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 realease the resource held by another process is called deadlock.

4.2 SYSTEM MODEL

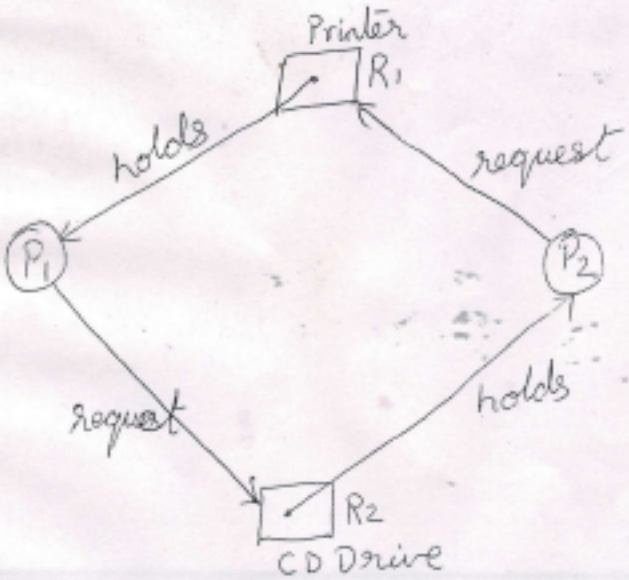
- A system consists of finite number of resources and is distributed among number of processes.
- A process must request a resource before using it and it must release the resource after using it. It can request any number of resources to carry out a designated task. The amount of resource requested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

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

1. Request:-If the request is not granted immediately then the requesting process must wait until it can acquire the resources.
2. Use:-The process can operate on the resource.
3. 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 Pi currently holds a printer R1 and a process Pj holds the tape drive R2. If process Pi request a tape drive and process Pj 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 must be 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 {P0,P1.....Pn} of waiting process must exist such that P0 is waiting for a resource held by P1, P1 is waiting for a resource held by P2. Pn-1 is waiting for resource held by process Pn and Pn is waiting for the resource held by P1.

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={P1,P2.....Pn} i.e., set consisting of all active processes, represented by circle.
2. R={R1,R2.....Rn} 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 :

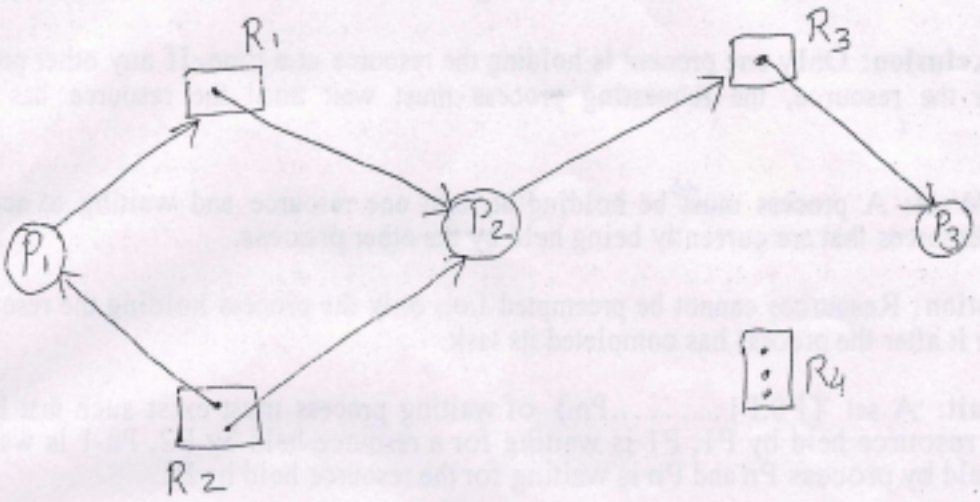
1. A directed edge from process Pi to resource type Rj denoted by Pi→Ri indicates that Pi requested an instance of resource Rj and is waiting. This edge is called **Request edge**.
2. A directed edge Ri Pj signifies that resource Rj is held by process Pi. This is called **Assignment edge/ Allocation edge.**

The sets P, *R*, and E:

P={P1,P2,P3}
R= {R1,R2,R3,R4}
E= {P1→R1, P2→P3, R1→P2, R2→P2, R2→P1, R3→P3 }

Resource instances:

- One instance of resource type R1
- Two instances of resource type R2
- One instance of resource type *R3*
- Three instances of resource type *R4*



- 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

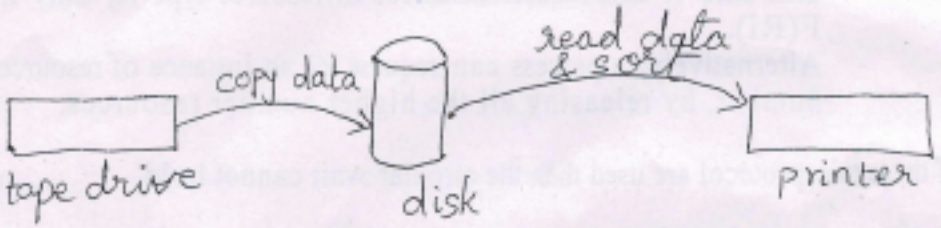
For a deadlock to occur each of the four necessary conditions must hold. If at least one of the conditions does not hold then we can prevent occurrence of deadlock.

