.g. Data General's RDOS). More commonly, a system will use one particular method for all files.

Contiguous Allocation :

The contiguous allocation method requires each file to occupy a set of contiguous address on the disk. Disk addresses define a linear ordering on the disk. Notice that, with this ordering, 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), it is only one track. Thus, the number of disk seeks required for accessing contiguous allocated files is minimal, as is seek time when a seek is finally needed. Contiguous allocation of a file is defined by the disk address and the length of the first block. If the file is n blocks long, and starts at location b , then it occupies blocks $b, b+1, b+2, \dots, b+n-1$. The directory entry for each file indicates the address of the starting block and the length of the area allocated for this file.

The difficulty with contiguous allocation is finding space for a new file. If the file to be created is n blocks long, then the OS must search for n free contiguous blocks. First-fit, best-fit, and worst-fit strategies (as discussed in Chapter 4 on multiple partition allocation) are the most common strategies used to select a free hole from the set of available holes. Simulations have shown that both first-fit and best-fit are better than worst-fit in terms of both time and storage utilization. Neither first-fit nor best-fit is clearly best in terms of storage utilization, but first-fit is generally faster.

These algorithms also suffer from external fragmentation. As files are allocated and deleted, the free disk space is broken into little pieces. External fragmentation exists when enough total disk space exists to satisfy a request, but this space is not contiguous; storage is fragmented into a large number of small holes.

Another problem with contiguous allocation is determining how much disk space is needed for a file. When the file is created, the total amount of space it will need must be known and allocated. How does the creator (program or person) know the size of the file to be created. In some cases, this determination may be fairly simple (e.g. copying an existing file), but in general the size of an output file may be difficult to estimate.

Linked Allocation :

The problems in contiguous allocation can be traced directly to the requirement that the spaces be allocated contiguously and that the files that need these spaces are of different sizes. These requirements can be avoided by using linked allocation.

In linked allocation, each file is a linked list of disk blocks. The directory contains a pointer to the first and (optionally the last) block of the file. For example, a file of 5 blocks which starts at block 4, might continue at block 7, then block 16, block 10, and finally block 27. Each block contains a pointer to the next block and the last block contains a NIL pointer. The value -1 may be used for NIL to differentiate it from block 0.

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. A write to a file removes the first free

block and writes to that block. This new block is then linked to the end of the file. To read a file, the pointers are just followed from block to block.

There is no external fragmentation with linked allocation. Any free block can be used to satisfy a request. Notice also that there is no need to declare the size of a file when that file is created. A file can continue to grow as long as there are free blocks. Linked allocation, does have disadvantages, however. The major problem is that it is inefficient to support direct-access; it is effective only for sequential-access files. To find the *i*th block of a file, it must start at the beginning of that file and follow the pointers until the *i*th block is reached. Note that each access to a pointer requires a disk read.

Another severe problem is reliability. A bug in OS or disk hardware failure might result in pointers being lost and damaged. The effect of which could be picking up a wrong pointer and linking it to a free block or into another file.

Indexed Allocation :

The indexed allocation method is the solution to the problem of both contiguous and linked allocation. This is done by bringing all the pointers together into one location called the index block. Of course, the index block will occupy some space and thus could be considered as an overhead of the method in indexed allocation, each file has its own index block, which is an array of disk sector addresses. The *i*th entry in the index block points to the *i*th sector of the file. The directory contains the address of the index block of a file. To read the *i*th sector of the file, the pointer in the *i*th index block entry is read to find the desired sector. Indexed allocation supports direct access, without suffering from external fragmentation. Any free block anywhere on the disk may satisfy a request for more space.

File access methods :

There are several ways that the information in the file can be accessed. Some systems provide only one access method for files. On other systems, many different access methods are supported, and choosing the right one for a particular application is a major design problem.

Sequential Access :

Information in the file is processed in order, one record after the other. This is by far the most common mode of access of files. For example, computer editors usually access files in this fashion. A read operation reads the next portion of the file and automatically advances the file pointer. Similarly, a write appends to the end of the file and the file pointer. Similarly, a write appends to the end of the file and advances to the end of the newly written material (the new end of file). Such a file can be reset to the beginning, and, on some systems, a program may be able to skip forward or backward *n* records, for some integer *n*. This scheme is known as sequential access to a file. Sequential access is based on a tape model of a file.

