

4.1 DEADLOCKS

- When processes request a resource and if the resources are not available at that time the process enters into waiting state. Waiting process may not change its state because the resources they are requested are held by other process. This situation is called deadlock.
- The situation where the processes wait for each other to release the resource held by another process is called deadlock.

4.2 SYSTEM MODEL

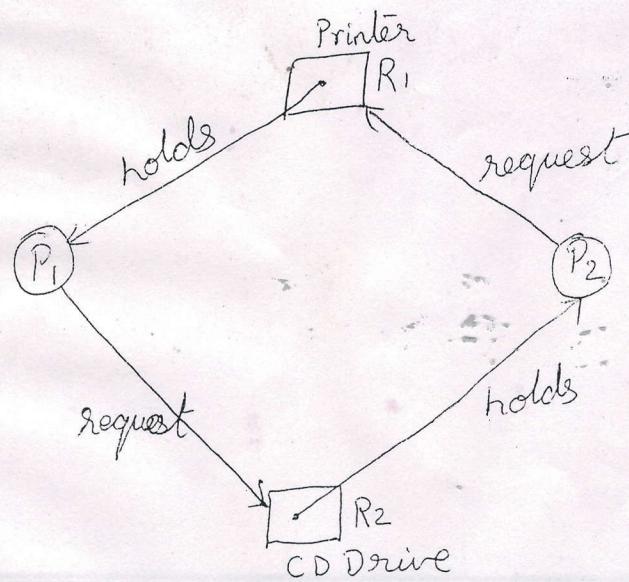
- A system consists of finite number of resources and is distributed among number of processes.
- A process must request a resource before using it and it must release the resource after using it. It can request any number of resources to carry out a designated task. The amount of resource requested may not exceed the total number of resources available.

A process may utilize the resources in only the following sequences:

- Request:-If the request is not granted immediately then the requesting process must wait until it can acquire the resources.
- Use:-The process can operate on the resource.
- Release:-The process releases the resource after using it.

Deadlock may involve different types of resources.

For eg:-Consider a system with one printer (R1) and one tape drive (R2). If a process P_i currently holds a printer R1 and a process P_j holds the tape drive R2. If process P_i request a tape drive and process P_j request a printer then a deadlock occurs.



4.3 DEADLOCK CHARACTERIZATION

A deadlock situation can occur if the following 4 conditions occur simultaneously in a system:-

1. **Mutual Exclusion:** Only one process is holding the resource at a time. If any other process requests for the resource, the requesting process must wait until the resource has been released.
2. **Hold and Wait:** A process ^{is} holding at least one resource and waiting to acquire additional resources that are currently being held by the other process.
3. **No Preemption:** Resources cannot be preempted i.e., only the process holding the resources must release it after the process has completed its task.
4. **Circular Wait:** A set $\{P_0, P_1, \dots, P_n\}$ of waiting processes must exist 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 resource held by process P_n and P_n is waiting for the resource held by P_1 .

All the four conditions must hold for a deadlock to occur.

Resource Allocation Graph:

Deadlocks are described by using a directed graph called system resource allocation graph. The graph consists of set of vertices (v) and set of edges (e).

The set of vertices (v) can be described into two different types of nodes

1. $P = \{P_1, P_2, \dots, P_n\}$ i.e., set consisting of all active processes, represented by circle.
2. $R = \{R_1, R_2, \dots, R_n\}$ i.e., set consisting of all resource types in the system, represented by rectangle. With dot representing the instances of that resource type.

There are two types of edges :

1. A directed edge from process P_i to resource type R_j denoted by $P_i \rightarrow R_j$ indicates that P_i requested an instance of resource R_j and is waiting. This edge is called **Request edge**.
2. A directed edge $R_i \rightarrow P_j$ signifies that resource R_j is held by process P_i . This is called **Assignment edge/ Allocation edge**.

