T.E. (Computer Science and Engg.) (Part -1) OPERATING SYSTEM -1

MCQ/Objective Type Questions

1. Attempt the following MCQ:

1) Round robin scheduling is essentially the preemptive version of _____.
1 FIFO
2 Shortest job first
3 Shortes remaining
4 Longest time first

2) A page fault occurs
1 when the page is not in the memory
2 when the page is in the memory
3 when the process enters the blocked state
4 when the process is in the ready state

3) Which of the following will determine your choice of systems software for your computer ?
1 Is the applications software you want to use compatible with it ?
2 Is it expensive ?
3 Is it compatible with your hardware ?
4 Both 1 and 3

4) Let S and Q be two semaphores initialized to 1, where P0 and P1 processes the following statements wait(S);wait(Q); ---; signal(S);signal(Q) and wait(Q); wait(S);---;signal(Q);signal(S); respectively. The above situation depicts a _____ .
1 Semaphore
2 Deadlock
3 Signal
4 Interrupt

5) What is a shell ?
1 It is a hardware component
2 It is a command interpreter
3 It is a part in compiler
4 It is a tool in CPU scheduling

6) Routine is not loaded until it is called. All routines are kept on disk in a relocatable load format. The main program is loaded into memory & is executed. This type of loading is called _____
1 Static loading
2 Dynamic loading
3 Dynamic linking
4 Overlays

7) In the blocked state
1 the processes waiting for I/O are found
2 the process which is running is found
3 the processes waiting for the processor are found
4 none of the above

8) What is the memory from 1K - 640K called ?
1 Extended Memory
2 Normal Memory
3 Low Memory
4 Conventional Memory

9) Virtual memory is _____.
1 An extremely large main memory
2 An extremely large secondary memory
3 An illusion of extremely large main memory
4 A type of memory used in super computers.

10) The process related to process control, file management, device management, information about system and communication that is requested by any higher level language can be performed by _____.
1 Editors
2 Compilers
3 System Call

4 Caching

11) If the Disk head is located initially at 32, find the number of disk moves required with FCFS if the disk queue of I/O blocks requests are 98,37,14,124,65,67.
1 310
2 324
3 315
4 321

12) Multiprogramming systems _____.
1 Are easier to develop than single programming systems
2 Execute each job faster
3 Execute more jobs in the same time
4 Are used only on large main frame computers

13) Which is not the state of the process ?
1 Blocked
2 Running
3 Ready
4 Privileged

14) The solution to Critical Section Problem is : Mutual Exclusion, Progress and Bounded Waiting.
1 The statement is false
2 The statement is true.
3 The statement is contradictory.
4 None of the above

15) The problem of thrashing is effected scientifically by _____.
1 Program structure
2 Program size
3 Primary storage size
4 none of the above

16) The state of a process after it encounters an I/O instruction is _____.

1 Ready
2 blocked/Waiting
3 Idle
4 Running

17) The number of processes completed per unit time is known as _____.
1 Output
2 Throughput
3 Efficiency
4 Capacity

18) _____ is the situation in which a process is waiting on another process, which is also waiting on another process ... which is waiting on the first process. None of the processes involved in this circular wait are making progress.
1 Deadlock
2 Starvation
3 Dormant
4 None of the above

19) Which of the following file name extension suggests that the file is Backup copy of another file ?
1 TXT
2 COM
3 BAS
4 BAK

20) Which technique was introduced because a single job could not keep both the CPU and the I/O devices busy?
1 Time-sharing
2 SPOOLing
3 Preemptive scheduling
4 Multiprogramming

SECTION- I

1. a) What is a process? Write and explain a typical 'Process Control Block' (PCB).

Ans :

A process in an operating system is represented by a data structure known as a process control block (PCB) or process descriptor. The PCB contains important information about the specific process including

- The current state of the process i.e., whether it is ready, running, waiting, or whatever.
- Unique identification of the process in order to track "which is which" information.
- A pointer to parent process.
- Similarly, a pointer to child process (if it exists).
- The priority of process (a part of CPU scheduling information).
- Pointers to locate memory of processes.
- A register save area.
- The processor it is running on.