Direct Access :

Direct access is based on a disk model of a file. For direct access, the file is viewed as a numbered sequence of block or records. A direct-access file allows arbitrary blocks to be read or written. Thus, after block 18 has been read, block 57 could be next, and then block 3. There are no restrictions on the order of reading and writing for a direct access file. Direct access files are of great use for intermediate access to large amounts of information.

The file operations must be modified to include the block number as a parameter. Thus, we have "read n", where n is the block number, rather than "read next", and "write n", rather than "write next". An alternative approach is to retain "read next" and "write next" and to add an operation; "position file to n" where n is the block number. Then, to effect a "read n", we would issue the commands "position to n" and then "read next".

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

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

Ans. Advantages of distributed system over centralized system :

- **Economics** : a collection of microprocessors offer a better price/performance than mainframes. Low price/performance ratio: cost effective way to increase computing power.

- **Speed** : a distributed system may have more total computing power than a mainframe. Ex. 10,000 CPU chips, each running at 50 MIPS. Not possible to build 500,000 MIPS single processor since it would require 0.002 nsec instruction cycle. Enhanced performance through load distributing.

- **Inherent distribution** : Some applications are inherently distributed. Ex. a supermarket chain.

- **Reliability** : If one machine crashes, the system as a whole can still survive. Higher availability and improved reliability.

- **Incremental growth** : Computing power can be added in small increments. Modular expendability

- **Another deriving force** : The existence of large number of personal computers, the need for people to collaborate and share information.

Q. 5. (a) Discuss the following :

- Critical section problem**
- Race condition**
- Priority inversion problem**
- Monitors**

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

- Set of instructions that must be controlled so as to allow exclusive access to one process
- Execution of the critical section by processes is mutually exclusive in time.
- Critical Section (S & G, p. 166) (for example, "for the process table")

repeat

entry section

critical section

exit section

remainder section

until FALSE

Solution to the Critical Section Problem must meet three conditions.

1. Mutual Exclusion :

If process p_i is executing in its critical section, no other process is executing in its critical section

2. Progress :

If no process is executing in its critical section and there exists some processes that wish to enter their critical sections, then only those processes that are not executing in their remainder section can participate in the decision of which will enter its critical section next and this decision cannot be postponed indefinitely.

- If no process is in critical section, can decide quickly who enters.
- Only one process can enter the critical section so in practice, others are put on the queue

3. Bounded Waiting :

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

- The wait is the time from when a process makes a request to enter its critical section until that request is granted.
- In practice, once a process enters its critical section, it does not get another turn until a waiting process gets a turn (managed as a queue).

(ii) Race Condition : Conditions for race conditions to happen

- Concurrent processes/tasks access shared variables.
- Preemption/interruption at a "wrong" time.
- Atomic section : section of code that cannot be interrupted by another process.
- Critical section : section of code that must not be concurrently accessed by more than one thread of execution.
- POSIX:
- preemptive scheduling \square race among processes.
- TinyOS.
- Non-preemptive scheduling for tasks \square no race among tasks.

Solution to Race Condition :

- Race-Free Invariant
- Any update to shared state is either not a potential race condition (SC only), or occurs within an atomic section.

- Compiler identifies all shared states and return errors if the above invariant is violated.
- How to fix race condition?
- Make the access to all shared states with potential race conditions atomic.
- Move access to SC.

Atomic Sections :

atomic { <Statement list> }

- Implements critical region.
- Disable interrupt when atomic code is being executed.
- But cannot disable interrupt for long! □ restrictions.
- No loops.
- No commands/events.
- Calls OK, but called must meet restrictions too.

(iii) Priority Inversion Problem :

In scheduling, priority inversion is the scenario where a low priority task holds a shared resource that is required by a high priority task. This causes the execution of the high priority task to be blocked until the low priority task has released the resource, effectively "inverting" the relative priorities of the two tasks. If some other medium priority task, one that does not depend on the shared resource, attempts to run in the interim, it will take precedence over both the low priority task and the high priority task. In some cases, priority inversion can occur without causing immediate harm-the delayed execution of the high priority task goes unnoticed, and eventually the low priority task releases the shared resource. However, there are also many situations in which priority inversion can cause serious problems.

