Dr. SNS RAJALAKSHMI COLLEGE OF ARTS & SCIENCE(Autonomous)

Coimbatore – 49.

DEPARTMENT OF COMPUTER APPLICATIONS

COURSE : OPERATING SYSTEM

CLASS : I BCA 'B'

Unit III

DEADLOCKS

Deadlocks :

- Deadlock is a situation where a set of processes are blocked because each process is
  - → holding a resource and
  - → waiting for another resource held by some other process.
- Real life example:

  When 2 trains are coming toward each other on same track and there is only one track, none of the trains can move once they are in front of each other.
- Similar situation occurs in operating systems when there are two or more processes hold some resources and wait for resources held by other(s).
- Here is an example of a situation where deadlock can occur (Figure 3.1).

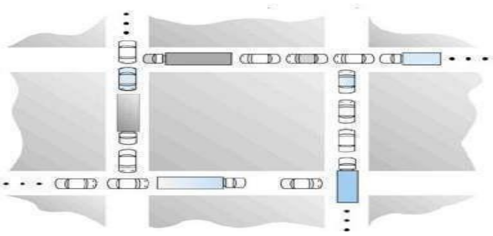


Figure 3.1 Deadlock Situation

System Model :

- A system consist of finite number of resources. (For ex: memory, printers, CPUs).
- These resources are distributed among number of processes.

- A process must

  → request a resource before using it and

  → release the resource after using it.

- The process can request any number of resources to carry out a given task.
- The total number of resource requested must not exceed the total number of resources available.
- In normal operation, a process must perform following tasks in sequence:

  1) Request

  ➢ If the request cannot be granted immediately (for ex: the resource is being used by another process), then the requesting-process must wait for acquiring the resource.

  ➢ For example: open( ), malloc( ), new( ), and request( )

  2) Use

  ➢ The process uses the resource.

  ➢ For example: prints to the printer or reads from the file.

  3) Release

  ➢ The process releases the resource.

  ➢ So that, the resource becomes available for other processes.

  ➢ For example: close( ), free( ), delete( ), and release( ).

- A set of processes is deadlocked when every process in the set is waiting for a resource that is currently allocated to another process in the set.
- Deadlock may involve different types of resources.

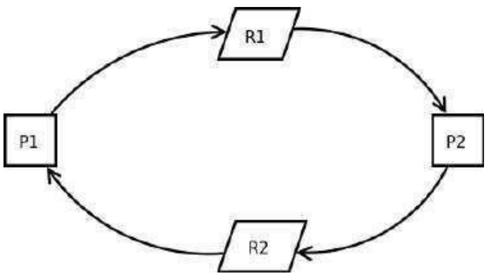


Figure 3.2

- As shown in figure 3.2,

  Both processes P1 & P2 need resources to continue execution.

  P1 requires additional resource R1 and is in possession of resource R2.

  P2 requires additional resource R2 and is in possession of R1.

- Thus, neither process can continue.
- Multithread programs are good candidates for deadlock because they compete for shared resources.

3.1 Deadlock Characterization

- In a deadlock, processes never finish executing, and system resources are tied up, preventing other jobs from starting.

3.1.1 Necessary Conditions

- There are four conditions that are necessary to achieve deadlock:

1) Mutual Exclusion

➢ At least one resource must be held in a non-sharable mode.

➢ If any other process requests this resource, then the requesting-process must wait for the resource to be released.

2) Hold and Wait

➢ A process must be simultaneously

→ holding at least one resource and

→ waiting to acquire additional resources held by the other process.

3) No Preemption

➢ Once a process is holding a resource (i.e. once its request has been granted), then that resource cannot be taken away from that process until the process voluntarily releases it.

4) Circular Wait

➢ A set of processes { P0, P1, P2, . . ., PN } 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, and so on

3.1.2 Resource-Allocation-Graph

- The resource-allocation-graph (RAG) is a directed graph that can be used to describe the deadlock situation.
- RAG consists of a

→ set of vertices (V) and

→ set of edges (E).

- V is divided into two types of nodes

1) P={P1,P2 .... Pn} i.e., set consisting of all active processes in the system.

2) R={R1,R2 ....Rn} i.e., set consisting of all resource types in the system.