The sets P , R , and E :

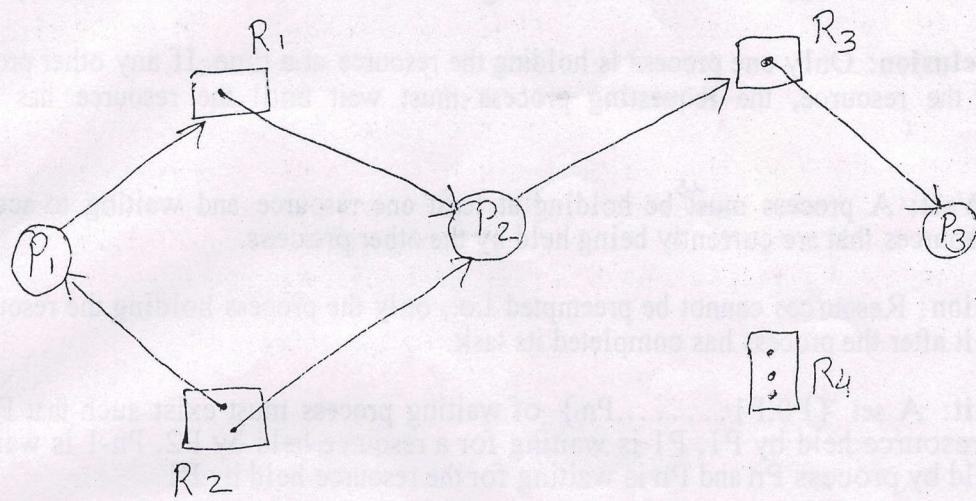
$$P = \{P_1, P_2, P_3\}$$

$$R = \{R_1, R_2, R_3, R_4\}$$

$$E = \{P_1 \rightarrow R_1, P_2 \rightarrow R_3, R_1 \rightarrow P_2, R_2 \rightarrow P_2, R_2 \rightarrow P_1, R_3 \rightarrow P_3\}$$

Resource instances:

- o One instance of resource type R_1
- o Two instances of resource type R_2
- o One instance of resource type R_3
- o Three instances of resource type R_4



- If the graph contains no cycle, then no process in the system is deadlock. If the graph contains a cycle then a deadlock may exist.
- If each resource type has exactly one instance than a cycle implies that a deadlock has occurred. If each resource has several instances then a cycle do not necessarily implies that a deadlock has occurred.
- Note that a **request edge** can be converted into an **assignment edge** by reversing the direction of the arc when the request is granted.

4.4 METHODS FOR HANDLING DEADLOCKS

Deadlock problem can be solved in one of three ways:

- Use a protocol to prevent or avoid deadlocks, ensuring that the system will *never* enter a deadlock state.
- Allow the system to enter a deadlock state, detect it, and recover.
- Ignore the problem altogether and pretend that deadlocks never occur in the system.

4.5 DEADLOCK PREVENTION

For a deadlock to occur each of the four necessary conditions must hold. If at least one of the conditions does not hold then we can prevent occurrence of deadlock.

1. Mutual Exclusion: This holds for non-sharable resources. Eg:- A printer can be used by only one process at a time.
Mutual exclusion is not possible in sharable resources and thus they cannot be involved in deadlock. Read-only files are good examples for sharable resources. A process never

waits for accessing in a sharable resource. So we cannot prevent deadlock by denying the mutual exclusion condition in non-sharable resources.

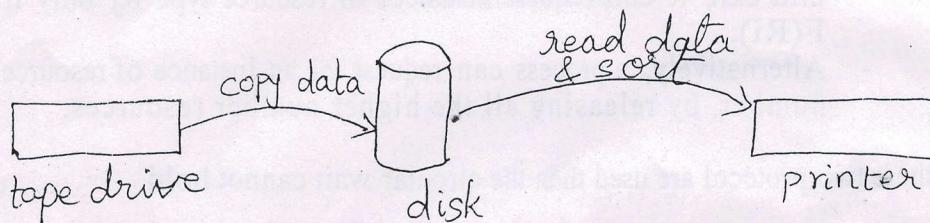
2. Hold and Wait: This condition can be eliminated by forcing a process to release all its resources held by it when it request a resource.

