UNIT 4 DEADLOCKS

Structure

4.1 INTRODUCTION

In a computer system, we have a finite number of *resources* to be distributed among a number of competing processes. These system resources are classified in several types which may be either physical or logical. Examples of physical resources are Printers, Tape drivers, Memory space, and CPU cycles. Examples of logical resources are Files, Semaphores and Monitors. Each resource type can have some identical instances.

A process must request a resource before using it and release the resource after using it. It is clear that the number of resources requested cannot exceed the total number of resources available in the system.

In a normal operation, a process may utilize a resource only in the following sequence:

- *Request*: if the request cannot be immediately granted, then the requesting process must wait until it can get the resource.
- *Use:* the requesting process can operate on the resource.
- *Release:* the process releases the resource after using it.

Examples for request and release of system resources are:

- Request and release the device,
- Opening and closing file,
- Allocating and freeing the memory.

The operating system is responsible for making sure that the requesting process has been allocated the resource. A system table indicates if each resource is free or allocated, and if allocated, to which process. If a process requests a resource that is currently allocated to another process, it can be added to a queue of processes waiting for this resource.

In some cases, several processes may compete for a fixed number of resources. A process requests resources and if the resources are not available at that time, it enters a wait state. It may happen that it will never gain access to the resources, since those resources are being held by other waiting processes.

For example, assume a system with one tape drive and one plotter. Process P1 requests the tape drive and process P2 requests the plotter. Both requests are granted. Now PI requests the plotter (without giving up the tape drive) and P2 requests the tape drive (without giving up the plotter). Neither request can be granted so both processes enter a situation called the deadlock situation.

A deadlock is a situation where a group of processes is permanently blocked as a result of each process having acquired a set of resources needed for its completion and having to wait for the release of the remaining resources held by others thus making it impossible for any of the deadlocked processes to proceed.

In the earlier units, we have gone through the concept of process and the need for the interprocess communication and synchronization. In this unit we will study about the deadlocks, its characterisation, deadlock avoidance and its recovery.

4.2 OBJECTIVES

After going through this unit, you should be able to:

- define a deadlock;
- understand the conditions for a deadlock;
- know the ways of avoiding deadlocks, and
- describe the ways to recover from the deadlock situation.

4.3 DEADLOCKS

Before studying about deadlocks, let us look at the various types of resources. There are two types of resources namely: Pre-emptable and Non-pre-emptable Resources.

- **Pre-emptable resources**: This resource can be taken away from the process with no ill effects. Memory is an example of a pre-emptable resource.
- **Non-Preemptable resource**: This resource cannot be taken away from the process (without causing ill effect). For example, CD resources are not preemptable at an arbitrary moment.

Reallocating resources can resolve deadlocks that involve preemptable resources. Deadlocks that involve nonpreemptable resources are difficult to deal with. Let us see how a deadlock occurs.

Definition: A set of processes is in a deadlock state if each process in the set is waiting for an event that can be caused by only another process in the set. In other words, each member of the set of deadlock processes is waiting for a resource that can be released only by a deadlock process. None of the processes can run, none of them can release any resources and none of them can be awakened. It is important to note that the number of processes and the number and kind of resources possessed and requested are unimportant.

Let us understand the deadlock situation with the help of examples.

Example 1: The simplest example of deadlock is where process 1 has been allocated a non-shareable resource *A*, say, a tap drive, and process 2 has been allocated a non-sharable resource *B*, say, a printer. Now, if it turns out that process 1 needs resource *B* (printer) to proceed and process 2 needs resource *A* (the tape drive) to proceed and these are the only two processes in the system, each has blocked the other and all useful work in the system stops. This situation is termed as deadlock.

The system is in deadlock state because each process holds a resource being requested by the other process and neither process is willing to release the resource it holds.

Example 2: Consider a system with three disk drives. Suppose there are three processes, each is holding one of these three disk drives. If each process now requests another disk drive, three processes will be in a deadlock state, because each process is waiting for the event "disk drive is released", which can only be caused by one of the other waiting processes. Deadlock state involves processes competing not only for the same resource type, but also for different resource types.