- E is divided into two types of edges:

  1) Request Edge

  ➢ A directed-edge $P_i \rightarrow R_j$ is called a request edge.

  ➢ $P_i \rightarrow R_j$ indicates that process P_i has requested a resource R_j.

  2) Assignment Edge

  ➢ A directed-edge $R_j \rightarrow P_i$ is called an assignment edge.

  ➢ $R_j \rightarrow P_i$ indicates that a resource R_j has been allocated to process P_i.

- Suppose that process Pi requests resource Rj.

  Here, the request for Rj from Pi can be granted only if the converting request-edge to assignment-edge do not form a cycle in the resource-allocation graph.

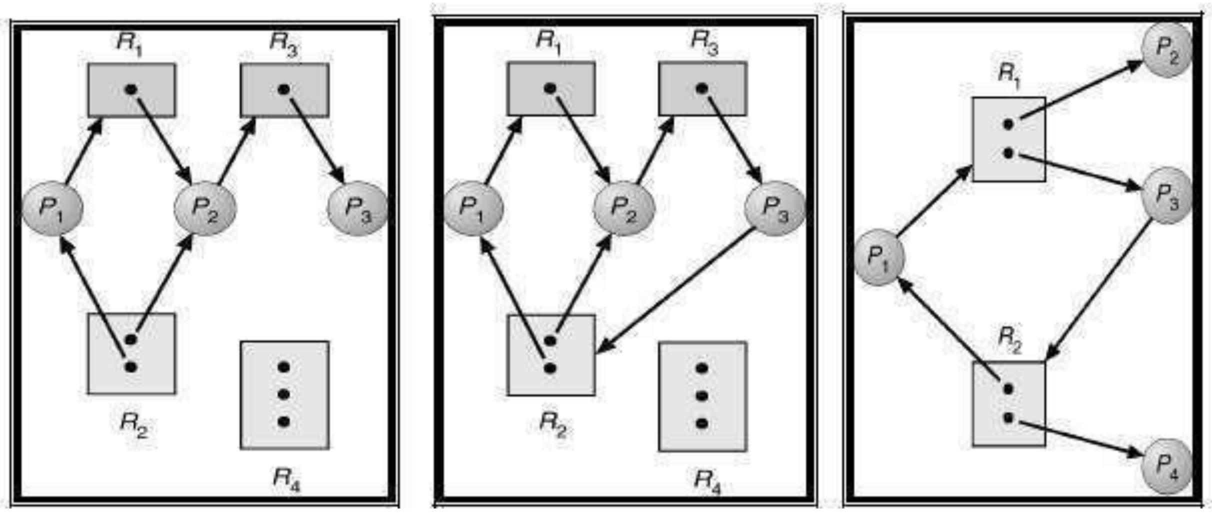
- Pictorially,

  → We represent each process P_i as a **circle**.

  → We represent each resource-type R_j as a **rectangle**.

- As shown in below figures, the RAG illustrates the following 3 situation (Figure 3.3):

  1) RAG with a deadlock
  2) RAG with a cycle and deadlock
  3) RAG with a cycle but no deadlock



(a) Resource allocation Graph (b) With a deadlock (c) with cycle but no deadlock Figure 3.3 Resource allocation graphs

Conclusion:

1) If a graph contains no cycles, then the system is not deadlocked.

2) If the graph contains a cycle then a deadlock may exist.

Therefore, a cycle means deadlock is possible, but not necessarily present.

3.2 Methods for Handling Deadlocks

• There are three ways of handling deadlocks:

1) Deadlock prevention or avoidance - Do not allow the system to get into a deadlocked state.
2) Deadlock detection and recovery - Abort a process or preempt some resources when deadlocks are detected.
3) Ignore the problem all together - If deadlocks only occur once a year or so, it may be better to simply let them happen and reboot the system.

• In order to avoid deadlocks, the system must have additional information about all processes.

• In particular, the system must know what resources a process will or may request in the future.

• Deadlock detection is fairly straightforward, but deadlock recovery requires either aborting processes or preempting resources.

• If deadlocks are neither prevented nor detected, then when a deadlock occurs the system will gradually slow down.

3.3 Deadlock-Prevention

• Deadlocks can be eliminated by preventing at least one of the four required conditions:

1) Mutual exclusion
2) Hold-and-wait
3) No preemption
4) Circular-wait.

