Unit 6 Deadlocks

Structure:

6.1 Introduction
Objectives
6.2 System Model
6.3 Deadlock Characterization
Necessary conditions for deadlock
Resource-allocation graph
6.4 Deadlock Handling
6.5 Deadlock Prevention
6.6 Deadlock Avoidance
Safe state
Resource-allocation graph algorithm
Banker's algorithm
6.7 Deadlock Detection
Single instance of a resource
Multiple instances of a resource
Recovery from deadlock
6.8 Summary
6.9 Terminal Questions
6.10 Answers

6.1 Introduction

In the previous unit, you have learned about process synchronization. In this unit, let's study about deadlocks.

Several processes compete for a finite set of resources in a multi-programmed environment. A process requests for resources that may not be readily available at the time of the request. In such a case the process goes into a wait state. It may so happen that this process may never change state because the requested resources are held by other processes which themselves are waiting for additional resources and hence in a wait state. This situation is called a deadlock.

Deadlock occurs when we have a set of processes [not necessarily all the processes in the system], each holding some resources, each requesting some resources, and none of them is able to obtain what it needs, i.e. to

make progress. We will usually reason in terms of resources R1, R2... Rm and processes P1, P2... Pn. A process Pi that is waiting for some currently unavailable resource is said to be **blocked**.

Resources can be **pre-emptable** or **non-pre-emptable**. A resource is pre-emptable if it can be taken away from the process that is holding it [we can think that the original holder waits, frozen, until the resource is returned to it]. Memory is an example of a pre-emptable resource. Of course, one may choose to deal with intrinsically pre-emptable resources as if they were non-pre-emptable. In our discussion we only consider non-pre-emptable resources.

Resources can be **reusable** or **consumable**. They are reusable if they can be used again after a process is done using them. Memory, printers, tape drives are examples of reusable resources. Consumable resources are resources that can be used only once, for example a message, a signal or an event. If two processes are waiting for a message and one receives it, then the other process remains waiting. To reason about deadlocks when dealing with consumable resources is extremely difficult. Thus we will restrict our discussion to reusable resources.

Objectives:

After studying this unit, you should be able to:

- explain the system model
- describe the characteristics of deadlocks
- discuss handling, preventing, avoiding and detecting deadlocks

6.2 System Model

The number of resources in a system is always finite. But the competing processes are many. Resources are of several types, each type having identical instances of the resource. Examples for resources could be memory space, CPU time, files, I/O devices and so on. If a system has 2 CPUs that are equivalent, then the resource type CPU time has 2 instances. If they are not equivalent, then each CPU is of a different resource type. Similarly the system may have 2 dot matrix printers and 1 line printer. Here the resource type of dot matrix printer has 2 instances whereas there is a single instance of type line printer.

A process requests for resources, uses them if granted and then releases the resources for others to use. It goes without saying that the number of resources requested shall not exceed the total of each type available in the system. If a request for a resource cannot be granted immediately then the process requesting the resource goes into a wait state and joins the wait queue for the resource.

A set of processes is in a state of deadlock if every process in the set is in some wait queue of a resource and is waiting for an event (release resource) to occur that can be caused by another process in the set. For example, there are 2 resources, 1 printer and 1 tape drive. Process P1 is allocated tape drive and P2 is allocated printer. Now if P1 requests for printer and P2 for tape drive, a deadlock occurs.

6.3 Deadlock Characterization

Let's discuss characteristics of deadlocks.

6.3.1 Necessary Conditions for Deadlock

A deadlock occurs in a system if the following four conditions hold simultaneously:

1) *Mutual exclusion*: At least one of the resources is non-sharable, that is, only one process at a time can use the resource.
2) *Hold and wait*: A process exists that is holding on to at least one resource and waiting for an additional resource held by another process.
3) *No preemption*: Resources cannot be preempted, that is, a resource is released only by the process that is holding it.
4) *Circular wait*: There exist a set of processes P_0, P_1… P_n of waiting processes such that P_0 is waiting for a resource held by P_1, P_1 is waiting for a resource held by P_2… P_{n-1} is waiting for a resource held P_n and P_n is in turn waiting for a resource held by P_0.

6.3.2 Resource-allocation graph

Deadlocks can be described by a resource allocation graph. The resource allocation graph is a directed graph consisting of vertices and directed edges. The vertex set is partitioned into two types, a subset representing processes and another subset representing resources. Pictorially, the resources are represented by rectangles with dots within, each dot representing an instance of the resource and circles represent processes.