- Request all its resources before execution begins

Eg:-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. If all the resources are allocated at the beginning then the tape drive, disk files and printer are assigned to the process. The main problem with this is it leads to low resource utilization because it requires printer at the last and is allocated with it from the beginning so that no other process can use it.

- Request for resource only when it has none of the resources.

The process is allocated with tape drive and disk file, first. It performs the required operation and releases both. Then the process once again request for disk file and the printer.



3. NoPreemption: To ensure that this condition never occurs the resources must be preempted. The following protocol can be used.

- If a process is holding some resource(R1,R2,R3) and request another resource(R4) that cannot be immediately allocated to it, then all the resources currently held by the requesting process are preempted and added to the list of resources for which other processes may be waiting. The process will be restarted only when it regains the old resources and the new resources that it is requesting.
- When a process (P1) request resources, we check whether they are available or not. If they are available we allocate them else we check that whether they are allocated to some other waiting process(P2). If so we preempt the resources from the waiting process(P2) and allocate them to the requesting process(P1). If resource is not available with P2, P1 has to wait.

4. Circular Wait:-The fourth and the final condition for deadlock is the circular wait condition. One way to ensure that this condition never occurs, is to impose ordering on all resource types and each process requests resource in an increasing order.

-Order all resource types as per number of instances of each type.

-Process should request resource in an increasing order of enumeration

Let $R = \{R_1, R_2, \dots, R_n\}$ be the set of resource types. We assign each resource type, with a unique integer value. This will allows us to compare two

resources and determine whether one precedes the other in ordering.

Eg:-we can define a one to one function

$F: R \rightarrow N$ as follows : where the value 'N' indicates the no. of instances of the resource.

$$F(\text{tape drive})=1$$

$$F(\text{disk drive})=5$$

$$F(\text{printer})=12$$

A process holding 'disk drive', cannot request for tapedrive. but it can request for printer.

Deadlock can be prevented by using the following protocol:

- Each process can request the resource in increasing order. A process can request any number of instances of resource type say R_i initially and then, it can request instances of resource type R_j only if $F(R_j) > F(R_i)$.
- Alternatively, a process can request for an instance of resource of lower number, by releasing all the higher number resources.

If these two protocol are used then the circular wait cannot hold.

Eg; If a process is having an instance of disk drive, whose 'N' value is set as '5'.

- Then it can request for printer only. Because $F(\text{Printer}) > F(\text{disk drive})$.
- If the process wants the tape drive, it has to release all the resources held.

4.6 DEADLOCK AVOIDANCE

Deadlock prevention algorithm may lead to low device utilization and reduces system throughput.

- Avoiding deadlocks requires additional information about the sequence in which the resources are to be requested. With the knowledge of the complete sequences of requests and releases we can decide for each requests whether or not the process should wait.
- For each requests it requires to check whether the resources is currently available. If resources is available, the OS checks if the resource is assigned will the system lead to deadlock. If no then the actual allocation is done.

Safe State:

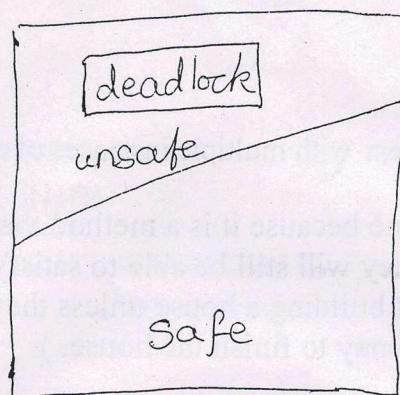
A safe state is a state in which there exists at least one order in which all the process will run completely without resulting in a deadlock.

A system is in safe state if there exists a safe sequence.

A sequence of processes $\langle P_1, P_2, \dots, P_n \rangle$ is a safe sequence for the current allocation state

if for each P_i process request can be satisfied by the currently available resources.