The PCB is a certain store that allows the operating systems to locate key information about a process. Thus, the PCB is the data structure that defines a process to the operating systems.

2. b) Explain in detail the following schedulers:

   i) Short-term schedulers

   Ans :

   The short-term scheduler (also known as the CPU scheduler) decides which of the ready, in-memory processes are to be executed (allocated a CPU) next following a clock interrupt, an IO interrupt, an operating system call or another form of signal. Thus the short-term scheduler makes scheduling decisions much more frequently than the long-term or mid-term schedulers - a scheduling decision will at a minimum have to be made after every time slice, and these are very short. This scheduler can be preemptive, implying that it is capable of forcibly removing processes from a CPU when it decides to allocate that CPU to another

process, or non-preemptive (also known as "voluntary" or "co-operative"), in which case the scheduler is unable to "force" processes off the CPU. [Stallings, 396].

ii) Long-term schedulers

Ans:

The long-term, or admission, scheduler decides which jobs or processes are to be admitted to the ready queue; that is, when an attempt is made to execute a program, its admission to the set of currently executing processes is either authorized or delayed by the long-term scheduler. Thus, this scheduler dictates what processes are to run on a system, and the degree of concurrency to be supported at any one time - ie: whether a high or low amount of processes are to be executed concurrently, and how the split between IO intensive and CPU intensive processes is to be handled. In modern OS's, this is used to make sure that real time processes get enough CPU time to finish their tasks. Without proper real time scheduling, modern GUI interfaces would seem sluggish. [Stallings, 399].

Long-term scheduling is also important in large-scale systems such as batch processing systems, computer clusters, supercomputers and render farms. In these cases, special purpose job scheduler software is typically used to assist these functions, in addition to any underlying admission scheduling support in the operating system.

iii) Medium-term schedulers.

Ans:

The mid-term scheduler temporarily removes processes from main memory and places them on secondary memory (such as a disk drive) or vice versa. This is commonly referred to as "swapping out" or "swapping in" (also incorrectly as "paging out" or "paging in"). The mid-term scheduler may decide to swap out a process which has not been active for some time, or a process which has a low priority, or a process which is page faulting frequently, or a process which is taking up a large amount of memory in order to free up main memory for other processes, swapping the process back in later when more memory is available, or when the process has been unblocked and is no longer waiting for a resource. [Stallings,

396] [Stallings, 370]

In many systems today (those that support mapping virtual address space to secondary storage other than the swap file), the mid-term scheduler may actually perform the role of the long-term scheduler, by treating binaries as "swapped out processes" upon their execution. In this way, when a segment of the binary is required it can be swapped in on demand, or "lazy loaded". [Stallings, 394]

3. a) Explain the working of Multiprogramming OS in detail.

Ans :

Multiprogramming: When multitasking is just talking about executing multiple programs concurrently then the term multitasking term is referred as multiprogramming.

In the early days of computing, CPU time was expensive, and peripherals were very slow. When the computer ran a program that needed access to a peripheral, the CPU would have to stop executing program instructions while the peripheral processed the data. This was deemed very inefficient. The first computer using a multitasking system was the British *Leo III* owned by J. Lyons and Co.. Several different programs in batch were loaded in the computer memory, and the first one began to run. When the first program reached an instruction waiting for a peripheral, the context of this program was stored away, and the second program in memory was given a chance to run. The process continued until all programs finished running.

Multiprogramming doesn't give any guarantee that a program will run in a timely manner. Indeed, the very first program may very well run for hours without needing access to a peripheral. As there were no users waiting at an interactive terminal, this was no problem: users handed on a deck of punched cards to an operator, and came back a few hours later for printed results. Multiprogramming greatly reduced wait times when multiple batches were being processed.