A directed edge from a process to a resource ($P_i \rightarrow R_j$) signifies a request from a process P_i for an instance of the resource R_j and P_i is waiting for R_j. A directed edge from a resource to a process ($R_j \rightarrow P_i$) indicates that an instance of the resource R_j has been allotted to process P_i. Thus a resource allocation graph consists of vertices which include resources and processes and directed edges which consist of request edges and assignment edges. A request edge is introduced into the graph when a process requests for a resource. This edge is converted into an assignment edge when the resource is granted. When the process releases the resource, the assignment edge is deleted.

Consider the following system:

There are 3 processes P_1, P_2 and P_3.

Resources R_1, R_2, R_3 and R_4 have instances 1, 2, 1, and 3 respectively.

P_1 is holding R_2 and waiting for R_1.

P_2 is holding R_1, R_2 and is waiting for R_3.

P_3 is holding R_3.

The resource allocation gragh for a system in the above situation is as shown below (Refer to figure 6.1).

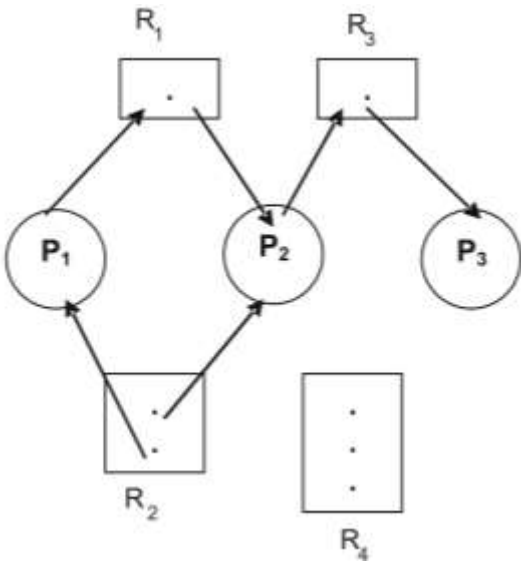

Fig. 6.1: Resource allocation graph

If a resource allocation graph has no cycles (a closed loop in the direction of the edges), then the system is not in a state of deadlock. If on the other hand, there are cycles, then a deadlock may exist. If there are only single instances of each resource type, then a cycle in a resource allocation graph is a necessary and sufficient condition for existence of a deadlock (Refer to figure 6.2). Here two cycles exist:

$P_1 \rightarrow R_1 \rightarrow P_2 \rightarrow R_3 \rightarrow P_3 \rightarrow R_2 \rightarrow P_1$
$P_2 \rightarrow R_3 \rightarrow P_3 \rightarrow R_2 \rightarrow P_2$

Processes P_0, P_1 and P_3 are deadlocked and are in a circular wait. P_2 is waiting for R_3 held by P_3. P_3 is waiting for P_1 or P_2 to release R_2. So also P_1 is waiting for P_2 to release R_1.

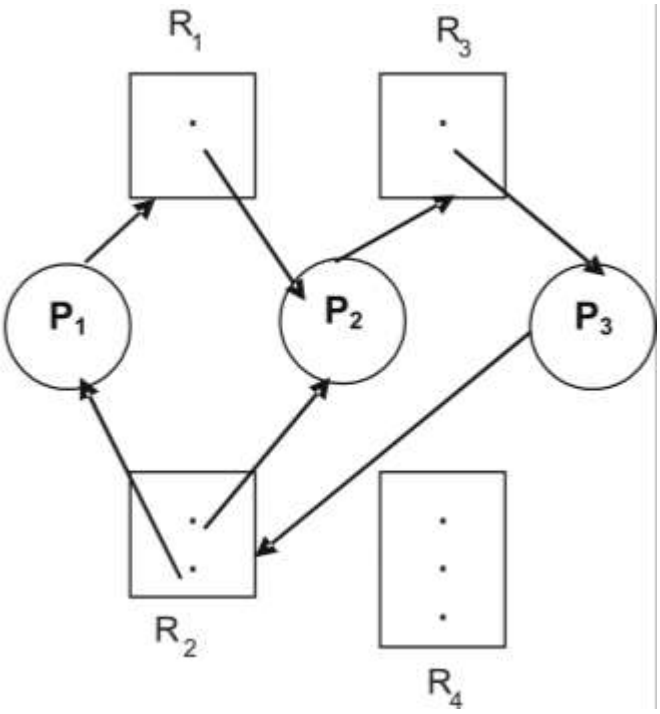

Fig. 6.2: Resource allocation graph with deadlock

If there are multiple instances of resources types, then a cycle does not necessarily imply a deadlock. Here a cycle is a necessary condition but not a sufficient condition for the existence of a deadlock (Refer to figure 6.3). Here also there is a cycle:

$P_1 \rightarrow R_1 \rightarrow P_3 \rightarrow R_2 \rightarrow P_1$

The cycle above does not imply a deadlock because an instance of R_1 released by P_2 could be assigned to P_1 or an instance of R_2 released by P_4 could be assigned to P_3 there by breaking the cycle.