(iv) Monitors :

A monitor is an approach to synchronize two or more computer tasks that use a shared resource, usually a hardware device or a set of variables. With monitor-based concurrency, the compiler or interpreter transparently inserts locking and unlocking code to appropriately designated procedures, instead of the programmer having to access concurrency primitives explicitly. A monitor consists of :

- Set of procedures that allow interaction with the shared resource.
- A mutual exclusion lock.
- The variables associated with the resource.
- A monitor invariant that defines the assumptions needed to avoid race conditions

A monitor procedure takes the lock before doing anything else, and holds it until it either finishes or waits for a condition (explained below). If every procedure guarantees that the invariant is true before it releases the lock, then no task can ever find the resource in a state that might lead to a race condition.

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. Binary Semaphore :

Binary semaphore is sufficient for mutex

- Binary semaphore has boolean value (not integer)
- bsem_wait() : Waits until value is 1, then sets to 0
- bsem_signal() : Sets value to 1, waking one waiting process

General semaphore is also called counting semaphore

Binary semaphore signal/wait routines

Signal(s)

S=s+1;

Wait(s)

Try_again : if(s>0) then (if s is 1)

S = 0

Else

Go to try_again;

- busy-waiting in the wait() procedure is undesirable in time-sharing systems- busy waiting process ties up the processor, but does not execute any useful instruction
- say process Q has to perform signal(s) to allow waiting process P to run - more efficient allow Q to run
- wait(s)
try_again : if (test-and-set (addr(s))==0) then
{ yield();
Goto try_again; }

Indiscriminate use of semaphores can lead the system to be stuck in a state from which it never emerges.

- Directed cycle in interprocess dependencies
- Process P waiting for Q which is waiting for P.



We want to implement counting semaphores using binary semaphores as the only synchronization construct, that is, without accessing pcbs, disabling interrupts, etc. We also do not want an implementation that uses busy waiting,

Consider an implementation of a counting semaphore S. Clearly the implementation needs to keep track of the value of S, say in an integer variable S.val. Furthermore, if a process is supposed to wait on S, then the implementation has to make the process wait. Because binary semaphores are the only synchronization construct allowed in the implementation, the only way that the implementation can make the process wait is to have it wait on a binary semaphore, say S.wait (recall that busy waiting is not acceptable).

These considerations lead us to the following first cut at an implementation :

| Counting Semaphore construct | Implementation using Binary Semaphores |
|------------------------------|---|
| Semaphore S initially K | record S { integer val initially K, BinarySemaphore wait initially 0 } |
| P(S) | P(S) { if S.val = 0 then { P (S.wait) } else { S.val := S.val - 1 } } |
| V(S) | V(S) { if "processes are waiting on S.wait" then { V (S.wait) } else { S.val := S.val + 1 } } |

Q. 6. Explain I/O Device organization. Describe I/O Interrupts and I/O Buffering.

Ans. Section 6 I/O Interrupt :

The temporary stopping of the current program routine, in order to execute some higher priority I/O subroutine, is called an interrupt. The interrupt mechanism in the CPU forces a branch out of the current program routine to one of several subroutines, depending upon which level of interrupt occurs.

I/O operations are started as a result of the execution of a program instruction. Once started, the I/O device continues its operation at the same time that the job program is being executed. Eventually the I/O operation reaches a point at which a program routine that is related to the I/O operation must be executed. At that point an interrupt is requested by the I/O device involved. The interrupt action results in a forced branch to the required subroutine.

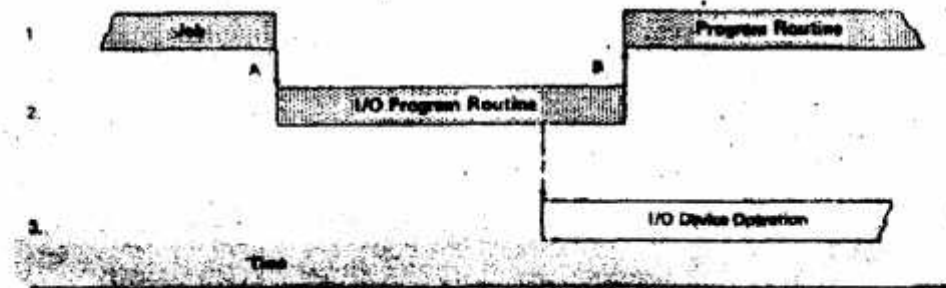
In addition to the routine needed to start an I/O operation, subroutines are required to :

1. Transfer a data word between an I/O device and main storage (for write or read operations).
2. Handle unusual (or check) conditions related to the I/O device..

