IInd Midterm session 2017-18 Subject: Operating System (V CSE-B)

Q1. What is Deadlock? Explain essential conditions for deadlock to occur?

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 waiting state. Sometimes, a waiting process is never again able to change state, because the resources it has requested are held by other waiting processes. This situation is called a deadlock.

A deadlock situation can arise if the following four conditions hold simultaneously in a system:

Mutual exclusion: At least one resource must be held in a non-sharable mode; that is, only one process at a time can use the resource.
Hold and wait: A process must be holding at least one resource and waiting to acquire additional resources that are currently being held by other processes.
No preemption: Resources cannot be preempted; that is, a resource can be released only voluntarily by the process holding it, after that process has completed its task.
Circular wait: A set { *P0* , Pl, ... , P11 }of waiting processes must exist such that *Po* is waiting for a resource held by P1, P1 is waiting for a resource held by P2, ... , Pn-1 is waiting for a resource held by *P,v* and P11 is waiting for a resource held by *Po.*

Q2. Explain Bankers algorithm for deadlock avoidance with an example?

The banker's algorithm is a resource allocation and deadlock avoidance algorithm that tests for safety by simulating the allocation for predetermined maximum possible amounts of all resources, then makes an "s-state" check to test for possible activities, before deciding whether allocation should be allowed to continue.

Following Data structures are used to implement the Banker's Algorithm:
Let **'n'** be the number of processes in the system and **'m'** be the number of resources types.
Available :

- It is a 1-d array of size **'m'** indicating the number of available resources of each type.
- Available[j] = k means there are **'k'** instances of resource type $\mathbf{R_j}$

Max :

- It is a 2-d array of size '**n*m'** that defines the maximum demand of each process in a system.
- Max[i, j] = k means process $\mathbf{P_i}$ may request at most **'k'** instances of resource type $\mathbf{R_j}$.

Allocation :

- It is a 2-d array of size **'n*m'** that defines the number of resources of each type currently allocated to each process.
- Allocation[i, j] = k means process $\mathbf{P_i}$ is currently allocated **'k'** instances of resource type $\mathbf{R_j}$

Need :

- It is a 2-d array of size **'n*m'** that indicates the remaining resource need of each process.

- Need [i, j] = k means process $\mathbf{P_i}$ currently allocated **'k'** instances of resource type $\mathbf{R_j}$
- Need [i, j] = Max [i, j] – Allocation [i, j]

$Allocation_i$ specifies the resources currently allocated to process P_i and $Need_i$ specifies the additional resources that process P_i may still request to complete its task.

Banker's algorithm consist of Safety algorithm and Resource request algorithm

Safety Algorithm

The algorithm for finding out whether or not a system is in a safe state can be described as follows:

1) Let Work and Finish be vectors of length '**m**' and '**n**' respectively.

   Initialize: Work = Available

   Finish [i] = false; for i=1, 2, ..., n

2) Find an i such that both

   a) Finish [i] = false

   b) $Need_i \leq work$

      If no such i exists goto step (4)

3) Work = Work + $Allocation_i$

   Finish [i] = true

   goto step (2)

4) If Finish [i] = true for all i,

   then the system is in safe state.

Resource-Request Algorithm

Let $Request_i$ be the request array for process P_i. $Request_i[j]$ = k means process P_i wants k instances of resource type R_j. When a request for resources is made by process P_i, the following actions are taken:

1) If $Request_i \leq Need_i$

   Goto step (2) ; otherwise, raise an error condition, since the process has exceeded its maximum claim.

2) If $Request_i \leq Available$

   Goto step (3); otherwise, P_i must wait, since the resources are not available.

3) Have the system pretend to have allocated the requested resources to process P_i by modifying the state follows:

   $Available = Available - Request_i$

   $Allocation_i = Allocation_i + Request_i$

   $Need_i = Need_i - Request_i$

Example:

Considering a system with five processes P_0 through P_4 and three resources types A, B, C. Resource type A has 10 instances, B has 5 instances and type C has 7 instances. Suppose at time t_0 following snapshot of the system has been taken:

Process	Allocation	Max	Available
	A B C	A B C	A B C
P_0	0 1 0	7 5 3	3 3 2
P_1	2 0 0	3 2 2	
P_2	3 0 2	9 0 2	
P_3	2 1 1	2 2 2	
P_4	0 0 2	4 3 3	

Q3. Write short notes on page replacement algorithms?

In a operating systems that use paging for memory management, page replacement algorithm are needed to decide which page needed to be replaced when new page comes in. Whenever a new page is referred and not present in memory, page fault occurs and Operating System replaces one of the existing pages with newly needed page. Different page replacement algorithms suggest different ways to decide which page to replace. The target for all algorithms is to reduce number of page faults.