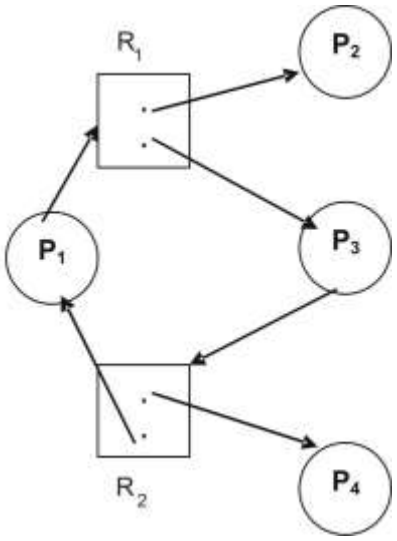

Fig. 6.3: Resource allocation graph with a cycle but no deadlock

Self Assessment Questions

1. Several processes compete for a finite set of resources in a multi-programmed environment. (True / False)
2. A process Pi that is waiting for some currently unavailable resource is said to be _____.
3. A condition where at least one of the resources is non-sharable in a system is called _____. (Pick correct option)
   a) Mutual Exclusion
   b) Hold and Wait
   c) Circular Wait
   d) None of the above

6.4 Deadlock Handling

Different methods to deal with deadlocks include methods to ensure that the system will never enter into a state of deadlock, methods that allow the system to enter into a deadlock and then recover or to just ignore the problem of deadlocks.

To ensure that deadlocks never occur, deadlock prevention / avoidance schemes are used. The four necessary conditions for deadlocks to occur are mutual exclusion, hold and wait, no preemption and circular wait. Deadlock prevention ensures that at least one of the four necessary conditions for deadlocks does not hold. To do this the scheme enforces constraints on requests for resources. Dead-lock avoidance scheme requires the operating system to know in advance, the resources needed by a process for its entire lifetime. Based on this a priori information, the process making a request is either made to wait or not to wait in case the requested resource is not readily available. If none of the above two schemes are used, then deadlocks may occur. In such a case, an algorithm to recover from the state of deadlock is used.

If the problem of deadlocks is ignored totally, that is, to say the system does not ensure that a deadlock does not occur and also does not provide for recovery from deadlock and such a situation arises, then there is no way out of the deadlock. Eventually, the system may crash because more and more processes request for resources and enter into deadlock.

6.5 Deadlock Prevention

The four necessary conditions for deadlocks to occur are mutual exclusion, hold and wait, no preemption and circular wait. If any one of the above four conditions does not hold, then deadlocks will not occur. Thus prevention of deadlock is possible by ensuring that at least one of the four conditions cannot hold.

***Mutual exclusion*:** Resources that can be shared are never involved in a deadlock because such resources can always be granted simultaneous access by processes. Hence processes requesting for such a sharable resource will never have to wait. Examples of such resources include read-only files. Mutual exclusion must therefore hold for non-sharable resources. But it is not always possible to prevent deadlocks by denying mutual exclusion condition because some resources are by nature non-sharable, for example printers.

***Hold and wait*:** To avoid hold and wait, the system must ensure that a process that requests for a resource does not hold on to another. There can be two approaches to this scheme:

1) a process requests for and gets allocated all the resources it uses before execution begins.
2) a process can request for a resource only when it does not hold on to any other.

Algorithms based on these approaches have poor resource utilization. This is because resources get locked with processes much earlier than they are actually used and hence not available for others to use as in the first approach. The second approach seems applicable only when there is assurance about reusability of data and code on the released resources. The algorithms also suffer from starvation since popular resources may never be freely available.

***No preemption*:** This condition states that resources allocated to processes cannot be preempted. To ensure that this condition does not hold, resources could be preempted. When a process requests for a resource, it is allocated the resource if it is available. If it is not available, then a check is made to see if the process holding the wanted resource is also waiting for additional resources. If so, the wanted resource is preempted from the waiting process and allotted to the requesting process. If both the above are not true, that is, the resource is neither available nor held by a waiting process then the requesting process waits. During its waiting period, some of its resources could also be preempted in which case the process will be restarted only when all the new and the preempted resources are allocated to it.