3. Handle the ending of the I/O device operation.

Shown in the following diagram are :

1. The job program routine (which, for this discussion, includes those program steps not used for I/O operation handling).



2. The I/O program routine that starts an I/O device.
3. The I/O device operation (such as moving a punched card through the read feed of a card reader).

Branching from the job routine to the I/O routine occurs at point A in the diagram. This branch is a program controlled operation that is started because the job program is at a point at which the I/O operation is required (such as the reading of a card in the card reader). Similarly, when the end of the I/O routine is reached, a program-controlled branch is made back to the job routine (point B). (Program-controlled means that the logical point at which the branch occurs is determined by the program; no forcing is performed by the CPU.)

I/O Buffering :

I/O accesses are reads or writes (e.g., to files)

Application access is arbitrary (offset, len)

Convert accesses to read/write of fixed-size blocks or pages

Blocks have an (object, logical block) identity

Blocks/pages are cached in memory

- Spatial and temporal locality.
- Fetch/replacement issues just as VM paging
- Trade off of block size.

A driver that services an interactive or slow device, or one that usually transfers relatively small amounts of data at a time, should use the buffered I/O transfer method. Using buffered I/O for small, interactive transfers improves overall physical memory usage, because the memory manager doesn't need to lock down a full physical page for each transfer, as it does for drivers that request direct I/O. Generally, video, keyboard, mouse, serial, and parallel drivers request buffered I/O.

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

Ans. Thread :

A thread of execution is a fork of a computer program into two or more concurrently running tasks. The implementation of threads and processes differs from one operating system to another, but in general, a thread is contained inside a process and different threads in the same process share some resources (most commonly memory), while different processes do not. On a single processor, multithreading generally occurs by time-division multiplexing (as in multitasking): the processor switches between different threads. This context switching generally happens frequently enough that the user perceives the threads or tasks to be running at the same time. On a multiprocessor or multi-core system, the threads or tasks actually do run at the same time, with each processor or core running a particular thread or task.

User-level, vs Kernel Level Threads :

There are two distinct models of thread controls, and they are user-level threads and kernel-level threads. The thread function library to implement user-level threads usually runs on top of the system in user mode. Thus, these threads within a process are invisible to the operating system. User-level threads have extremely low overhead, and can achieve high performance in computation. However, using the blocking system calls like read(), the entire process would block. Also, the scheduling control by the thread runtime system may cause some threads to gain exclusive access to the CPU and prevent other threads from obtaining the CPU. Finally, access to multiple processors is not guaranteed since the operating system is not aware of existence of these types of threads.

On the other hand, kernel-level threads will guarantee multiple processor access but the computing performance is lower than user-level threads due to load on the system. The synchronization and sharing resources among threads are still less expensive than multiple-process model, but more expensive than user-level threads. The thread function library available today is often implemented as a hybrid model, as having advantages from both user-level and kernel-level threads. The design consideration of thread packages today consists of how to minimize the system overhead while providing access to the multiple processors

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

Ans. Interrupt :

An interrupt is an asynchronous signal from hardware indicating the need for attention or a synchronous event in software indicating the need for a change in execution. A hardware interrupt causes the processor to save its state of execution via a context switch, and begin execution of an interrupt handler. Software interrupts are usually implemented as instructions in the instruction set, which cause a context switch to an interrupt handler similar to a hardware interrupt. Interrupts are a commonly used technique for computer multitasking, especially in real-time computing. Such a system is said to be interrupt-driven.

Trap :

A trap is a form of exception and therefore can usually be recovered from.