3.3.1 Mutual Exclusion

• This condition must hold for non-sharable resources.

• For example:

A printer cannot be simultaneously shared by several processes.

• On the other hand, shared resources do not lead to deadlocks.

• For example:

Simultaneous access can be granted for read-only file.

• A process never waits for accessing a sharable resource.

• In general, we cannot prevent deadlocks by denying the mutual-exclusion condition because some resources are non-sharable by default.

3.3.2 Hold and Wait

- To prevent this condition:

  The processes must be prevented from holding one or more resources while simultaneously waiting for one or more other resources.

- There are several solutions to this problem.
- For example:

  Consider a process that

  → copies the data from a tape drive to the disk

  → sorts the file and

  → then prints the results to a printer.

  Protocol-1

  ➢ Each process must be allocated with all of its resources before it begins execution.

  ➢ All the resources (tape drive, disk files and printer) are allocated to the process at the beginning.

  Protocol-2

  ➢ A process must request a resource only when the process has none.

  ➢ Initially, the process is allocated with tape drive and disk file.

  ➢ The process performs the required operation and releases both tape drive and disk file.

  ➢ Then, the process is again allocated with disk file and the printer

  ➢ Again, the process performs the required operation & releases both disk file and the printer.

- Disadvantages of above 2 methods:

  1) Resource utilization may be low, since resources may be allocated but unused for a long period.

  2) Starvation is possible.

3.3.3 No Preemption

• To prevent this condition: the resources must be preempted.

• There are several solutions to this problem.

Protocol-1

• If a process is holding some resources and requests another resource that cannot be immediately allocated to it, then all resources currently being held are preempted.

• The preempted resources are added to the list of resources for which the process is waiting.

• The process will be restarted only when it regains the old resources and the new resources that it is requesting.

Protocol-2

```
If (resources are available)
then
{
        allocate resources to the process
}
else
{
        If (resources are allocated to waiting process)
        then
        {
                preempt the resources from the waiting process
                allocate the resources to the requesting-process
                the requesting-process must wait
        }

}
```

• When a process request resources, we check whether they are available or not.

• These 2 protocols may be applicable for resources whose states are easily saved and restored, such as registers and memory.

• But, these 2 protocols are generally not applicable to other devices such as printers and tape drives.

3.3.4 Circular-Wait

• Deadlock can be prevented by using the following 2 protocol:

Protocol-1

- Assign numbers all resources.
- Require the processes to request resources only in increasing/decreasing order.

Protocol-2

- Require that whenever a process requests a resource, it has released resources with a lower number.

- One big challenge in this scheme is determining the relative ordering of the different resources.

3.4 Deadlock Avoidance

- The general idea behind deadlock avoidance is to prevent deadlocks from ever happening.
- Deadlock-avoidance algorithm

  → requires more information about each process, and

  → tends to lead to low device utilization.
- For example:

  1) In simple algorithms, the scheduler only needs to know the maximum number of each resource that a process might potentially use.

  2) In complex algorithms, the scheduler can also take advantage of the schedule of exactly what resources may be needed in what order.
- A deadlock-avoidance algorithm dynamically examines the resources allocation state to ensure that a circular-wait condition never exists.
- The resource-allocation state is defined by

  → the number of available and allocated resources and

  → the maximum demand of each process.

3.4.1 Safe State

- A state is safe if the system can allocate all resources requested by all processes without entering a deadlock state.
- A state is safe if there exists a safe sequence of processes {P0, P1, P2, ..., PN} such that

  the requests of each process(Pi) can be satisfied by the currently available resources.
- If a safe sequence does not exist, then the system is in an unsafe state, which may lead to deadlock.
- All safe states are deadlock free, but not all unsafe states lead to deadlocks. (Figure 3.4).