Another alternative approach could be as follows: If a process requests for a resource which is not available immediately, then all other resources it currently holds are preempted. The process restarts only when the new and the preempted resources are allocated to it as in the previous case.

Resources can be preempted only if their current status can be saved so that processes could be restarted later by restoring the previous states. *Example:* CPU memory and main memory. But resources such as printers cannot be preempted, as their states cannot be saved for restoration later.

***Circular wait*:** Resource types need to be ordered and processes requesting for resources will do so in an increasing order of enumeration. Each resource type is mapped to a unique integer that allows resources to be compared and to find out the precedence order for the resources. Thus F: R → N is a 1:1 function that maps resources to numbers. For example:

F (tape drive) = 1, F (disk drive) = 5, F (printer) = 10.

To ensure that deadlocks do not occur, each process can request for resources only in an increasing order of these numbers. A process, to start with in the very first instance can request for any resource say R_i. Thereafter it can request for a resource R_j if and only if $F(R_j)$ is greater than $F(R_i)$. Alternately, if $F(R_j)$ is less than $F(R_i)$, then R_j can be allocated to the process if and only if the process releases R_i.

The mapping function F should be so defined that resources get numbers in the usual order of usage.

6.6 Deadlock Avoidance

Deadlock prevention algorithms ensure that at least one of the four necessary conditions for deadlocks namely mutual exclusion, hold and wait, no preemption and circular wait do not hold. The disadvantage with prevention algorithms is poor resource utilization and thus reduced system throughput.

An alternate method is to avoid deadlocks. In this case additional a priori information about the usage of resources by processes is required. This information helps to decide on whether a process should wait for a resource or not. Decision about a request is based on all the resources available, resources allocated to processes, future requests and releases by processes.

A deadlock avoidance algorithm requires each process to make known in advance the maximum number of resources of each type that it may need. Also known is the maximum number of resources of each type available. Using both the above a priori knowledge, deadlock avoidance algorithm ensures that a circular wait condition never occurs.

6.6.1 Safe state

A system is said to be in a safe state if it can allocate resources upto the maximum available and is not in a state of deadlock. A safe sequence of processes always ensures a safe state. A sequence of processes $< P_1, P_2, ..., P_n >$ is safe for the current allocation of resources to processes if resource requests from each P_i can be satisfied from the currently available resources and the resources held by all P_j where $j < i$. If the state is safe

then P_i requesting for resources can wait till P_j's have completed. If such a safe sequence does not exist, then the system is in an unsafe state.

A safe state is not a deadlock state. Conversely a deadlock state is an unsafe state. But all unsafe states are not deadlock states as in Figure 6.4.

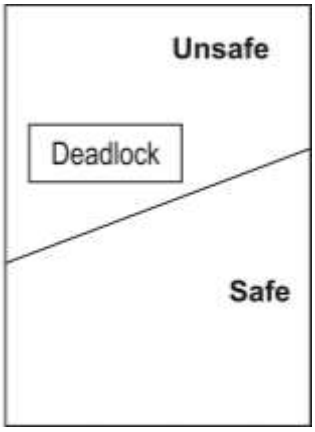

Fig. 6.4: Safe, unsafe and deadlock state spaces

If a system is in a safe state it can stay away from an unsafe state and thus avoid deadlock. On the other hand, if a system is in an unsafe state, deadlocks cannot be avoided.

Illustration: A system has 12 instances of a resource type and 3 processes using these resources. The maximum requirements for the resource by the processes and their current allocation at an instance say t0 is as shown below:

Process	Maximum	Current
P0	10	5
P1	4	2
P3	9	2

At the instant t_0, the system is in a safe state and one safe sequence is < P_1, P_0, P_2 >. This is true because of the following facts:

- Out of 12 instances of the resource, 9 are currently allocated and 3 are free.
- P_1 needs only 2 more, its maximum being 4, can be allotted 2.
- Now only 1 instance of the resource is free.
- When P_1 terminates, 5 instances of the resource will be free.

- P_0 needs only 5 more, its maximum being 10, can be allotted 5.
- Now resource is not free.
- Once P_0 terminates, 10 instances of the resource will be free.
- P_3 needs only 7 more, its maximum being 9, can be allotted 7.
- Now 3 instances of the resource are free.
- When P_3 terminates, all 12 instances of the resource will be free.

Thus the sequence < P_1, P_0, P_3 > is a safe sequence and the system is in a safe state.