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

- (i) **Swapping**
- (ii) **Disk Scheduling**
- (iii) **Symmetric v/s Asymmetric Multiprocessing**
- (iv) **Deadlock**
- (v) **Bankers algorithm.**

Ans. (i) Swapping :

When the physical memory in the system runs out and a process needs to bring a page into memory then the operating system must decide what to do. It must fairly share the physical pages in the system between the processes running in the system, therefore it may need to remove one or more pages from the system to make room for the new page to be brought into memory. How virtual pages are selected for removal from physical memory affects the efficiency of the system. Linux uses a page aging technique to fairly choose pages which might be removed from the system. This scheme involves every page in the system having an age which

changes as the page is accessed. The more that a page is accessed, the younger it is; the less that it is accessed the older it becomes. Old pages are good candidates for swapping. If the page to be removed came from an image or data file and has not been written to then the page does not need to be saved. Instead it can be discarded and if the process needs that page again it can be brought back into memory from the image or data file again. However, if the page has been written to then the operating system must preserve the contents of that page so that it can be accessed at a later time.

(ii) Disk scheduling :

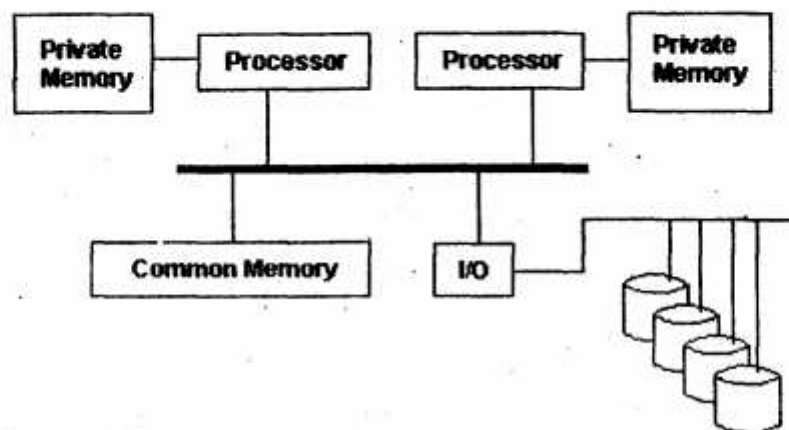
In multiprogramming systems several different processes may want to use the system's resources simultaneously. For example, processes will contend to access an auxiliary storage device such as a disk. The disk drive needs some mechanism to resolve this contention, sharing the resource between the processes fairly and efficiently. In order to satisfy an I/O request the disk controller must first move the head to the correct track and sector. Moving the head between cylinders takes a relatively long time so in order to maximise the number of I/O requests which can be satisfied the scheduling policy should try to minimise the movement of the head. On the other hand, minimising head movement by always satisfying the request of the closest location may mean that some requests have to wait a long time. Thus, there is a trade-off between throughput (the average number of requests satisfied in unit time) and response time (the average time between a request arriving and it being satisfied). Various different disk scheduling policies are used: First Come First Served (FCFS),

Shortest Seek Time First (SSTF), SCAN, Circular SCAN (C-SCAN), LOOK, Circular LOOK (C-LOOK).

(iii) Symmetric vs Asymmetric multiprocessing: Asymmetric multiprocessing.

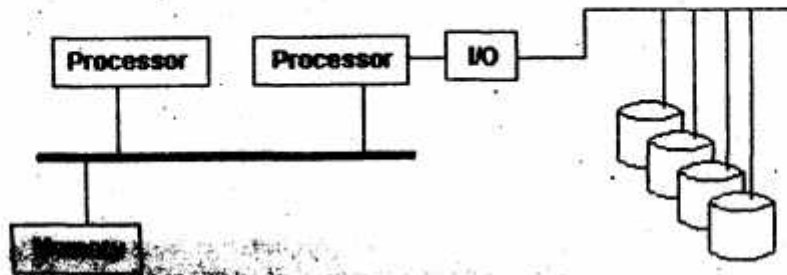
In asymmetric multiprocessing the program tasks (or threads) are strictly divided by type between processors and typically, each processor has its own memory address space. These features make asymmetric multiprocessing difficult to implement. The two figures below are two examples of asymmetric multiprocessor configurations. The PS/2 Server 195 and Server 295 were examples of servers using asymmetric multiprocessing.

Asymmetric multiprocessing Example 1 :



This configuration has multiple memory units with some of those not shared by all processors.