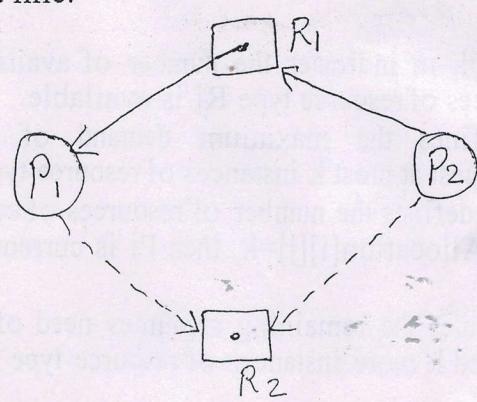
- If the resources that P_i requests are not currently available then P_i can obtain all of its needed resource to complete its designated task.
- A safe state is not a deadlock state.
- Whenever a process request a resource i.e., currently available, the system must decide whether resources can be allocated immediately or whether the process must wait. The request is granted only if the allocation leaves the system in safe state.



The system in unsafe state may lead to deadlock.

Resource Allocation Graph Algorithm:

This algorithm is used only if we have one instance of a resource type. In addition to the request edge and the assignment edge a new edge called claimed edge is used. For eg:- A claim edge $P_i \rightarrow R_j$ indicates that process P_i may request R_j in future. The claim edge is represented by a dotted line.



Process P_1 & P_2 has claim edge to $R_2 \Rightarrow P_1$ & P_2 may request for R_2 in future

When a process P_i requests the resource R_j , the claim edge is converted to a request edge.

When resource R_j is released by process P_i , the assignment edge $R_j \rightarrow P_i$ is replaced by the

claim edge $P_i \rightarrow R_j$.

When a process P_i requests resource R_j the request is granted only if converting the request edge $P_i \rightarrow R_j$ to an assignment edge $R_j \rightarrow P_i$ do not result in a cycle. Cycle detection algorithm is used to detect the cycle. If there are no cycles then the allocation of the resource to process leave the system in safe state.

Banker's Algorithm:

This algorithm is applicable to the system with multiple instances of each resource types.

The Banker's Algorithm gets its name because it is a method that bankers could use to assure that when they lend out resources they will still be able to satisfy all their clients. (A banker won't loan out a little money to start building a house unless they are assured that they will later be able to loan out the rest of the money to finish the house.)

When a process starts up, it must state in advance the maximum allocation of resources it may request, up to the amount available on the system.

When a request is made, the scheduler determines whether granting the request would leave the system in a safe state. If not, then the process must wait until the request can be granted safely.

Several data structures are used to implement the banker's algorithm.

Let ' n ' be the number of processes in the system
and ' m ' be the number of resources types.

We need the following data structures:

- **Available**:-A vector of length m indicates the number of available resources. If $\text{Available}[i]=k$, then k instances of resource type R_j is available.
- **Max**:-An $n*m$ matrix defines the maximum demand of each process if $\text{Max}[i][j]=k$, then P_i may request at most k instances of resource type R_j .
- **Allocation**:-An $n*m$ matrix defines the number of resources of each type currently allocated to each process. If $\text{Allocation}[i][j]=k$, then P_i is currently k instances of resource type R_j .
- **Need**:-An $n*m$ matrix indicates the remaining resources need of each process. If $\text{Need}[i][j]=k$, then P_i may need k more instances of resource type R_j to complete its task.

$$\text{So } \text{Need}[i][j] = \text{Max}[i][j] - \text{Allocation}[i]$$

Safety Algorithm:

This algorithm is used to find out whether or not a system is in safe state or not.

Step 1. Let work and finish be two vectors of length m and n respectively.

Initialize work = available and Finish[i]=false for $i=1, 2, 3, \dots, n$

Step 2. Find i such that both Finish[i]=false and Need[i] \leq work

If no such i exist then go to step 4

Step 3. Work = work + Allocation[Finish[i]]=true Go to step 2

Step 4. If finish[i]=true for all i, then the system is in safe state. This algorithm may require an order of $m \times n \times n$ operation to decide whether a state is safe.

Resource Request Algorithm:

Let Request_i be the request vector of process P_i. If Request[i][j]=k, then process P_i wants K instances of the resource type R_j. When a request for resources is made by process P_i the following actions are taken.

1) If Request(i) \leq Need(i) go to step 2

otherwise raise an error condition since the process has exceeded its maximum claim.