Let us now consider the following scenario at an instant t_1. In addition to the allocation shown in the table above, P_2 requests for 1 more instance of the resource and the allocation is made. At the instance t_1, a safe sequence cannot be found as shown below:

- Out of 12 instances of the resource, 10 are currently allocated and 2 are free.
- P_1 needs only 2 more, its maximum being 4, can be allotted 2.
- Now resource is not free.
- Once P_1 terminates, 4 instances of the resource will be free.
- P_0 needs 5 more while P_2 needs 6 more.
- Since both P_0 and P_2 cannot be granted resources, they wait.
- The result is a deadlock.

Thus the system has gone from a safe state at time instant t_0 into an unsafe state at an instant t_1. The extra resource that was granted to P_2 at the instant t_1 was a mistake. P_2 should have waited till other processes finished and released their resources.

Since resources available should not be allocated right away as the system may enter an unsafe state, resource utilization is low if deadlock avoidance algorithms are used.

6.6.2 Resource allocation graph algorithm

A resource allocation graph could be used to avoid deadlocks. If a resource allocation graph does not have a cycle, then the system is not in deadlock. But if there is a cycle then the system may be in a deadlock. If the resource allocation graph shows only resources that have only a single instance, then a cycle does imply a deadlock. An algorithm for avoiding deadlocks where

resources have single instances in a resource allocation graph is as described below.

The resource allocation graph has request edges and assignment edges. Let there be another kind of edge called a claim edge. A directed edge $P_i \rightarrow R_j$ indicates that P_i may request for the resource R_j some time later. In a resource allocation graph a dashed line represents a claim edge. Later when a process makes an actual request for a resource, the corresponding claim edge is converted to a request edge $P_i \rightarrow R_j$. Similarly when a process releases a resource after use, the assignment edge $R_j \rightarrow P_i$ is reconverted to a claim edge $P_i \rightarrow R_j$. Thus a process must be associated with all its claim edges before it starts executing.

If a process Pi requests for a resource R_j, then the claim edge $P_i \rightarrow R_j$ is first converted to a request edge $P_i \rightarrow R_j$. The request of P_i can be granted only if the request edge when converted to an assignment edge does not result in a cycle.

If no cycle exists, the system is in a safe state and requests can be granted. If not the system is in an unsafe state and hence in a deadlock. In such a case, requests should not be granted. This is illustrated in Figure 6.5 a and b.

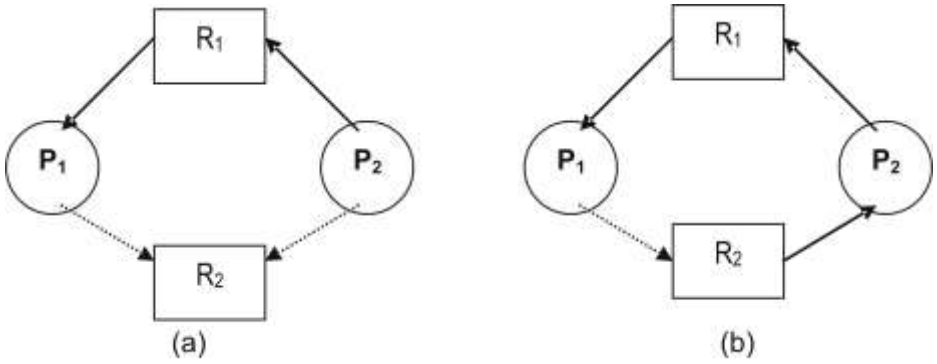

Fig. 6.5: Resource allocation graph showing safe and deadlock states

Consider the resource allocation graph shown on the left above. Resource R_2 is currently free. Allocation of R_2 to P_2 on request will result in a cycle as shown on the right. Therefore the system will be in an unsafe state. In this situation if P_1 requests for R_2, then a deadlock occurs

6.6.3 Banker's algorithm

The resource allocation graph algorithm is not applicable where resources have multiple instances. In such a case Banker's algorithm is used.

A new process entering the system must make known a priori the maximum instances of each resource that it needs subject to the maximum available for each type. As execution proceeds and requests are made, the system checks to see if the allocation of the requested resources ensures a safe state. If so only are the allocations made, else processes must wait for resources.

The following are the data structures maintained to implement the Banker's algorithm:

1) n: Number of processes in the system.
2) m: Number of resource types in the system.
3) Available: is a vector of length m. Each entry in this vector gives maximum instances of a resource type that are available at the instant. Available[j] = k means to say there are k instances of the jth resource type R_j.
4) Max: is a demand vector of size n x m. It defines the maximum needs of each resource by the process. Max[i][j] = k says the ith process P_i can request for atmost k instances of the jth resource type R_j.
5) Allocation: is a n x m vector which at any instant defines the number of resources of each type currently allocated to each of the m processes. If Allocation[i][j] = k then ith process P_i is currently holding k instances of the jth resource type R_j.
6) Need: is also a n x m vector which gives the remaining needs of the processes. Need[i][j] = k means the ith process P_i still needs k more instances of the jth resource type R_j. Thus Need[i][j] = Max[i][j] – Allocation[i][j].