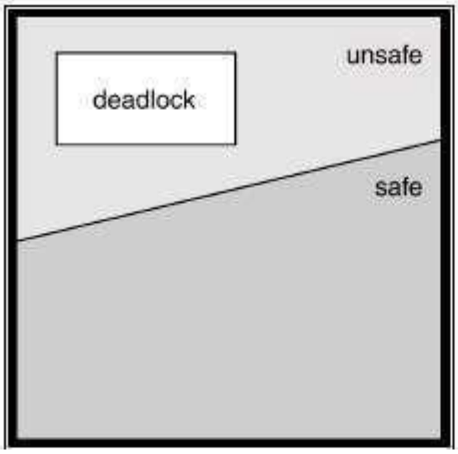


Figure 3.4 Safe, unsafe, and deadlock state spaces

3.4.2 Resource-Allocation-Graph Algorithm

- If resource categories have only single instances of their resources, then deadlock states can be detected by cycles in the resource-allocation graphs.
- In this case, unsafe states can be recognized and avoided by augmenting the resource-allocation graph with claim edges (denoted by a dashed line).
- Claim edge Pi → Rj indicated that process Pi may request resource Rj at some time in future.
- The important steps are as below:

  1) When a process Pi requests a resource Rj, the claim edge Pi → Rj is converted to a request edge.

  2) Similarly, when a resource Rj is released by the process Pi, the assignment edge Rj → Pi is reconverted as claim edge Pi → Rj.

  3) The request for Rj from Pi can be granted only if the converting request edge to assignment edge do not form a cycle in the resource allocation graph.
- To apply this algorithm, each process Pi must know all its claims before it starts executing.
- Conclusion:

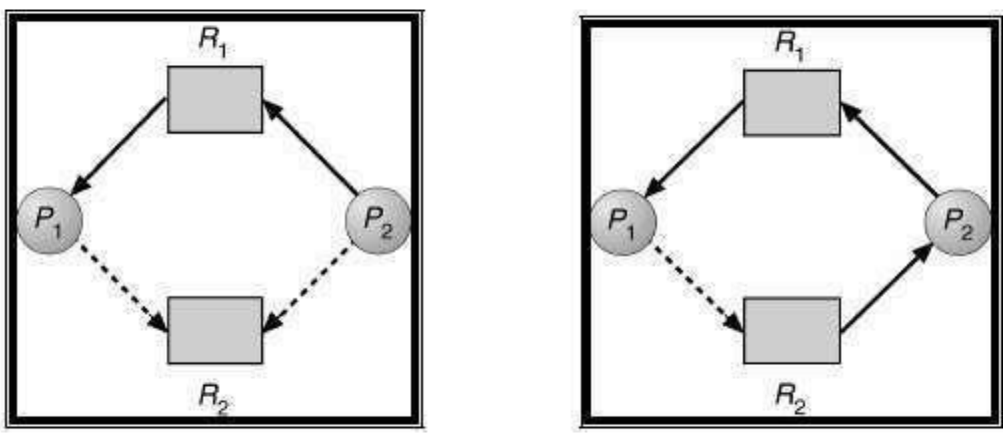
  1) If no cycle exists, then the allocation of the resource will leave the system in a safe state.

  2) If cycle is found, system is put into unsafe state and may cause a deadlock.
- For example: Consider a resource allocation graph shown in Figure 3.5(a).

  ➢ Suppose P2 requests R2.

  ➢ Though R2 is currently free, we cannot allocate it to P2 as this action will create a cycle in the graph as shown in Figure 3.5(b).

  ➢ This cycle will indicate that the system is in unsafe state: because, if P1 requests R2 and P2 requests R1 later, a deadlock will occur.



(a) For deadlock avoidance (b) an unsafe state

Figure 3.5 Resource Allocation graphs

- Problem:

  The resource-allocation graph algorithm is not applicable when there are multiple instances for each resource.

- Solution:

  Use banker's algorithm.

3.4.3 Banker's Algorithm

- This algorithm is applicable to the system with multiple instances of each resource types.
- However, this algorithm is less efficient then the resource-allocation-graph algorithm.
- When a process starts up, it must declare the maximum number of resources that it may need.
- This number may not exceed the total number of resources in the system.
- When a request is made, the system determines whether granting the request would leave the system in a safe state.
- If the system in a safe state,

  the resources are allocated;

  else

  the process must wait until some other process releases enough resources.
- Assumptions:

  Let n = number of processes in the system

  Let m = number of resources types.
- Following data structures are used to implement the banker's algorithm.

  1) Available [m]

  - This vector indicates the no. of available resources of each type.
  - If Available[j]=k, then k instances of resource type Rj is available.