2) If Request(i) \leq Available go to step 3

otherwise P_i must wait. Since the resources are not available.

3) If the system want to allocate the requested resources to process P_i then modify the state as follows

$$\text{Available} = \text{Available} - \text{Request}(i)$$

$$\text{Allocation}(i) = \text{Allocation}(i) + \text{Request}(i)$$

$$\text{Need}(i) = \text{Need}(i) - \text{Request}(i)$$

If the resulting resource allocation state is safe [for this, safety algorithm has to be checked], the transaction is complete and P_i is allocated its resources. If the new state is unsafe then P_i must wait for Request_i and old resource allocation state is restored.

Consider the following snapshot of a system.

Process	Allocation			Maximum			Available			Available \leftarrow freely available R in system
	A	B	C	A	B	C	A	B	C	
P ₀	0	1	0	7	5	3	3	3	2	
P ₁	2	0	0	3	2	2				
P ₂	3	0	2	9	0	2				
P ₃	2	1	1	2	2	2				
P ₄	0	0	2	4	3	3				

Answer the following questions using Bankers Algorithm.

i) Is the system in a safe state?

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

Total instances of resources in system -
 $A = 3+7=10$, $B=3+2=5$, $C=2+5=7$

Need Matrix

	A	B	C
P ₀	7	4	3
P ₁	1	2	2
P ₂	6	0	0
P ₃	0	1	1
P ₄	4	3	1

i) Safety Algorithm

Work = [3, 3, 2], Finish[P₀-P₄] = false.

1) $i = P_0$; Need[P₀] \leq Work

$$[7, 4, 3] \leq [3, 3, 2] \times$$

P₀ cannot execute now.

2) $i = P_1$, Need[P₁] \leq Work

$$[1, 2, 2] \leq [3, 3, 2]$$

P₁ executes...

$$\text{work} = \text{Work} + \text{Allocation}[P_1]$$

$$= [3, 3, 2] + [2, 0, 0] = [5, 3, 2]$$

Finish[P₁] = true

3) $i = P_2$, Need[P₂] \leq Work

$$[6, 0, 0] \leq [5, 3, 2] \times$$

P₂ cannot execute now.

4) $i = P_3$, Need[P₃] \leq Work.

$$[0, 1, 1] \leq [5, 3, 2] \checkmark$$

$\xrightarrow{P_3 \text{ executes}}$ work = Work + Allocation[P₃]

$$= [5, 3, 2] + [2, 1, 1] = [7, 4, 3] \text{ Finish}[P_3] = \text{true}$$

5) $i = P_4, \quad \text{Need}_{[P_4]} \leq \text{Work}$

$$[4, 3, 1] \leq [7, 4, 3]$$

P_4 executes
 $\text{Work} = \text{Work} + \text{Allocation}_{[P_4]}$

$$= [7, 4, 3] + [0, 0, 2]$$

$$= [7, 4, 5].$$

$\text{Finish}_{[P_4]} = \text{true}.$

6) $i = P_0, \quad \text{Need}_{[P_0]} \leq \text{Work}$

$$[7, 4, 3] \leq [7, 4, 5]$$

P_0 executes
 $\text{Work} = \text{Work} + \text{Allocation}_{[P_0]}$

$$= [7, 4, 5] + [0, 1, 0] = [7, 5, 5]$$

$\text{Finish}_{[P_0]} = \text{true}.$

7) $i = P_2, \quad \text{Need}_{[P_2]} \leq \text{Work}$

$$[6, 0, 0] \leq [7, 5, 5]$$

P_2 executes.

$$\text{Work} = \text{Work} + \text{Allocation}_{[P_2]}$$

$$= [7, 5, 5] + [3, 0, 2] = [10, 5, 7]$$

$\text{Finish}_{[P_2]} = \text{true}.$

$\text{Finish}_{[P_0 - P_4]} = \text{true}$, All the processes in the system can be executed. So the system is in safe state

The safe sequence is $\langle P_1, P_3, P_4, P_0, P_2 \rangle$

ii) $P_1 \xrightarrow{\text{request}} [1, 0, 2]$

$$1) \text{Request}_{[P_1]} \leq \text{Need}_{[P_1]}$$

$$[1, 0, 2] \leq [1, 2, 2]. \checkmark$$

Requested resource is needed.

$$2) \text{Request}_{[P_1]} \leq \text{Available}$$

$$[1, 0, 2] \leq [3, 3, 2]$$

Requested resource is available.