Safety algorithm

Using the above defined data structures, the Banker's algorithm to find out if a system is in a safe state or not is described below:

1. Define a vector Work of length m and a vector Finish of length n.
2. Initialize Work = Available and Finish[i] = false for i = 1, 2, ., n.
3. Find an i such that
   a. Finish[i] = false and

b. $Need_i$ <= Work ($Need_i$ represents the ith row of the vector Need).
If such an i does not exist , go to step 5.

4. Work = Work + $Allocation_i$
Go to step 3.
5. If finish[i] = true for all i, then the system is in a safe state.

Resource-request algorithm

Let $Request_i$ be the vector representing the requests from a process P_i. Requesti[j] = k shows that process P_i wants k instances of the resource type R_j. The following is the algorithm to find out if a request by a process can immediately be granted:

1) If $Request_i$ <= $Need_i$, go to step 2.
else Error "request of P_i exceeds Max_i".

2) If $Request_i$ <= $Available_i$, go to step 3.
else P_i must wait for resources to be released.

3) An assumed allocation is made as follows:
Available = Available – $Request_i$
$Allocation_i$ = $Allocation_i$ + $Request_i$
$Need_i$ = $Need_i$ – $Request_i$

If the resulting state is safe, then process P_i is allocated the resources and the above changes are made permanent. If the new state is unsafe, then P_i must wait and the old status of the data structures is restored.

Illustration: n = 5 < P_0, P_1, P_2, P_3, P_4 >
M = 3 < A, B, C >
Initially Available = < 10, 5, 7 >
At an instant t0, the data structures have the following values:

	Allocation	**Max**	**Available**	**Need**
	A B C	A B C	A B C	A B C
Po	0 1 0	7 5 3	3 3 2	7 4 3
P_1	2 0 0	3 2 2		1 2 2
P_2	3 0 2	9 0 2		6 0 0
P_3	2 1 1	2 2 2		0 1 1
P_4	0 0 2	4 3 3		4 3 1

To find a safe sequence and to prove that the system is in a safe state, use the safety algorithm as follows:

Step	Work	Finish	Safe sequence
0	3 3 2	F F F F F	< >
1	5 3 2	F T F F F	< P_1 >
2	7 4 3	F T F T F	< P_1, P_3 >
3	7 4 5	F T F T T	< P_1, P_3, P_4 >
4	7 5 5	T T F T T	< P_1, P_3, P_4, P_0 >
5	10 5 7	T T T T T	< P_1, P_3, P_4, P_0, P_2 >

Now at an instant t_1, $Request_1$ = < 1, 0, 2 >. To actually allocate the requested resources, use the request-resource algorithm as follows:

$Request_1$ < $Need_1$ and $Request_1$ < Available so the request can be considered. If the request is fulfilled, then the new the values in the data structures are as follows:

	Allocation	**Max**	**Available**	**Need**
	A B C	A B C	A B C	A B C
P0	0 1 0	7 5 3	2 3 0	7 4 3
P_1	3 0 2	3 2 2		0 2 0
P_2	3 0 2	9 0 2		6 0 0
P_3	2 1 1	2 2 2		0 1 1
P_4	0 0 2	4 3 3		4 3 1

Use the safety algorithm to see if the resulting state is safe:

Step	Work	Finish	Safe sequence
0	2 3 0	F F F F F	< >
1	5 3 2	F T F F F	< P_1 >
2	7 4 3	F T F T F	< P_1, P_3 >
3	7 4 5	F T F T T	< P_1, P_3, P_4 >
4	7 5 5	T T F T T	< P_1, P_3, P_4, P_0 >
5	10 5 7	T T T T T	< P_1, P_3, P_4, P_0, P_2 >

Since the resulting state is safe, request by P_1 can be granted.

Now at an instant t_2 $Request_4$ = < 3, 3, 0 >. But since $Request_4$ > Available, the request cannot be granted. Also $Request_0$ = < 0, 2, 0> at t_2 cannot be granted since the resulting state is unsafe as shown below:

	Allocation	Max	Available	Need
	A B C	A B C	A B C	A B C
P0	0 3 0	7 5 3	2 1 0	7 2 3
P_1	3 0 2	3 2 2		0 2 0
P_2	3 0 2	9 0 2		6 0 0
P_3	2 1 1	2 2 2		0 1 1
P_4	0 0 2	4 3 3		4 3 1

Using the safety algorithm, the resulting state is unsafe since Finish is false for all values of i and we cannot find a safe sequence.