4. Explain distributed systems and its advantages.

Ans :

Distributed computing is a field of computer science that studies distributed systems. A **distributed system** consists of multiple autonomous computers that communicate through a computer network. The computers interact with each other in order to achieve a common goal. A computer program that runs in a distributed system is called a **distributed program**, and **distributed programming** is the process of writing such programs.[1]

Distributed computing also refers to the use of distributed systems to solve computational problems. In distributed computing, a problem is divided into many tasks, each of which is solved by one or more computers.[2]

Advantages of Distributed Systems over Centralized ones

1:Incremental growth:Computing power can be added in small increments

2:Reliability:If one machine crashes, the system as a whole can still survive

3:Speed:
A distributed system may have more total computing power than a mainframe

4:Open system:
This is the most important point and themost characteristic point of a distributed system.
Since it is an open system it is always ready to communicate with other systems.
an open system that scales has an advantage over a perfectly closed and self-contained system.

Economic:ANd Microprocessors offer a better price/performance than mainframes

5 Write and explain an algorithm for the reader's writer's problem in

Ans :

In computer science, the **first and second readers-writers problems** are examples of a common computing problem in concurrency. The two problems deal with situations in which many threads must access the same shared memory at one time, some reading and some writing, with the natural constraint that no process may access the share for reading or writing while another process is in the act of writing to it. (In particular, it *is* allowed for two or more readers to access the share at the same time.) A readers-writer lock is a data structure that solves one or more of the readers-writers problems.

The **third readers-writers problem** is sometimes proposed, which adds the constraint that *no thread shall be allowed to starve*; that is, the operation of obtaining a lock on the shared data will always terminate in a bounded amount of time.

A solution with fairness for both readers and writers might be as follows:

semaphores: no_writers, no_readers, counter_mutex (initial value is 1)
shared variables: nreaders (initial value is 0)
local variables: prev, current

WRITER:

```
    P( no_writers );
      P( no_readers );
      V( no_readers );
      ...  write ...
    V( no_writers );

READER:
    P( no_writers );
      P( counter_mutex );
        prev := nreaders;
        nreaders := nreaders + 1;
      V( counter_mutex );
      if prev = 0  then P( no_readers );
    V( no_writers );
    ... read ...
    P( counter_mutex );
      nreaders := nreaders - 1;
      current := nreaders;
    V( counter_mutex );
    if current = 0 then V( no_readers );
```

Note that sections protected by counter_mutex could be replaced by a suitable fetch-and-add atomic instruction, saving two potential context switches in reader's code.

Note also that this solution can only satisfy the condition that "no thread shall be allowed to starve" if and only if semaphores preserve first-in first-out ordering when blocking and releasing threads. Otherwise, a blocked writer, for example, may be continually preempted by newly blocked writers.

Q.2 Write short notes on :

a) Spooling

Ans:

In computer science, **spooling** refers to the process of placing data in a temporary working area for another program to process. The most common use is in writing files on a magnetic tape or disk and entering them in the work queue (possibly just linking it to a designated folder in the file system) for another process. Spooling is useful because devices access data at different rates. Spooling allows one program to assign work to another without directly communicating with it.

The most common spooling application is print spooling: documents formatted for printing are stored usually into an area on a disk and retrieved and printed by a printer at its own rate. Printers typically can print only a single document at a time and require seconds or minutes to do so. With spooling, multiple processes can write documents to a print queue without waiting. As soon as a process has written its document to the spool device, the process can perform other tasks, while a separate printing process operates the printer.

For example, when a city prepares payroll checks, the actual computation may take a matter of minutes or even seconds, but the printing process might take hours. If the program printed

directly, computing resources (CPU, memory, peripherals) would be tied up until the program was able to finish. The same is true of personal computers. Without spooling, a word processor would be unable to continue until printing finished. Without spooling, most programs would be relegated to patterns of fast processing and long waits, an inefficient paradigm.[1]

Spooler or print management software may allow priorities to be assigned to jobs, notify users when they have printed, distribute jobs among several printers, allow stationery to be changed or select it automatically, generate banner pages to identify and separate print jobs, etc.

b) Test and set

Ans :

In computer science, the **test-and-set** instruction is an instruction used to write to a memory location and return its old value as a single atomic (*i.e.* non-interruptible) operation. If multiple processes may access the same memory, and if a process is currently performing a test-and-set, no other process may begin another test-and-set until the first process is done. CPUs may use test-and-set instructions offered by other electronic components, such as Dual-Port RAM; CPUs may also offer a test-and-set instruction themselves.