  2) Max [n][m]

  - This matrix indicates the maximum demand of each process of each resource.
  - If Max[i,j]=k, then process Pi may request at most k instances of resource type Rj.

  3) Allocation [n][m]

  - This matrix indicates no. of resources currently allocated to each process.
  - If Allocation[i,j]=k, then Pi is currently allocated k instances of Rj.

  4) Need [n][m]

  - This matrix indicates the remaining resources need of each process.
  - If Need[i,j]=k, then Pi may need k more instances of resource Rj to complete its task.
  - So, Need[i,j] = Max[i,j] - Allocation[i]
- The Banker's algorithm has two parts: 1) Safety Algorithm

  2) Resource – Request Algorithm

3.4.3.1 Safety Algorithm

- This algorithm is used for finding out whether a system is in safe state or not.
- Assumptions:

  Work is a working copy of the available resources, which will be modified during the analysis. Finish is a vector of boolean values indicating whether a particular process can finish.

```
Step 1:
        Let Work and Finish be two vectors of length m and n respectively.
                Initialize:
                        Work =  Available
                        Finish[i] = false for i=1,2,3,.......n
Step 2:
        Find an index(i) such that both
                a) Finish[i] = false
                b) Need i <= Work.
        If no such i exist, then go to step 4
Step 3:
                Set:
                    Work = Work + Allocation(i)
                    Finish[i] = true
        Go to step 2
Step 4:
        If Finish[i] = true for all i, then the system is in safe state.
```

3.4.3.2 Resource-Request Algorithm

- This algorithm determines if a new request is safe, and grants it only if it is safe to do so.
- When a request is made (that does not exceed currently available resources), pretend it has been granted, and then see if the resulting state is a safe one. If so, grant the request, and if not, deny the request.
- Let Request(i) be the request vector of process Pi.
- If Request(i)[j]=k, then process Pi wants K instances of the resource type Rj.

```
Step 1:
        If Request(i) <= Need(i)
        then
                go to step 2
        else
                raise an error condition, since the process has exceeded its maximum claim.
Step 2:
        If Request(i) <= Available
        then
                go to step 3
        else
                Pi must wait, since the resources are not available.
Step 3:
        If the system want to allocate the requested resources to process Pi then modify the
        state as follows:
                Available = Available – Request(i)
                Allocation(i) = Allocation(i) + Request(i)
                Need(i) = Need(i) – Request(i)
Step 4:
        If the resulting resource-allocation state is safe,
                then    i) transaction is complete and
                        ii) Pi is allocated its resources.
Step 5:
         If the new state is unsafe,
                 then   i) Pi must wait for Request(i) and
                        ii) old resource-allocation state is restored.
```

3.4.3.3 An Illustrative Example

Question: Consider the following snapshot of a system:

	Allocation			Max			Available		
	A	B	C	A	B	C	A	B	C
P0	0	1	0	7	5	3	3	3	2
P1	2	0	0	3	2	2			
P2	3	0	3	9	0	2			
P3	2	1	1	2	2	2			
P4	0	0	2	4	3	3			

Answer the following questions using Banker's algorithm.

i) What is the content of the matrix need?

ii) Is the system in a safe state?

iii) If a request from process P1 arrives for (1 0 2) can the request be granted immediately?

Solution (i):

- The content of the matrix Need is given

  by Need = Max - Allocation

- So, the content of Need Matrix is:

	Need		
	A	**B**	**C**
P0	7	4	3
P1	1	2	2
P2	6	0	0
P3	0	1	1
P4	4	3	1

Solution (ii):

- Applying the Safety algorithm on the given

  system, Step 1: Initialization

  Work = Available i.e. Work =3 3 2

……P0………P1……..P2……..P3…

…P4….. Finish = | false | false | false |

false | false |

Step 2: For i=0

Finish[P0] = false and Need[P0]<=Work i.e. (7 4 3)<=(3 3 2) false

So P0 must wait.

Step 2: For i=1

Finish[P1] = false and Need[P1]<=Work i.e. (1 2 2)<=(3 3 2) true

So P1 must be kept in safe sequence.

Step 3: Work = Work + Allocation[P1] =(3 3 2)+(2 0 0)=(5 3 2)

…….P0………P1…….P2…….P3 P4……

Finish = | false | true | false | false | false |

Step 2: For i=2

Finish[P2] = false and Need[P2]<=Work i.e. (6 0 0)<=(5 3 2) false

So P2 must wait.