Deadlocks occur most commonly in multitasking and client/server environments and are also known as a "Deadly Embrace". Ideally, the programs that are deadlocked or the operating system should resolve the deadlock, but this doesn't always happen.

From the above examples, we have understood the concept of deadlocks. In the examples we were given some instances, but we will study the necessary conditions for a deadlock to occur, in the next section.

4.4 CHARACTERIZATION OF A DEADLOCK

Coffman (1971) identified **four necessary conditions** that must hold simultaneously for a deadlock to occur.

4.4.1 Mutual Exclusion Condition

The resources involved are non-shareable. At least one resource must be held in a non-shareable mode, that is, only one process at a time claims exclusive control of the resource. If another process requests that resource, the requesting process must be delayed until the resource has been released.

4.4.2 Hold and Wait Condition

In this condition, a requesting process already holds resources and waiting for the requested resources. A process, holding a resource allocated to it waits for an additional resource(s) that is/are currently being held by other processes.

4.4.3 No-Preemptive Condition

Resources already allocated to a process cannot be preempted. Resources cannot be removed forcibly from the processes. After completion, they will be released voluntarily by the process holding it.

4.4.4 Circular Wait Condition

The processes in the system form a circular list or chain where each process in the list is waiting for a resource held by the next process in the list.

Let us understand this by a common example. Consider the traffic deadlock shown in the *Figure 1*.

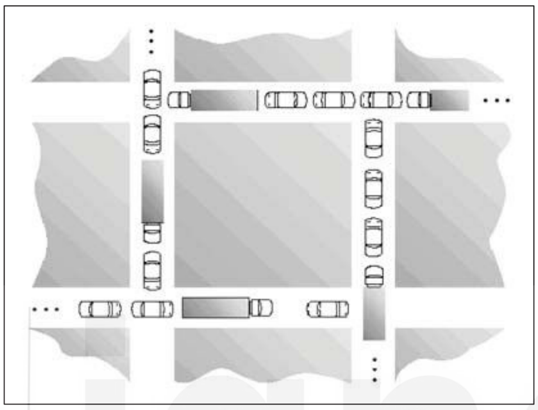

Figure 1: Traffic Deadlock

Consider each section of the street as a resource. In this situation:

- Mutual exclusion condition applies, since only one vehicle can be on a section of the street at a time.
- Hold-and-wait condition applies, since each vehicle is occupying a section of the street, and waiting to move on to the next section of the street.
- Non-preemptive condition applies, since a section of the street that is occupied by a vehicle cannot be taken away from it.
- Circular wait condition applies, since each vehicle is waiting for the next vehicle to move. That is, each vehicle in the traffic is waiting for a section of the street held by the next vehicle in the traffic.

The simple rule to avoid traffic deadlock is that a vehicle should only enter an intersection if it is assured that it will not have to stop inside the intersection.

It is not possible to have a deadlock involving only one single process. The deadlock involves a circular “hold-and-wait” condition between two or more processes, so “one” process cannot hold a resource, yet be waiting for another resource that it is holding. In addition, deadlock is not possible between two threads in a process, because it is the process that holds resources, not the thread, that is, each thread has access to the resources held by the process.

4.4 RESOURCE ALLOCATION GRAPH

The idea behind the resource allocation graph is to have a graph which has two different types of nodes, the process nodes and resource nodes (process represented by circles, resource node represented by rectangles). For different instances of a resource, there is a dot in the resource node rectangle. For example, if there are two identical printers, the printer resource might have two dots to indicate that we don't really care which is used, as long as we acquire the resource.

The edges among these nodes represent resource allocation and release. Edges are directed, and if the edge goes from resource to process node that means the process has acquired the resource. If the edge goes from process node to resource node that means the process has requested the resource.