A lock can be built using an atomic test-and-set instruction as follows:

```
function Lock(boolean *lock) {
   while (test_and_set (lock) == 1)
     ;
}
```

The calling process obtains the lock if the old value was 0. It spins writing 1 to the variable until this occurs.

c) Semaphore.

Ans :

In computer science, a **semaphore** is a protected variable or abstract data type that provides a simple but useful abstraction for controlling access by multiple processes to a common resource in a parallel programming environment.

A useful way to think of a semaphore is as a record of how many units of a particular resource are available, coupled with operations to *safely* (i.e. without race conditions) adjust that record as units are required or become free, and if necessary wait until a unit of the resource becomes available. Semaphores are a useful tool in the prevention of race conditions and deadlocks; however, their use is by no means a guarantee that a program is free from these problems. Semaphores which allow an arbitrary resource count are called **counting semaphores**, whilst semaphores which are restricted to the values 0 and 1 (or locked/unlocked, unavailable/available) are called **binary semaphores**.

Q.3 A) What is critical-section problem? How it is solved using semaphores?

Ans :

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

critical 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)

Semaphores

- originally, semaphores were flags for signalling between ships
- a variable used for signalling between processes

- operations possible on a semaphore:
    - initialization

- done before individual processes try to operate on the semaphore

- two main operations:
  - **wait** (or **acquire**)
  - **signal** (or **release**)
- the **wait** and **signal** operations are atomic operations (e.g., the test-and-set at the top of the loop of **wait** is done before losing the processor)
- e.g., A resource such as a shared data structure is protected by a semaphore. You must acquire the semaphore before using the resource.

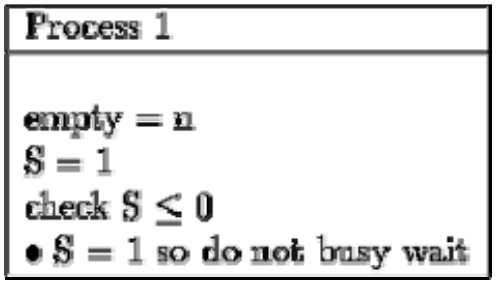
wait(S):
while S $\leq$ 0 **do** no-op;
S := S - 1;

signal(S):
S := S + 1;

In either case, the initial value for S:

1. equals 1 if only one process is allowed in the critical section (binary semaphore)
2. equals *n* if at most *n* processes are allowed in the critical section

Semaphore Solution to the Critical Selection Problem
repeat
wait(mutex);
critical section
signal(mutex);
remainder section
until false;



B) Describe Dinning-Philosophers problem, Specify its solution.

Ans :

The dining philosophers problem is summarized as five silent philosophers sitting at a circular table doing one of two things: eating or thinking. While eating, they are not thinking, and while thinking, they are not eating. A large bowl of Spaghetti is placed in the center, which requires two forks to serve and to eat (the problem is therefore sometimes explained using rice and chopsticks rather than spaghetti and forks). A fork is placed in between each pair of adjacent philosophers, and each philosopher may only use the fork to his left and the fork to his right. However, the philosophers do not speak to each other.

Solutions

[edit] Conductor solution

A relatively simple solution is achieved by introducing a waiter at the table. Philosophers must ask his permission before taking up any forks. Because the waiter is aware of which forks are in use, he is able to arbitrate and prevent deadlock. When four of the forks are in use, the next philosopher to request one has to wait for the waiter's permission, which is not given until a fork has been released. The logic is kept simple by specifying that philosophers always seek to pick up their left hand fork before their right hand fork (or vice versa).