- 3) Assume that resources is allocated to P_1 , then the changes are -

$$\text{Allocation}_{[P_1]} = \text{Allocation}_{[P_1]} + \text{Request}_{[P_1]}$$

$$\text{Allocation}_{[P_1]} = [2, 0, 0] + [1, 0, 2] = [3, 0, 2]$$

$$\text{Need}_{[P_1]} = \text{Need}_{[P_1]} - \text{Request}_{[P_1]}$$

$$\text{Need}_{[P_1]} = [1, 2, 2] - [1, 0, 2] = [0, 2, 0]$$

$$\text{Available} = \text{Available} - \text{Request}_{[P_1]}$$

$$= [3, 3, 2] - [1, 0, 2] = [2, 3, 0]$$

Now, with these changes safety algorithm is checked to find if the system is in safe state.

	<u>Allocation</u>			<u>Maximum</u>			<u>Need</u>		
	A	B	C	A	B	C	A	B	C
P_0	0	1	0	7	5	3	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

$$\text{Available} [2, 3, 0]$$

$$\text{Work} = [2, 3, 0] \quad \text{Finish}[P_0 - P_4] = \text{false}$$

1) $i = P_0$, $\text{Need}_{[P_0]} \leq \text{Work}$

$$[7, 4, 3] \leq [2, 3, 0] \times$$

P_0 cannot execute now.

2) $i = P_1$, $\text{Need}_{[P_1]} \leq \text{Work}$

$$[0, 2, 0] \leq [2, 3, 0] \checkmark$$

P_1 executes.

$$\text{Work} = \text{Work} + \text{Allocation}_{[P_1]}$$

$$= [2, 3, 0] + [3, 0, 2] = \underline{\underline{[5, 3, 2]}}$$

$$\text{Finish}[P_1] = \text{true}$$

3) $i = P_2$, $\text{Need}_{[P_2]} \leq \text{Work}$

$$[6, 0, 0] \leq [5, 3, 2] \times$$

P_2 cannot execute now.

4) $i = P_3$, $\text{Need}_{[P_3]} \leq \text{Work}$

$$[0, 1, 1] \leq [5, 3, 2] \checkmark$$

P_3 executes.

$$\text{Work} = \text{Work} + \text{Allocation}_{[P_3]}$$

$$= [5, 3, 2] + [2, 1, 1] = \underline{\underline{[7, 4, 3]}}$$

$$\text{Finish}[P_3] = \text{true}.$$

5) $i = P_4$, $\text{Need}_{[P_4]} \leq \text{Work}$

$$[4, 3, 1] \leq [7, 4, 3] \checkmark$$

P_4 executes.

$$\text{Work} = \text{Work} + \text{Allocation}_{[P_4]}$$

$$= [7, 4, 3] + [0, 0, 2] = \underline{\underline{[7, 4, 5]}}.$$

$$\text{Finish}[P_4] = \text{true}$$

6) $i = P_0$, $\text{Need}_{[P_0]} \leq \text{Work}$

$$[7, 4, 3] \leq [7, 4, 5] \checkmark$$

$$Work = [7, 4, 5] + [0, 1, 0] = [7, 5, 5]$$

Finish $[P_0] = \text{true}$.

Step 7) $i = P_2$, $\text{Need}[P_2] \leq \text{Work}$
 $[6, 0, 0] \leq [7, 5, 5]$
 $\text{Work} = \text{Work} + \text{Allocation}[P_2]$
 $= [7, 5, 5] + [3, 0, 2]$
 $= [10, 5, 7]$
 $\text{Available} = \text{Available} - \text{Allocation}[P_2]$
 $\text{Need}[P_2] = \text{Need}[P_2] - \text{Allocation}[P_2]$
Finish $[P_2] = \text{true}$

If its result is As all $\text{Finish}[P_0 - P_4] = \text{true}$. Even if requested
All the processes have executed completely.
The system is in safe state.