We can use these graphs to determine if a deadline has occurred or may occur. If for example, all resources have only one instance (all resource node rectangles have one dot) and the graph is circular, then a deadlock *has* occurred. If on the other hand some resources have several instances, then a deadlock *may* occur. If the graph is not circular, a deadlock cannot occur (the *circular wait* condition wouldn't be satisfied).

The following are the tips which will help you to check the graph easily to predict the presence of cycles.

- If no cycle exists in the resource allocation graph, there is no deadlock.
- If there is a cycle in the graph and each resource has only one instance, then there is a deadlock. In this case, a cycle is a necessary and sufficient condition for deadlock.
- If there is a cycle in the graph, and each resource has more than one instance, there may or may not be a deadlock. (A cycle may be broken if some process outside the cycle has a resource instance that can break the cycle). Therefore, a cycle in the resource allocation graph is a necessary but not sufficient condition for deadlock, when multiple resource instances are considered.

Example:

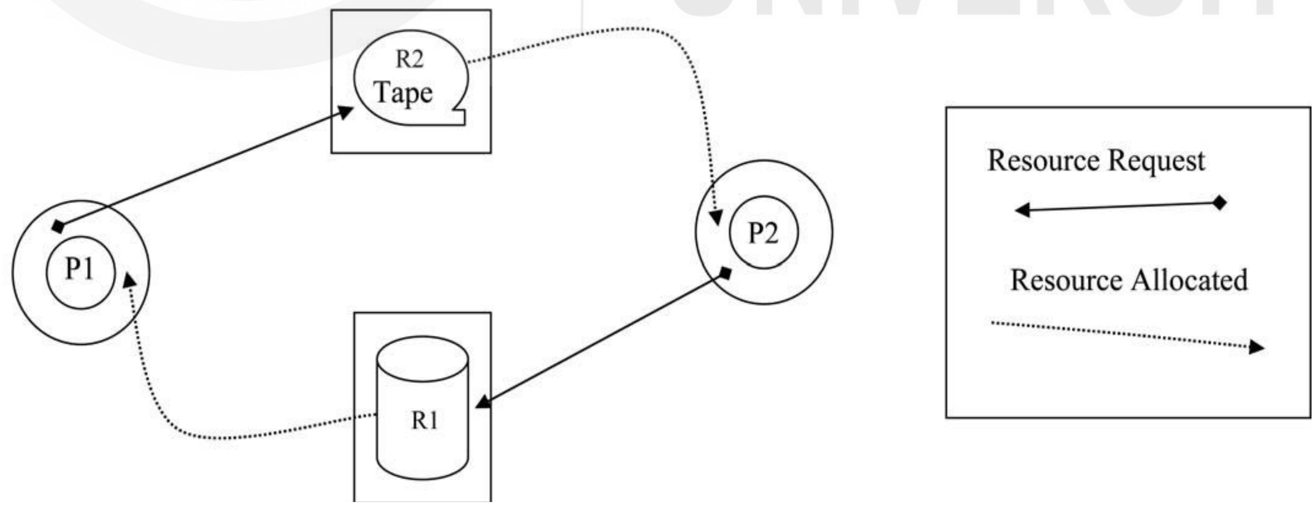

Figure 2: Resource Allocation Graph Showing Deadlock

The above graph shown in *Figure 2* has a cycle and is in Deadlock.

$R1 \longrightarrow P1 \quad P1 \longrightarrow R2$
$R2 \longrightarrow P2 \quad P2 \longrightarrow R1$