To illustrate how this works, consider that the philosophers are labelled clockwise from A to E. If A and C are eating, four forks are in use. B sits between A and C so has neither fork available, whereas D and E have one unused fork between them. Suppose D wants to eat. Were he to take up the fifth fork, deadlock becomes likely. If instead he asks the waiter and is told to wait, we can be sure that next time two forks are released there will certainly be at least one philosopher who could successfully request a pair of forks. Therefore deadlock cannot happen.

[edit] Resource hierarchy solution

Another simple solution is achieved by assigning a partial order to the resources (the forks, in this case), and establishing the convention that all resources will be requested in order, and released in reverse order, and that no two resources unrelated by order will ever be used by a single unit of work at the same time. Here, the resources (forks) will be numbered 1 through 5, in some order, and each unit of work (philosopher) will always pick up the lower-numbered fork first, and then the higher-numbered fork, from among the two forks he plans to use. Then, he will always put down the higher numbered fork first, followed by the lower numbered fork. In this case, if four of the five philosophers simultaneously pick up their lower-numbered fork, only the

highest numbered fork will remain on the table, so the fifth philosopher will not be able to pick up any fork. Moreover, only one philosopher will have access to that highest-numbered fork, so he will be able to eat using two forks. When he finishes using the forks, he will put down the highest-numbered fork first, followed by the lower-numbered fork, freeing another philosopher to grab the latter and begin eating.

This solution to the problem is the one originally proposed by Dijkstra.

While the resource hierarchy solution avoids deadlocks, it is not always practical, especially when the list of required resources is not completely known in advance. For example, if a unit of work holds resources 3 and 5 and then determines it needs resource 2, it must release 5, then 3 before acquiring 2, and then it must re-acquire 3 and 5 in that order. Computer programs that access large numbers of database records would not run efficiently if they were required to release all higher-numbered records before accessing a new record, making the method impractical for that purpose.

[edit] Monitor solution

The example below shows a solution where the forks are not represented explicitly. Philosophers can eat if neither of their neighbors are eating. This is comparable to a system where philosophers that cannot get the second fork must put down the first fork before they try again.

In the absence of locks associated with the forks, philosophers must ensure that the decision to begin eating is not based on stale information about the state of the neighbors. E.g. if philosopher B sees that A is not eating, then turns and looks at C, A could begin eating while B looks at C. This solution avoids this problem by using a single mutual exclusion lock. This lock is not associated with the forks but with the decision procedures that can change the states of the philosophers. This is ensured by the monitor. The procedures *test*, *pickup* and *putdown* are local to the monitor and share a mutual exclusion lock. Notice that philosophers wanting to eat do not hold a fork. When the monitor allows a philosopher who wants to eat to continue, the philosopher will reacquire the first fork before picking up the now available second fork. When done eating, the philosopher will signal to the monitor that both forks are now available.

Notice that this example does not tackle the starvation problem. For example, philosopher B can wait forever if the eating periods of philosophers A and C always overlap.

To also guarantee that no philosopher starves, one could keep track of the number of times a hungry philosopher cannot eat when his neighbors put down their forks. If this number exceeds some limit, the state of the philosopher could change to *Starving*, and the decision procedure to pick up forks could be augmented to require that none of the neighbors are starving.

A philosopher that cannot pick up forks because a neighbor is starving, is effectively waiting for the neighbor's neighbor to finish eating. This additional dependency reduces concurrency. Raising the threshold for transition to the *Starving* state reduces this effect.

C) Explain following CPU scheduling algorithms with help of example and Gantt chart.

i) FCFS

Ans :

First-Come-First-Served algorithm is the simplest scheduling algorithm is the simplest scheduling algorithm. Processes are dispatched according to their arrival time on the ready queue. Being a nonpreemptive discipline, once a process has a CPU, it runs to completion. The FCFS scheduling is fair in the formal sense or human sense of fairness but it is unfair in the sense that long jobs make short jobs wait and unimportant jobs make important jobs wait.