Step	Work	Finish	Safe sequence
0	2 1 0	F F F F F	< >

Self Assessment Questions

4. Shared resources always involve in deadlocks. (True / False)
5. A deadlock _____ algorithm requires each process to make known in advance the maximum number of resources of each type that it may need.
6. _____ algorithm is used where resources have multiple instances.
   a) Resource Allocation Graph
   b) Banker's
   c) Resource Request
   d) None of the above

6.7 Deadlock Detection

If the system does not ensure that a deadlock cannot be prevented or a deadlock cannot be avoided, then a deadlock may occur. In case a deadlock occur the system must-

1) Detect the deadlock
2) Recover from the deadlock

6.7.1 Single instance of a resource

If the system has resources, all of which have only single instances, then a deadlock detection algorithm, which uses a variant of the resource allocation graph, can be used. The graph used in this case is called a wait-for graph. The wait-for graph is a directed graph having vertices and edges. The

vertices represent processes and directed edges are present between two processes, one of which is waiting for a resource held by the other. Two edges $P_i \rightarrow R_q$ and $R_q \rightarrow P_j$ in the resource allocation graph are replaced by one edge $P_i \rightarrow P_j$ in the wait-for graph. Thus, the wait-for graph is obtained by removing vertices representing resources and then collapsing the corresponding edges in a resource allocation graph. An Illustration is shown in Figure 6.6.

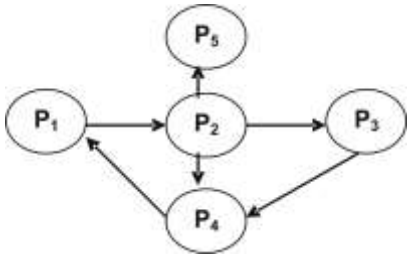

Fig. 6.6: Wait-for graph

As in the previous case, a cycle in a wait-for graph indicates a deadlock. Therefore, the system maintains a wait-for graph and periodically invokes an algorithm to check for a cycle in the wait-for graph.

6.7.2 Multiple instances of a resource

A wait-for graph is not applicable for detecting deadlocks where multiple instances of resources exist. This is because there is a situation where a cycle may or may not indicate a deadlock. If this is so then a decision cannot be made. In situations where there are multiple instances of resources, an algorithm similar to Banker's algorithm for deadlock avoidance is used.

Data structures used are similar to those used in Banker's algorithm and are given below:

1) n: Number of processes in the system.
2) m: Number of resource types in the system.
3) Available: is a vector of length m. Each entry in this vector gives maximum instances of a resource type that are available at the instant.
4) Allocation: is a n x m vector which at any instant defines the number of resources of each type currently allocated to each of the m processes.

5) Request: is also a n x m vector defining the current requests of each process. Request[i][j] = k means the ith process P_i is requesting for k instances of the jth resource type R_j.

ALGORITHM

1) Define a vector Work of length m and a vector Finish of length n.
2) Initialize Work = Available and
   For I = 1, 2, ….., n
   If $Allocation_i$!= 0
   Finish[i] = false
   Else
   Finish[i] = true
3) Find an i such that
   a. Finish[i] = false and
   b. $Request_i$ <= Work
   If such an i does not exist , go to step 5.
4) Work = Work + $Allocation_i$
   Finish[i] = true
   Go to step 3.
5) If finish[i] = true for all i, then the system is not in deadlock.

Else the system is in deadlock with all processes corresponding to Finish[i] = false being deadlocked.

Illustration: n = 5< P_0, P_1, P_2, P_3, P_4 >
M = 3 < A, B, C >
Initially Available = < 7, 2, 6 >

At an instant t0, the data structures have the following values:

	Allocation	**Request**	**Available**
	A B C	A B C	A B C
P0	0 1 0	0 0 0	0 0 0
P_1	2 0 0	2 0 2	
P_2	3 0 3	0 0 0	
P_3	2 1 1	1 0 0	
P_4	0 0 2	0 0 2	

To prove that the system is not deadlocked, use the above algorithm as follows:

Step	Work	Finish	Safe sequence
0	0 0 0	F F F F F	< >
1	0 1 0	T F F F F	< P_0 >
2	3 1 3	T F T F F	< P_0, P_2 >
3	5 2 4	T F T T F	< P_0, P_2, P_3 >
4	5 2 6	T F T T T	< P_0, P_2, P_3, P_4 >
5	7 2 6	T T T T T	< P_0, P_2, P_3, P_4, P_1 >