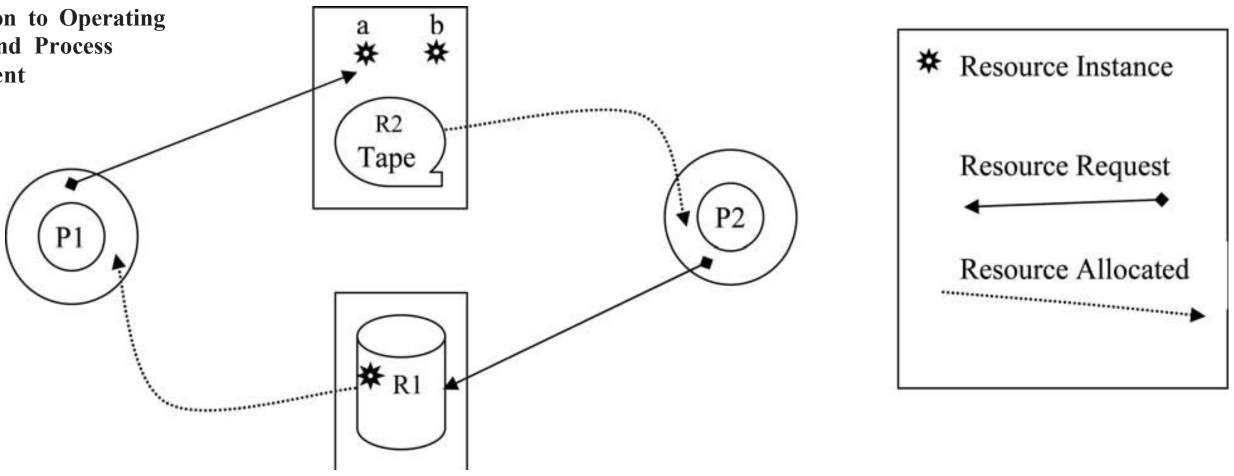

Figure 3: Resource Allocation Graph having a cycle and not in a Deadlock

The above graph shown in *Figure 3* has a cycle and is not in Deadlock.

(Resource 1 has one instance shown by a star)

(Resource 2 has two instances a and b, shown as two stars)

R1 ⟶ P1 P1 ⟶ R2 **(a)**

R2 **(b)** ⟶ P2 P2 ⟶ R1

If P1 finishes, P2 can get R1 and finish, so there is no Deadlock.

4.5 DEALING WITH DEADLOCK SITUATIONS

There are possible strategies to deal with deadlocks. They are:

- Deadlock Prevention
- Deadlock Avoidance
- Deadlock Detection and Recovery

Let's examine each strategy one by one to evaluate their respective strengths and weaknesses.

4.5.1 Deadlock Prevention

Havender in his pioneering work showed that since all four of the conditions are necessary for deadlock to occur, it follows that deadlock might be prevented by denying any one of the conditions. Let us study Havender's algorithm.

Havender's Algorithm

Elimination of "Mutual Exclusion" Condition

The mutual exclusion condition must hold for non-shareable resources. That is, several processes cannot simultaneously share a single resource. This condition is difficult to eliminate because some resources, such as the tap drive and printer, are inherently non-shareable. Note that shareable resources like read-only-file do not require mutually exclusive access and thus cannot be involved in deadlock.

Elimination of "Hold and Wait" Condition

There are two possibilities for the elimination of the second condition. The first alternative is that a process request be granted all the resources it needs at once, prior to execution. The second alternative is to disallow a process from requesting resources whenever it has previously allocated resources. This strategy requires that all the resources a process will need must be requested at once. The system must grant resources on "all or none" basis. If the complete set of resources needed by a process is not currently available, then the process must wait until the complete set is available. While the process waits, however, it may not hold any resources. Thus the "wait for" condition is denied and deadlocks simply cannot occur. This strategy can lead to serious waste of resources.

For example, a program requiring ten tap drives must request and receive all ten drives before it begins executing. If the program needs only one tap drive to begin execution and then does not need the remaining tap drives for several hours then substantial computer resources (9 tape drives) will sit idle for several hours. This strategy can cause indefinite postponement (starvation), since not all the required resources may become available at once.

Elimination of "No-preemption" Condition

The nonpreemption condition can be alleviated by forcing a process waiting for a resource that cannot immediately be allocated, to relinquish all of its currently held resources, so that other processes may use them to finish their needs. Suppose a system does allow processes to hold resources while requesting additional resources. Consider what happens when a request cannot be satisfied. A process holds resources a second process may need in order to proceed, while the second process may hold the resources needed by the first process. This is a deadlock. This strategy requires that when a process that is holding some resources is denied a request for additional resources, the process must release its held resources and, if necessary, request them again together with additional resources. Implementation of this strategy denies the "no-preemptive" condition effectively.