FCFS is more predictable than most of other schemes since it offers time. FCFS scheme is not useful in scheduling interactive users because it cannot guarantee good response time. The code for FCFS scheduling is simple to write and understand. One of the major drawback of this scheme is that the average time is often quite long.

The First-Come-First-Served algorithm is rarely used as a master scheme in modern operating systems but it is often embedded within other schemes.

li) SJF

Ans :

Shortest-Job-First (SJF) is a non-preemptive discipline in which waiting job (or process) with the smallest estimated run-time-to-completion is run next. In other words, when CPU is available, it is assigned to the process that has smallest next CPU burst.

The SJF scheduling is especially appropriate for batch jobs for which the run times are known in advance. Since the SJF scheduling algorithm gives the minimum average time for a given set of processes, it is probably optimal.

The SJF algorithm favors short jobs (or processors) at the expense of longer ones.

The obvious problem with SJF scheme is that it requires precise knowledge of how long a job or process will run, and this information is not usually available.

The best SJF algorithm can do is to rely on user estimates of run times.

> In the production environment where the same jobs run regularly, it may be possible to provide reasonable estimate of run time, based on the past performance of the process. But in the development environment users rarely know how their program will execute.

Like FCFS, SJF is non preemptive therefore, it is not useful in timesharing environment in which reasonable response time must be guaranteed.

SECTION – II

Q.4 a) What is Page fault ? Describe the actions taken by the OS when page fault occurs.

Ans :

A **page fault** is a trap to the software raised by the hardware when a program accesses a page that is mapped in the virtual address space, but not loaded in physical memory.

The hardware that detects this situation is the memory management unit in a processor. The exception handling software that handles the page fault is generally part of an operating system. The operating system tries to handle the page fault by making the required page accessible at a location in physical memory or kills the program in case it is an illegal access.

A page fault occurs when an access to a page that has not been brought into main memory takes place. The operating system verifies the memory access, aborting the program if it is invalid. If it is valid, a free frame is located and I/O is requested to read the needed page into the free frame. Upon completion of I/O, the process table and page table are updated and the instruction is restarted.

---

when a process is executed with only few pages in memory,& when an instruction is encountered which refers to any instruction or data in some other page,which is not present in the main memory,a page fault occurs.

b) Given memory partitions of 100 k, 500 k, 200 k, 300 k and 600 k (in order). How would each of the First-fit, Best-fit and Worst-fit algorithms place processes of 212 k, 417 k, 112 k and 426 k (in order) ? Which algorithms makes the most efficient use of memory ?

Q.5 a) Consider a logical address space of eight pages of 1024 words each, mapped onto a physical memory of 32 frames.

i) How many bits are there in the logical address ?

ii) How many bits are there in physical address? '

b) Explain Banker's Algorithm for Deadlock Avoidance with example.

Ans :

The **Banker's algorithm** is a resource allocation & deadlock avoidance algorithm developed by Edsger Dijkstra that tests for safety by simulating the allocation of pre-determined maximum possible amounts of all resources, and then makes a "safe-state" check to test for possible deadlock conditions for all other pending activities, before deciding whether allocation should be allowed to continue.

Assuming that the system distinguishes between four types of resources, (A, B, C and D), the following is an example of how those resources could be distributed. *Note that this example shows the system at an instant before a new request for resources arrives. Also, the types and number of resources are abstracted. Real systems, for example, would deal with much larger quantities of each resource.*

Available system resources are:
A B C D
3 1 1 2
Processes (currently allocated resources):
  A B C D
P1 1 2 2 1
P2 1 0 3 3
P3 1 2 1 0
Processes (maximum resources):
  A B C D
P1 3 3 2 2
P2 1 2 3 4
P3 1 3 5 0

Safe and Unsafe States

A state (as in the above example) is considered safe if it is possible for all processes to finish executing (terminate). Since the system cannot know when a process will terminate, or how many resources it will have requested by then, the system assumes that all processes will eventually attempt to acquire their stated maximum resources and terminate soon afterward. This is a reasonable assumption in most cases since the system is not particularly concerned with how long each process runs (at least not from a deadlock avoidance perspective). Also, if a process terminates without acquiring its maximum resources, it only makes it easier on the system.