Page Fault – A page fault is a type of interrupt, raised by the hardware when a running program accesses a memory page that is mapped into the virtual address space, but not loaded in physical memory.

Page Replacement Algorithms

1.First In First Out (FIFO):
This is the simplest page replacement algorithm. In this algorithm, operating system keeps track of all pages in the memory in a queue, oldest page is in the front of the queue. When a page needs to be replaced page in the front of the queue is selected for removal.

For example-1, consider page reference string 1, 3, 0, 3, 5, 6 and 3 page slots.
Initially all slots are empty, so when 1, 3, 0 came they are allocated to the empty slots —> 3 Page Faults. when 3 comes, it is already in memory so —> 0 Page Faults. Then 5 comes, it is not available in memory so it replaces the oldest page slot i.e 1. —>1 Page Fault. Finally 6 comes, it is also not available in memory so it replaces the oldest page slot i.e 3 —>1 Page Fault.

Example-2, Let's have a reference string: a, b, c, d, c, a, d, b, e, b, a, b, c, d and the size of the frame be 4. There are 9 page faults using FIFO algorithm.

Belady's anomaly – Belady's anomaly proves that it is possible to have more page faults when increasing the number of page frames while using the First in First Out (FIFO) page replacement algorithm. For example, if we consider reference string 3, 2, 1, 0, 3, 2, 4, 3, 2, 1, 0, 4 and 3 slots, we get 9 total page faults, but if we increase slots to 4, we get 10 page faults.

2.Optimal Page replacement :
In this algorithm, pages are replaced which are not used for the longest duration of time in the future. Optimal page replacement is perfect, but not possible in practice as operating system cannot know future requests. The use of Optimal Page replacement is to set up a benchmark so that other replacement algorithms can be analyzed against it.

3.Least Recently Used:

In this algorithm page will be replaced which is least recently used.

Let say the page reference string 7 0 1 2 0 3 0 4 2 3 0 3 2 . Initially we have 4 page slots empty.
Initially all slots are empty, so when 7 0 1 2 are allocated to the empty slots —> **4 Page faults**
0 is already their so —> **0 Page fault.**
when 3 came it will take the place of 7 because it is least recently used —>**1 Page fault**
0 is already in memory so —> **0 Page fault**.
4 will takes place of 1 —> **1 Page Fault**
Now for the further page reference string —> **0 Page fault** because they are already available in the memory.

Example-2, Let's have a reference string: a, b, c, d, c, a, d, b, e, b, a, b, c, d and the size of the frame be 4.

Time req. / Page frames		1	2	3	4	5	6	7	8	9	10	11	12	13	14
		a	b	c	d	c	a	d	b	e	b	a	b	c	d
0	a	**a**	a	a	a	a	a	a	a	a	e	e	e	e	e
1	b		**b**	b	b	b	b	b	b	b	b	a	a	a	a
2	c			**c**	c	c	c	c	c	**e**	c	c	b	b	**d**
3	d				**d**	d	d	d	d	d	d	d	d	**c**	c
FAULTS		**x**	**x**	**x**	**x**					**x**				**x**	**x**

Q4. Explain the following terms of memory management?

a. Fragmentation

As processes are loaded and removed from memory, the free memory space is broken into little pieces. External fragmentation exists when there is enough total memory space to satisfy a request but the available spaces are not contiguous; storage is fragmented into a large number of small holes. This fragmentation problem can be severe. In the worst case, we could have a block of free (or wasted) memory between every two processes. If all these small pieces of memory were in one big free block instead, we might be able to run several more processes. Whether we are using the first-fit or best-fit strategy can affect the amount of fragmentation. (First fit is better for some systems, whereas best fit is better for others.) Another factor is which end of a free block is allocated. (Which is the leftover piece-the one on the top or the one on the bottom?) No matter which algorithm is used, however, external fragmentation will be a problem.

Depending on the total amount of memory storage and the average process size, external fragmentation may be a minor or a major problem. Statistical analysis of first fit, for instance, reveals that, even with some optimization, given *N* allocated blocks, another 0.5 *N* blocks will be lost to fragmentation. That is, one-third of memory may be unusable! This property is known as the 50-percent rule.

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

b. Paging

Paging is a memory-management scheme that permits the physical address space of a process to be noncontiguous. Paging avoids external fragmentation and the need for compaction. It also solves the considerable problem of fitting memory chunks of varying sizes onto the backing

store; most memory management schemes used before the introduction of paging suffered from this problem. The problem arises because, when some code fragments or data residing in main memory need to be swapped out, space must be found on the backing store. The backing store has the same fragmentation problems discussed in connection with main memory, but access is much slower, so compaction is impossible. Because of its advantages over earlier methods, paging in its various forms is used in most operating systems. Traditionally, support for paging has been handled by hardware. However, recent designs have implemented paging by closely integrating the hardware and operating system, especially on 64-bit microprocessors.