High Cost

When a process releases resources, the process may lose all its work to that point. One serious consequence of this strategy is the possibility of indefinite postponement (starvation). A process might be held off indefinitely as it repeatedly requests and releases the same resources.

Elimination of "Circular Wait" Condition

The last condition, the circular wait, can be denied by imposing a total ordering on all of the resource types and than forcing all processes to request the resources in order (increasing or decreasing). This strategy imposes a total ordering of all resource types, and requires that each process requests resources in a numerical order of enumeration. With this rule, the resource allocation graph can never have a cycle.

For example, provide a global numbering of all the resources, as shown in the given *Table 1*:

Table 1: Numbering the resources

Number	Resource
1	Floppy drive
2	Printer
3	Plotter
4	Tape Drive
5	CD Drive

Now we will see the rule for this:

Rule: Processes can request resources whenever they want to, but all requests must be made in numerical order. A process may request first printer and then a tape drive (order: 2, 4), but it may not request first a plotter and then a printer (order: 3, 2). The problem with this strategy is that it may be impossible to find an ordering that satisfies everyone.

This strategy, if adopted, may result in low resource utilization and in some cases starvation is possible too.

4.5.2 Deadlock Avoidance

This approach to the deadlock problem anticipates a deadlock before it actually occurs. This approach employs an algorithm to access the possibility that deadlock could occur and act accordingly. This method differs from deadlock prevention, which guarantees that deadlock cannot occur by denying one of the necessary conditions of deadlock. The most famous deadlock avoidance algorithm, from Dijkstra [1965], is the Banker's algorithm. It is named as Banker's algorithm because the process is analogous to that used by a banker in deciding if a loan can be safely made a not.

The Banker's Algorithm is based on the banking system, which never allocates its available cash in such a manner that it can no longer satisfy the needs of all its customers. Here we must have the advance knowledge of the maximum possible claims for each process, which is limited by the resource availability. During the run of the system we should keep monitoring the resource allocation status to ensure that no circular wait condition can exist.

If the necessary conditions for a deadlock are in place, it is still possible to avoid deadlock by being careful when resources are allocated. The following are the features that are to be considered for avoidance of the deadlock s per the Banker's Algorithms.

- Each process declares maximum number of resources of each type that it may need.
- Keep the system in a safe state in which we can allocate resources to each process in some order and avoid deadlock.
- Check for the safe state by finding a safe sequence: <P1, P2, ..., Pn> where resources that Pi needs can be satisfied by available resources plus resources held by Pj where j < i.

- Resource allocation graph algorithm uses claim edges to check for a safe state.

The resource allocation state is now defined by the number of available and allocated resources, and the maximum demands of the processes. Subsequently the system can be in either of the following states:

- **Safe state:** Such a state occur when the system can allocate resources to each process (up to its maximum) in some order and avoid a deadlock. This state will be characterised by a safe sequence. It must be mentioned here that we should not falsely conclude that all unsafe states are deadlocked although it may eventually lead to a deadlock.
- **Unsafe State:** If the system did not follow the safe sequence of resource allocation from the beginning and it is now in a situation, which may lead to a deadlock, then it is in an unsafe state.
- **Deadlock State:** If the system has some circular wait condition existing for some processes, then it is in deadlock state.

Let us study this concept with the help of an example as shown below:

Consider an analogy in which 4 processes (P1, P2, P3 and P4) can be compared with the customers in a bank, resources such as printers etc. as cash available in the bank and the Operating system as the Banker.