Step 2: For i=3

Finish[P3] = false and Need[P3]<=Work i.e. (0 1 1)<=(5 3 2) true

So P3 must be kept in safe sequence.

Step 3: Work = Work + Allocation[P3] = (5 3 2)+(2 1 1)=(7 4 3)

……P0………P1……P2………..P3…

….P4…. Finish = | false | true | false |

true | false |

Step 2: For i=4

Finish[P4] = false and Need[P4]<=Work i.e. (4 3 1)<=(7 4 3) true

So P4 must be kept in safe sequence.

Step 3: Work = Work + Allocation[P4] =(7 4 3)+(0 0 2)=(7 4 5)

……P0………P1……P2………P3……

.P4….. Finish= | false | true | false | true |

true |

Step 2: For i=0

Finish[P0] = false and Need[P0]<=Work i.e. (7 4 3)<=(7 4 5) true

So P0 must be kept in safe sequence.

Step 3: Work = Work + Allocation[P0] =(7 4 5)+(0 1 0)=(7 5 5)

…..P0…….P1………P2…….P3…….P

4…. Finish= | true | true | false | true |

true |

Step 2: For i=2

Finish[P2] = false and Need[P2]<=Work i.e. (6 0 0) <=(7 5 5) true

So P2 must be kept in safe sequence.

Step 3: Work = Work + Allocation[P2] =(7 5 5)+(3 0 2)=(10 5 7)

…..P0…….P1………P2…….P3…….P

4…. Finish= | true | true | true | true |

true |

Step 4: Finish[Pi] = true for 0<=i<=4

Hence, the system is currently in a safe state.

The safe sequence is <P1, P3, P4, P0, P2>.

Conclusion: Yes, the system is currently in a safe state.

Solution (iii): P1 requests (1 0 2) i.e. Request[P1]=1 0 2

- To decide whether the request is granted, we use Resource Request algorithm. Step 1: Request[P1]<=Need[P1] i.e. (1 0 2)<=(1 2 2) true.

Step 2: Request[P1]<=Available i.e. (1 0 2)<=(3 3 2) true.

Step 3: Available = Available – Request[P1] = (3 3 2) - (1 0 2)= (2 3 0)

Allocation[P1] = Allocation[P1] + Request[P1] = (2 0 0) + (1 0 2)= (3 0 2)

Need[P1] = Need[P1] – Request[P1] = (1 2 2) - (1 0 2)= (0 2 0)

- We arrive at the following new system state:

	Allocation			**Max**			**Available**		
	A	**B**	**C**	**A**	**B**	**C**	**A**	**B**	**C**
P0	0	1	0	7	5	3	2	3	0
P1	3	0	2	3	2	2			
P2	3	0	2	9	0	2			
P3	2	1	1	2	2	2			
P4	0	0	2	4	3	3			

- The content of the Need is given
  matrix Need by Allocation
  = Max - Matrix is:
- So, the content of Need

	Need		
	A	**B**	**C**
P0	7	4	3
P1	0	2	0
P2	6	0	0
P3	0	1	1
P4	4	3	1

- To determine whether this new system state is safe, we again execute Safety algorithm. Step 1: Initialization

Here, m=3, n=5

Work = Available i.e. Work =2 3 0

…..P0………P1………P2……….P3…

….P4…. Finish = | false | false | false | false | false |

Step 2: For i=0

Finish[P0] = false and Need[P0]<=Work i.e. (7 4 3)<=(2 3 0) false

So P0 must wait.

Step 2: For i=1

Finish[P1] = false and Need[P1]<=Work i.e. (0 2 0)<=(2 3 0) true

So P1 must be kept in safe sequence.

Step 3: Work = Work + Allocation[P1] =(2 3 0)+(3 0 2)=(5 3 2)

……P0………P1…….P2………P3…

….P4…… Finish = | false | true | false |

false | false |

Step 2: For i=2

Finish[P2] = false and Need[P2]<=Work i.e. (6 0 0) <=(5 3 2) false

So P2 must wait.

Step 2: For i=3

Finish[P3] = false and Need[P3]<=Work i.e. (0 1 1)<=(5 3 2) true

So P3 must be kept in safe sequence.