Given that assumption, the algorithm determines if a state is **safe** by trying to find a hypothetical set of requests by the processes that would allow each to acquire its maximum resources and then terminate (returning its resources to the system). Any state where no such set exists is an **unsafe** state.

[edit] Example

We can show that the state given in the previous example is a safe state by showing that it is possible for each process to acquire its maximum resources and then terminate.

1. P1 acquires 2 A, 1 B and 1 D more resources, achieving its maximum
    - The system now still has 1 A, no B, 1 C and 1 D resource available
2. P1 terminates, returning 3 A, 3 B, 2 C and 2 D resources to the system
    - The system now has 4 A, 3 B, 3 C and 3 D resources available
3. P2 acquires 2 B and 1 D extra resources, then terminates, returning all its resources
    - The system now has 5 A, 3 B, 6 C and 6 D resources
4. P3 acquires 1 B and 4 C resources and terminates
    - The system now has all resources: 6 A, 5 B, 7 C and 6 D
5. Because all processes were able to terminate, this state is safe

Note that these requests and acquisitions are *hypothetical*. The algorithm generates them to check the safety of the state, but no resources are actually given and no processes actually terminate. Also note that the order in which these requests are generated – if several can be fulfilled – doesn't matter, because all hypothetical requests let a process terminate, thereby increasing the system's free resources.

For an example of an unsafe state, consider what would happen if process 2 was holding 1 more unit of resource B at the beginning.

Q.6.Write short notes on

i) Demand paging

Ans :

In computer operating systems, **demand paging** (as opposed to anticipatory paging) is an application of virtual memory. In a system that uses demand paging, the operating system copies a disk page into physical memory only if an attempt is made to access it (*i.e.*, if a page fault occurs). It follows that a process begins execution with none of its pages in physical memory, and many page faults will occur until most of a process's working set of pages is located in physical memory. This is an example of lazy loading techniques.

ii) Resource allocation Graph algorithm

Ans :

This algorithm is used only if we have one instance of a resource type. In addition to the request

edge and the assignment edge a new edge called claim edge is used.

For eg:- A claim edge Pi=>Rj indicates that process Pi may request Rj in future. The claim edge is
represented by a dotted line.
• When a process Pi requests the resource Rj, the claim edge is converted to a request edge.
• When resource Rj is released by process Pi, the assignment edge Rj=>Pi is replaced by the claim edge Pi=>Rj.

* When a process Pi requests resource Rj the request is granted only if converting the request
edge Pi=>Rj to as assignment edge Rj=>Pi do not result in a cycle. Cycle detection algorithm is
used to detect the cycle. If there are no cycles then the allocation of the resource to process leave the system in safe state

iii) Virtual memory

Ans:

In computing, **virtual memory** is a memory management technique developed for multitasking kernels. This technique virtualizes a computer architecture's various hardware memory devices (such as RAM modules and disk storage drives), allowing a program to be ***designed as though***:

- there is only one hardware memory device and this "virtual" device acts like a RAM module.
- the program has, by default, sole access to this virtual RAM module as the basis for a contiguous working memory (an address space).

iv) Thrashing.

Ans:

In computer science, **thrashing** is a situation where large amounts of computer resources are used to do a minimal amount of work, with the system in a continual state of resource contention. Once started, thrashing is typically self-sustaining until something occurs to remove the original situation that led to the initial thrashing behavior.

Usually thrashing refers to two or more processes accessing a shared resource repeatedly such that serious system performance degradation occurs because the system is spending a disproportionate amount of time just *accessing* the shared resource. Resource access time may generally be considered as wasted, since it does not contribute to the advancement of any process. This is often the case when a CPU can process more information than can be held in available RAM; consequently the system spends more time preparing to execute instructions than actually executing them.