The safe sequence is $\langle P_1, P_3, P_4, P_0, P_2 \rangle$

4.7 DEADLOCK DETECTION

Another way of handling deadlock is, allow the deadlock to occur, then detect and recover from deadlock.

In this environment the system may provide

- An algorithm that examines the state of the system to determine whether a deadlock has occurred.
- An algorithm to recover from the deadlock.

Single Instances of each Resource Type:

If all the resources have only a single instance then we can define deadlock detection algorithm that uses a variant of resource allocation graph called a **wait for graph**. This graph is obtained by removing the nodes of type resources and removing appropriate edges.

An edge from P_i to P_j in wait for graph implies that P_i is waiting for P_j to release a resource that P_i needs.

An edge from P_i to P_j exists in wait for graph if and only if the corresponding resource allocation graph contains the edges $P_i \rightarrow R_q$ and $R_q \rightarrow P_j$.

Deadlock exists within the system if and only if there is a cycle.

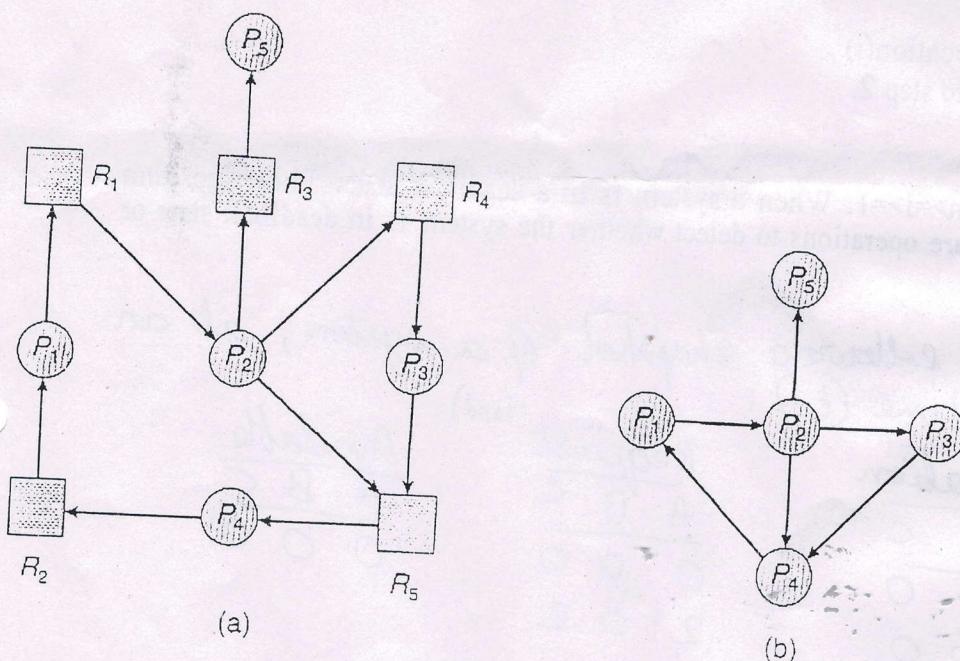


Figure 7.8 (a) Resource-allocation graph. (b) Corresponding wait-for graph.

Unit IV : Deadlocks

Several Instances of a Resource Types:

The wait for graph is applicable to only a single instance of a resource type. The following algorithm applies if there are several instances of a resource type. The following data structures are used:-

- Available:-Is a vector of length m indicating the number of available resources of each type .
- Allocation:-Is an m*n matrix which defines the number of resources of each type currently allocated to each process.
- Request:-Is an m*n matrix indicating the current request of each process. If $\text{request}[i,j]=k$ then P_i is requesting k more instances of resources type R_j .

Step 1. Let work and finish be vectors of length m and n respectively.