Table 2

Processes	Resources used	Maximum resources
P1	0	6
P2	0	5
P3	0	4
P4	0	7

Let us assume that total available resources = 10

In the above table, we see four processes, each of which has been granted a number of maximum resources that can be used. The Operating system reserved only 10 resources rather than 22 units to service them. At a certain moment, the situation becomes:

Table 3

Processes	Resources used	Maximum resources
P1	1	6
P2	1	5
P3	2	4
P4	4	7

Available resources = 2

Safe State: The key to a state being safe is that there is at least one way for all users to finish. In other words the state of *Table 2* is safe because with 2 units left, the operating system can delay any request except P3, thus letting P3 finish and release all four resources. With four units in hand, the Operating system can let either P4 or P2 have the necessary units and so on.

Unsafe State: Consider what would happen if a request from P2 for one more unit was granted in *Table 3*. We would have following situation as shown in *Table 4*.

Table 4

Processes	**Resources used**	**Maximum resources**
P1	1	6
P2	2	5
P3	2	4
P4	4	7

Available resource = 1

This is an unsafe state.

If all the processes request for their maximum resources respectively, then the operating system could not satisfy any of them and we would have a deadlock.

Important Note: It is important to note that an unsafe state does not imply the existence or even the eventual existence of a deadlock. What an unsafe state does imply is simply that some unfortunate sequence of events might lead to a deadlock.

The Banker's algorithm is thus used to consider each request as it occurs, and see if granting it leads to a safe state. If it does, the request is granted, otherwise, it is postponed until later. Haberman [1969] has shown that executing of the algorithm has a complexity proportional to N^2 where N is the number of processes and since the algorithm is executed each time a resource request occurs, the overhead is significant.

Limitations of the Banker's Algorithm

There are some problems with the Banker's algorithm as given below:

- It is time consuming to execute on the operation of every resource.
- If the claim information is not accurate, system resources may be underutilized.
- Another difficulty can occur when a system is heavily loaded. Lauesen states that in this situation "so many resources are granted away that very few safe sequences remain, and as a consequence, the jobs will be executed sequentially". Therefore, the Banker's algorithm is referred to as the "Most Liberal" granting policy; that is, it gives away everything that it can without regard to the consequences.
- New processes arriving may cause a problem.
  – The process's claim must be less than the total number of units of the resource in the system. If not, the process is not accepted by the manager.

  - Since the state without the new process is safe, so is the state with the new process. Just use the order you had originally and put the new process at the end.
  - Ensuring fairness (starvation freedom) needs a little more work, but isn't too hard either (once every hour stop taking new processes until all current processes finish).
- A resource becoming unavailable (e.g., a tape drive breaking), can result in an unsafe state.

4.5.3 Deadlock Detection and Recovery

Detection of deadlocks is the most practical policy, which being both liberal and cost efficient, most operating systems deploy. To detect a deadlock, we must go about in a recursive manner and simulate the most favoured execution of each unblocked process.

- An unblocked process may acquire all the needed resources and will execute.
- It will then release all the acquired resources and remain dormant thereafter.
- The now released resources may wake up some previously blocked process.
- Continue the above steps as long as possible.
- If any blocked processes remain, they are **deadlocked.**

Recovery from Deadlock

Recovery by process termination

In this approach we terminate deadlocked processes in a systematic way taking into account their priorities. The moment, enough processes are terminated to recover from deadlock, we stop the process terminations. Though the policy is simple, there are some problems associated with it.

Consider the scenario where a process is in the state of updating a data file and it is terminated. The file may be left in an incorrect state by the unexpected termination of the updating process. Further, processes should be terminated based on some criterion/ policy. Some of the criteria may be as follows:

- Priority of a process
- CPU time used and expected usage before completion
- Number and type of resources being used (can they be preempted easily?)
- Number of resources needed for completion
- Number of processes needed to be terminated
- Are the processes interactive or batch?

Recovery by Checkpointing and Rollback (Resource preemption)

Some systems facilitate deadlock recovery by implementing *checkpointing and rollback.* Checkpointing is saving *enough state of a process* so that the process can be restarted at the point in the computation where the checkpoint was taken.Autosaving file edits are a form of checkpointing. Checkpointing costs depend on the underlying algorithm. Very simple algorithms (like linear primality testing) can be checkpointed with a few words of data. More complicated processes may have to save all the process state and memory.