Step 3: Work = Work + Allocation[P3] = (5 3 2)+(2 1 1)=(7 4 3)

…..P0……….P1……P2……..P3…….

P4…... Finish = | false | true | false | true |

false |

Step 2: For i=4

Finish[P4] = false and Need[P4]<=Work i.e. (4 3 1)<=(7 4 3) true

So P4 must be kept in safe sequence.

Step 3: Work = Work + Allocation[P4] =(7 4 3)+(0 0 2)=(7 4 5)

……P0………P1……P2………P3…

….P4…. Finish = | false | true | false |

true | true |

Step 2: For i=0

Finish[P0] = false and Need[P0]<=Work i.e. (7 4 3)<=(7 4 5) true

So P0 must be kept in safe sequence.

Step 3: Work = Work + Allocation[P0] =(7 4 5)+(0 1 0)=(7 5 5)

……P0…….P1………P2……P3P4….

Finish = | true | true | false | true | true |

Step 2: For i=2

Finish[P2] = false and Need[P2]<=Work i.e. (6 0 0) <=(7 5 5) true

So P2 must be kept in safe sequence.

Step 3: Work = Work + Allocation[P2] =(7 5 5)+(3 0 2)=(10 5 7)

…..P0…….P1………P2…….P3…….P

4…. Finish= | true | true | true | true |

true |

Step 4: Finish[Pi] = true for 0<=i<=4

Hence, the system is in a safe state.

The safe sequence is <P1, P3, P4, P0, P2>.

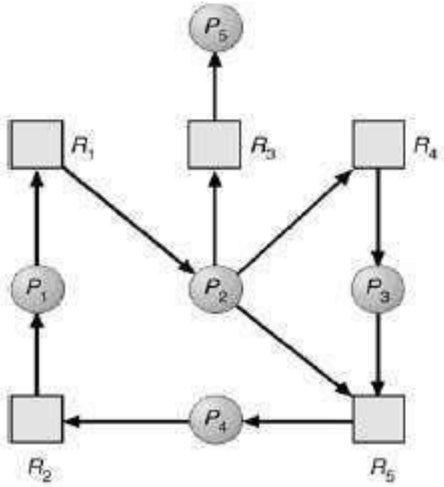
Conclusion: Since the system is in safe sate, the request can be granted.

3.5 Deadlock Detection

- If a system does not use either deadlock-prevention or deadlock-avoidance algorithm then a deadlock may occur.
- In this environment, the system must provide
  1) An algorithm to examine the system-state to determine whether a deadlock has occurred.
  2) An algorithm to recover from the deadlock.

3.5.1 Single Instance of Each Resource Type

- If all the resources have only a single instance, then deadlock detection-algorithm can be defined using a wait-for-graph.
- The wait-for-graph is applicable to only a single instance of a resource type.
- A wait-for-graph (WAG) is a variation of the resource-allocation-graph.
- The wait-for-graph can be obtained from the resource-allocation-graph by
  → removing the resource nodes and
  → collapsing the appropriate edges.
- An edge from P_i to P_j implies that process P_i is waiting for process P_j to release a resource that P_i needs.
- An edge $P_i \rightarrow P_j$ exists if and only if the corresponding graph contains two edges
  1) $P_i \rightarrow R_q$ and
  2) $R_q \rightarrow P_j$.
- For example:
  Consider resource-allocation-graph shown in Figure 3.6
  Corresponding wait-for-graph is shown in Figure 3.7.



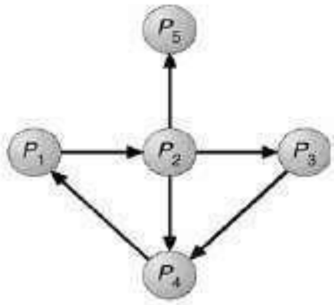


Figure 3.6 Resource-allocation-graph Figure 3.7 Corresponding wait-for-graph.

- A deadlock exists in the system if and only if the wait-for-graph contains a cycle.
- To detect deadlocks, the system needs to

  → maintain the wait-for-graph and

  → periodically execute an algorithm that searches for a cycle in the graph.