Asymmetric multiprocessing Example 2 :

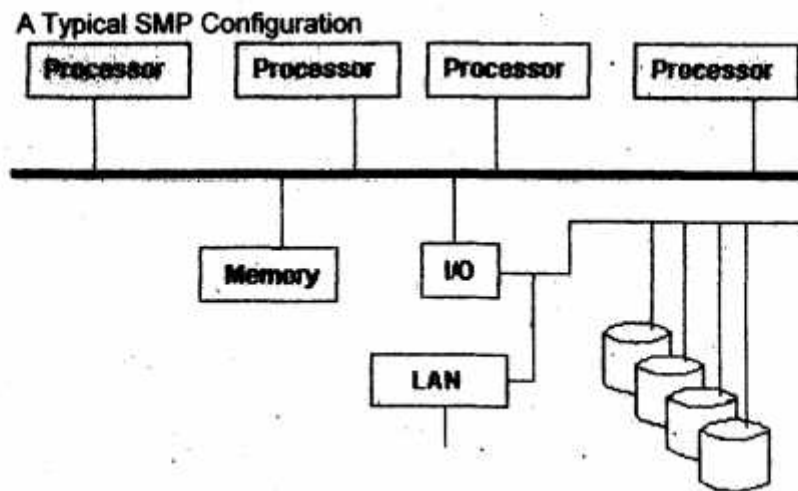


This configuration has one processor doing all I/O.

Symmetric multiprocessing :

Symmetric multiprocessing (SMP) is the most common configuration of multiple processors. A typical SMP configuration has the following items: It has multiple processors and exactly one of everything else: memory, I/O subsystem, operating system, etc. The processors are symmetric, that is, they can do anything the others can. Each can look at or alter any element of memory, and each can do any kind of I/O. It is symmetrical because the view from any processor of the rest of the system is exactly the same. The display below shows a typical implementation of SMP.

A Typical SMP configuration :



SMP is easier to implement in operating systems and is the method used most often in operating systems that support multiple processors. Operating systems that support SMP include :

OS/2 Warp Server 4.0

OS/2 for SMP 2.11

Windows NT 4.0

Novell NetWare 4.1 SMP

Novell UnixWare SMP 2.0

SCO Open Server 5.0 with seo MPX 3.0

Banyan Vines

(iv) Deadlock :

A deadlock is a situation in which two computer programs sharing the same resource are effectively preventing each other from accessing the resource, resulting in both programs ceasing to function. The earliest computer operating systems ran only one program at a time. All of the resources of the system were available to this one program. Later, operating systems ran multiple programs at once, interleaving them. Programs were required to specify in advance what resources they needed so that they could avoid conflicts with other programs running at the same time. Eventually some operating systems offered dynamic allocation of resources. Programs could request further allocations of resources after they had begun running. This led to the problem of the deadlock. Here is the simplest example :

Program 1 requests resource A and receives it.

Program 2 requests resource B and receives it.

Program 1 requests resource B and is queued up, pending the release of B.

Program 2 requests resource A and is queued up, pending the release of A.

Now neither program can proceed until the other program releases a resource. The operating system cannot know what action to take. At this point the only alternative is to abort (stop) one of the programs.

(v) Bankers Algorithm : Processes request 1 resource at a time.

- Request granted ONLY if it results in a safe state.
 - If request results in an unsafe state, it is denied and user holds resources and waits until request is eventually satisfied.
 - In finite time, all requests will be satisfied.
 - Can be extended to cater for multiple resource types.
 - + : No deadlock and not as restrictive as deadlock prevention
 - : Fixed number of resources and users
 - Guarantees finite time-NOT-reasonable response time.
 - Needs advanced knowledge of maximum needs.
- Not suitable for multi-access systems.
- Unnecessary delays in avoiding unsafe states which may not lead to deadlock.

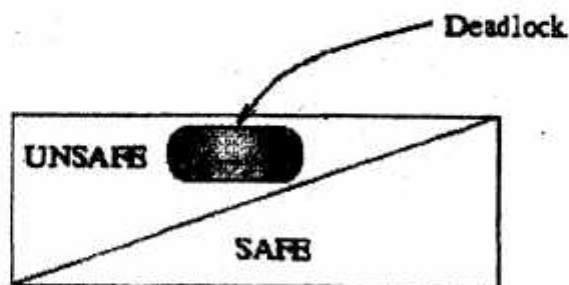


Figure 2 : Bankers algorithm (safe/unsafe states)

Therefore, from Bankers algorithm :

Allow a thread (process) to proceed if :

$(\text{Total available resources}) - (\text{number allocated})$

$\geq (\text{maximum remaining that might be needed by the thread (process)})$.