Now at an instant t_1, $Request_2$ = < 0, 0, 1 > and the new values in the data structures are as follows:

	Allocation	Request	Available
	A B C	A B C	A B C
P0	0 1 0	0 0 0	0 0 0
P_1	2 0 0	2 0 2	
P_2	3 0 3	0 0 1	
P_3	2 1 1	1 0 0	
P_4	0 0 2	0 0 2	

To prove that the system is deadlocked, use the above algorithm as follows:

Step	Work	Finish	Safe sequence
0	0 0 0	F F F F F	< >
1	0 1 0	T F F F F	< P_0 >

The system is in deadlock with processes P_1, P_2, P_3, and P_4 deadlocked.

6.7.3 Recovery from deadlock

Once a deadlock has been detected, it must be broken. Breaking a deadlock may be manual by the operator when informed of the deadlock or automatically by the system. There exist two options for breaking deadlocks:

1) abort one or more processes to break the circular-wait condition causing deadlock
2) preempting resources from one or more processes which are deadlocked.

In the first option, one or more processes involved in deadlock could be terminated to break the deadlock. Then abort either all processes or abort one process at a time till deadlock is broken. The first case guarantees that

the deadlock will be broken. But processes that have executed for a long time will have to restart all over again. The second case is better but has considerable overhead as detection algorithm has to be invoked after terminating every process. Also choosing a process that becomes a victim for termination is based on many factors like

- priority of the processes
- length of time each process has executed and how much more it needs for completion
- type of resources and the instances of each that the processes use
- need for additional resources to complete
- nature of the processes, whether iterative or batch

Based on these and many more factors, a process that incurs minimum cost on termination becomes a victim.

In the second option some resources are preempted from some processes and given to other processes until the deadlock cycle is broken. Selecting the victim whose resources can be preempted is again based on the minimum cost criteria. Parameters such as number of resources a process is holding and the amount of these resources used thus far by the process are used to select a victim. When resources are preempted, the process holding the resource cannot continue. A simple solution is to abort the process also. Better still is to rollback the process to a safe state to restart later. To determine this safe state, more information about running processes is required which is again an overhead. Also starvation may occur when a victim is selected for preemption; the reason being resources from the same process may again and again be preempted. As a result the process starves for want of resources. Ensuring that a process can be a victim only a finite number of times by having this information as one of the parameters for victim selection could prevent starvation.

Prevention, avoidance and detection are the three basic approaches to handle deadlocks. But they do not encompass all the problems encountered. Thus a combined approach of all the three basic approaches is used.

Self Assessment Questions

7. If the system does not ensure that a deadlock cannot be prevented or a deadlock cannot be avoided, then a deadlock may occur. (True / False)

8. _____ is a variant of the resource allocation graph, which can be used to detect deadlocks in the system that has resources, all of which have only single instances.
9. _____, _____ and _____ are the three basic approaches to handle deadlocks.

6.8 Summary

Let's summarize the key concepts covered in this unit:

- Since many processes compete for a finite set of resources, there is always a possibility that requested resources are not readily available. This makes processes wait.
- When there is a set of processes where each process in the set waits on another from the same set for release of a wanted resource, then a deadlock has occurred.
- Prevention, avoidance and detection are the three basic approaches to handle deadlocks.

6.9 Terminal Questions

1. What are the necessary conditions for deadlock to occur?
2. Write a note on Resource Allocation Graph.
3. Describe safe, unsafe and deadlock state of a system.
4. Explain how Banker's algorithm is used to check safe state of a system.

6.10 Answers

Self-Assessment Questions

1. True
2. Blocked
3. a) Mutual Exclusion
4. False
5. Avoidance
6. b) Banker's
7. True
8. Wait-for Graph
9. Prevention, Avoidance, Detection

Terminal Questions

1. A deadlock occurs in a system if the following four conditions hold simultaneously. They are Mutual Exclusion, Hold and Wait, No Preemption and Circular Wait. (Refer Section 6.3.1)
2. Deadlocks can be described by a resource allocation graph. The resource allocation graph is a directed graph consisting of vertices and directed edges. The vertex set is partitioned into two types, a subset representing processes and another subset representing resources. (Refer Section 6.3.2)
3. A system is said to be in a safe state if it can allocate resources upto the maximum available and is not in a state of deadlock. (Refer Section 6.6.1)
4. A new process entering the system must make known a priori the maximum instances of each resource that it needs subject to the maximum available for each type. (Refer Section 6.6.3)