Initialize Work = available

```
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 $\text{Finish}[i]=\text{false}$ and $\text{Request}(i) \leq \text{work}$
If no such i exist go to step 4.

Step 3. $\text{Work} = \text{work} + \text{Allocation}(i)$

$\text{Finish}[i] = \text{true}$ Go to step 2.

Step 4. If $\text{Finish}[i] = \text{false}$

for some i where $m \geq i \geq 1$. When a system is in a deadlock state. This algorithm needs an order of m^*n square operations to detect whether the system is in deadlock state or not.

Consider the following snapshot of a system, at an instance of time (t0):

	Allocation			Request ^(need)			Available		
	A	B	C	A	B	C	A	B	C
P_0	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			

If the requested resources are allocated, check

whether system will be in safe state,
Find the safe sequence.

$\text{Work} = (0, 0, 0)$, $\text{Finish}[P_0 - P_4] = \text{false}$; as all processes are allocated with resources.

1) $i = P_0$, $\text{Request}_i \leq \text{work}$.

$$(0, 0, 0) \leq (0, 0, 0)$$

$$\text{Work} = \text{Work} + \text{Allocation}_i$$

$$= (0, 0, 0) + (0, 1, 0) = (0, 1, 0)$$

2) $i = P_2$, $(0, 0, 0) < (0, 1, 0)$

$$\text{Work} = (0, 1, 0) + (3, 0, 3) = (3, 1, 3)$$

$$\text{Finish}[P_2] = \text{true}$$

3) $i = P_3$, $(1, 0, 0) < (3, 1, 3)$

$$\text{Work} = (3, 1, 3) + (2, 1, 1) = (5, 2, 4)$$

$$\text{Finish}[P_3] = \text{true}$$

4) $i = P_4$, $(0, 0, 2) < (5, 2, 4)$

$$\text{Work} = (5, 2, 4) + (0, 0, 2) = (5, 2, 6)$$

$$\text{Finish}[P_4] = \text{true}$$

5) $i = P_1$, $(2, 0, 2) < (5, 2, 6)$

$$\text{Work} = (5, 2, 6) + (2, 0, 0) = (7, 2, 6)$$

$$\text{Finish}[P_1] = \text{true}$$

As $\text{Finish}[P_0 - P_4] = \text{true}$, system is in safe state when the requested resources are allocated.

The safe sequence is $\langle P_0, P_2, P_3, P_4, P_1 \rangle$

7.8 Recovery From Deadlock

- There are two basic approaches to recovery from deadlock:
 1. Process Termination
 2. Resource Preemption

7.8.1 Process Termination

- Two basic approaches to terminate processes:
 - **Abort all deadlocked processes.** This method breaks the deadlock cycle by terminating all the processes involved in deadlock. But, at a great expense, the deadlocked processes may have computed for a long time, and the results of these partial computations must be discarded and probably will have to be recomputed later.
 - **Abort one process at a time until the deadlock cycle is eliminated.** This method incurs considerable overhead, since, after each process is aborted, a deadlock-detection algorithm must be invoked to determine whether any processes are still deadlocked.
- In the latter case there are many factors that can go into deciding which processes to terminate next:
 1. Process priorities.
 2. How long the process has been running, and how close it is to finishing.
 3. How many and what type of resources is the process holding. (Are they easy to preempt and restore?)
 4. How many more resources does the process need to complete.
 5. How many processes will need to be terminated
 6. Whether the process is interactive or batch.
 7. (Whether or not the process has made non-restorable changes to any resource.)

7.7.2 Resource Preemption

- When preempting resources to relieve deadlock, there are three important issues to be addressed:
 1. **Selecting a victim** - Deciding which resources to preempt from which processes involves many of the same decision criteria outlined above.
 2. **Rollback** - Ideally one would like to roll back a preempted process to a safe state prior to the point at which that resource was originally allocated to the process. Unfortunately it can be difficult or impossible to determine what such a safe state is, and so the only safe rollback is to roll back all the way back to the beginning. (i.e. abort the process and make it start over.)
 3. **Starvation** – There are chances that the same resource is picked from processes as a victim, every time the deadlock occurs and this continues. This is starvation. A count can be kept on number of rollback of a process and the process has to be a victim for finite number of times only.