3.5.2 Several Instances of a Resource Type

- The wait-for-graph is applicable to only a single instance of a resource type.
- Problem: However, the wait-for-graph is not applicable to a multiple instance of a resource type.
- Solution: The following detection-algorithm can be used for a multiple instance of a resource type.
- Assumptions:

  Let 'n' be the number of processes in the system

  Let 'm' be the number of resources types.
- Following data structures are used to implement this algorithm.

  1) Available [m]

  - This vector indicates the no. of available resources of each type.
  - If Available[j]=k, then k instances of resource type Rj is available.

  2) Allocation [n][m]

  - This matrix indicates no. of resources currently allocated to each process.
  - If Allocation[i,j]=k, then Pi is currently allocated k instances of Rj.

  3) Request [n][m]

  - This matrix indicates the current request of each process.
  - If Request [i, j] = k, then process Pi is requesting k more instances of resource type Rj.

```
Step 1:
        Let Work and Finish be vectors of length m and n respectively.
                a) Initialize Work = Available
                b)      For i=0,1,2..........n
                        if Allocation(i) != 0
                        then
                                Finish[i] = false;
                        else
                                Finish[i] = true;
Step 2:
        Find an index(i) such that both
                a) Finish[i] = false
                b) Request(i) <= Work.
                If no such i exist, goto step 4.
Step 3:
                Set:
                    Work = Work + Allocation(i)
                    Finish[i] = true
        Go to step 2.
Step 4:
        If Finish[i] = false for some i where 0 < i < n, then the system is in a deadlock state.
```

3.5.3 Detection-Algorithm Usage

- The detection-algorithm must be executed based on following factors:
    1) The frequency of occurrence of a deadlock. 2) The no. of processes affected by the deadlock.
- If deadlocks occur frequently, then the detection-algorithm should be executed frequently.
- Resources allocated to deadlocked-processes will be idle until the deadlock is broken.
- Problem:

    Deadlock occurs only when some processes make a request that cannot be granted immediately.
- Solution 1:
    - The deadlock-algorithm must be executed whenever a request for allocation cannot be granted immediately.
    - In this case, we can identify

        → set of deadlocked-processes and

        → specific process causing the deadlock.
- Solution 2:
    - The deadlock-algorithm must be executed in periodic intervals.
    - For example:

        → once in an hour

        → whenever CPU utilization drops below certain threshold

3.6 Recovery from deadlock

- Three approaches to recovery from deadlock:
  1) Inform the system-operator for manual intervention.
  2) Terminate one or more deadlocked-processes.
  3) Preempt(or Block) some resources.

3.6.1 Process Termination

- Two methods to remove deadlocks:

  1) Terminate all deadlocked-processes.

  ➢ This method will definitely break the deadlock-cycle.

  ➢ However, this method incurs great expense. This is because

  → Deadlocked-processes might have computed for a long time.

  → Results of these partial computations must be discarded.

  → Probably, the results must be re-computed later.

  2) Terminate one process at a time until the deadlock-cycle is eliminated.

  ➢ This method incurs large overhead. This is

  because after each process is aborted,

  deadlock-algorithm must be executed to determine if any other process is still deadlocked

- For process termination, following factors need to be considered:
  1) The priority of process.
  2) The time taken by the process for computation & the required time for complete execution.
  3) The no. of resources used by the process.
  4) The no. of extra resources required by the process for complete execution.
  5) The no. of processes that need to be terminated for deadlock-free execution.
  6) The process is interactive or batch.

3.6.2 Resource Preemption

- Some resources are taken from one or more deadlocked-processes.
- These resources are given to other processes until the deadlock-cycle is broken.
- Three issues need to be considered:

  1) Selecting a victim

  ➢ Which resources/processes are to be pre-empted (or blocked)?

- The order of pre-emption must be determined to minimize cost.
- Cost factors includes
    1. The time taken by deadlocked-process for computation.
    2. The no. of resources used by deadlocked-process.

2) Rollback

- If a resource is taken from a process, the process cannot continue its normal execution.
- In this case, the process must be rolled-back to break the deadlock.
- This method requires the system to keep more info. about the state of all running processes.

3) Starvation

- Problem: In a system where victim-selection is based on cost-factors, the same process may be always picked as a victim.
- As a result, this process never completes its designated task.
- Solution: Ensure a process is picked as a victim only a (small) finite number of times.