If a deadlock is detected, one or more processes are restarted from their last checkpoint. Restarting a process from a checkpoint is called *rollback*. It is done with the expectation that the resource requests will not interleave again to produce deadlock.

Deadlock recovery is generally used when deadlocks are rare, and the cost of recovery (process termination or rollback) is low.

Process checkpointing can also be used to improve reliability (long running computations), assist in process migration, or reduce startup costs.

☞ **Check Your Progress 1**

1) What is a deadlock and what are the four conditions that will create the deadlock situation?

....

....

....

....

2) How can deadlock be avoided? Explain with the help of an example.

....

....

....

....

4.6 SUMMARY

A deadlock occurs a process has some resource and is waiting to acquire another resource, while that resource is being held by some process that is waiting to acquire the resource that is being held by the first process.

A deadlock needs four conditions to occur: Mutual Exclusion, Hold and Wait, Non-Preemption and Circular Waiting.

We can handle deadlocks in three major ways: We can prevent them, handle them when we detect them, or simply ignore the whole deadlock issue altogether.

4.7 SOLUTIONS/ANSWERS

Check Your Progress 1

1) A set of processes is in a deadlock state when every process in the set is waiting for an event that can only be caused by another process in the set. 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-shareable mode; that is, only one process at a time can use the resource. If another process requests the resource, the requesting process must be delayed until the resource has been released.

- *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, P1, P2, …, Pn} of waiting processes must exist such that P0 is waiting for a resource that is held by P1, P1 is waiting for a resource that is held by P2, …, Pn-1 is waiting for a resource that is held by Pn, and Pn is waiting for a resource that is held by P0.

2) Deadlock avoidance deals with processes that declare before execution how many resources they may need during their execution. Given several processes and resources, if we can allocate the resources in some order as to prevent a deadlock, the system is said to be in a safe state. If a deadlock is possible, the system is said to be in an unsafe state. The idea of avoiding a deadlock is to simply not allow the system to enter an unsafe state which may cause a deadlock. We can define what makes an unsafe state.

   For example, consider a system with 12 tape drives and three processes: P0, P1 and P2. P0 may require 10 tape drives during execution, P1 may require 4, and P2 may require up to 9. Suppose that at some moment in time, P0 is holding on to 5 tape drives, P1 holds 2 and P2 holds 2 tape drives. The system is said to be in a safe state, since there is a safe sequence that avoids the deadlock. This sequence implies that P1 can instantly get all of its needed resources (there are 12 total tape drives, and P1 already has 2, so the maximum it needs is 2, which it can get since there are 3 free tape drives). Once it finishes executing, it releases all 4 resources, which makes for 5 free tape drives, at which point, P0 can execute (since the maximum it needs is 10), after it finishes, P2 can proceed since the maximum it needs is 9.

   Now, here's an unsafe sequence: Let's say at some point of time P2 requests one more resource to make its holding resources 3. Now the system is in an unsafe state, since only P1 can be allocated resources, and after it returns, it will only leave 4 free resources, which is not enough for either P0 or P2, so the system enters a deadlock.

4.8 FURTHER READINGS

1) Madnick and Donovan, *Operating systems – Concepts and Design,* McGrawHill Intl. Education.

2) Milan Milenkovic, *Operating Systems, Concepts and Design*, TMGH, 2000.

3) D.M. Dhamdhere, *Operating Systems, A concept-based approach*, TMGH, 2002.

4) Abraham Silberschatz and James L, *Operating System Concepts*. Peterson, Addition Wesely Publishing Company, New Delhi.

5) Harvay M. Deital, *Introduction to Operating systems,* Addition Wesely Publishing Company, New Delhi.

6) Andrew S. Tanenbaum, *Operating System Design and Implementation,* PHI, New Delhi.