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={R1,R2,.....Rn} 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→N as follows : where the value 'N' indicates the no. of instances of the resource.

F(tape drive)=1
F(disk drive)=5
F(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 Ri initially and then, it can request instances of resource type Rj only if F(Rj) > F(Ri).
- 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(Printer)>F(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:

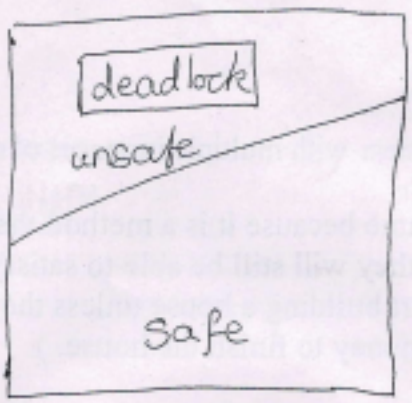
A safe state is a state in which there exists at least one order in which all the process will run completely without resulting in a deadlock.

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

A sequence of processes <P1,P2,.....Pn> is a safe sequence for the current allocation state

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

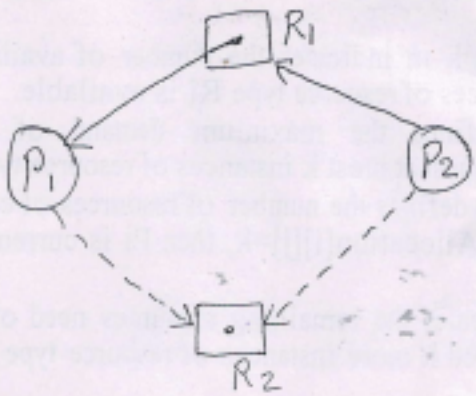
- If the resources that Pi requests are not currently available then Pi can obtain all of its needed resource to complete its designated task.
- A safe state is not a deadlock state.
- Whenever a process request a resource i.e., currently available, the system must decide whether resources can be allocated 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 claimedge is used. *For eg*:-A claim edge Pi---- >Rj indicates that process Pi may request Rj in future. The claim edge is represented by a dotted line.



Process P_1 & P_2 has claim edge to R_2 ⇒ P_1 & P_2 may request for R_2 in future

When a process Pi requests the resource Rj, the claim edge is converted to a request edge.
When resource Rj is released by process Pi, the assignment edge Rj→Pi is replaced by the

claim edge Pi----- >Rj.

When a process Pi requests resource Rj the request is granted only if converting the request edge Pi→Rj to as assignment edge Rj-→Pi 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 Available[i]=k, then k instances of resource type Rj is available.
- **Max**:-An n*m matrix defines the maximum demand of each process if Max[i][j]=k, then Pi may request at most k instances of resource type Rj.
- **Allocation**:-An n*m matrix defines the number of resources of each type currently allocated to each process. If Allocation[i][j]=k, then Pi is currently k instances of resource type Rj.
- **Need**:-An n*m matrix indicates the remaining resources need of each process. If Need[i][j]=k, then Pi may need k more instances of resource type Rj to compute its task.

So Need[i][j]=Max[i][j] – 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,.....n

Step 2. Find i such that both Finish[i]=false and Need i <= 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*n*n operation to decide whether a state is safe.

Resource Request Algorithm:

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. When a request for resources is made by process Pi the following actions are taken.

1) If Request(i) <= Need(i) go to step 2
otherwise raise an error condition since the process has exceeded its maximum claim.

2) If Request(i) <= Available go to step 3
otherwise Pi must wait. Since the resources are not available.

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)

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

Consider the following snapshot of a system.

Process	Allocation			Maximum			Available			← 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 P_1 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 [P_0 - P_4] = false.

1) i = P_0; Need[P_0] ≤ Work
[7, 4, 3] ≤ [3, 3, 2] ✗
P_0 cannot execute now.

2) i = P_1, Need[P_1] ≤ Work
[1, 2, 2] ≤ [3, 3, 2]
P_1 executes...
Work = Work + Allocation[P_1].
= [3, 3, 2] + [2, 0, 0] = [5, 3, 2]
Finish [P_1] = true

3) i = P_2, Need[P_2] ≤ Work
[6, 0, 0] ≤ [5, 3, 2] ✗
P_2 cannot execute now.

4) i = P_3, Need[P_3] ≤ Work.
[0, 1, 1] ≤ [5, 3, 2] ✓
P_3 executes → Work = Work + Allocation[P_3]
= [5, 3, 2] + [2, 1, 1] = [7, 4, 3] Finish

5) $i = P_4$, $Need[P_4] \leq Work$

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

P_4 executes

$Work = Work + Allocation[P_4]$

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

$= [7,4,5]$.

$Finish[P_4] = true$.

6) $i = P_0$, $Need[P_0] \leq Work$

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

→ So executes

$Work = Work + Allocation[P_0]$

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

$Finish[P_0] = true$.

7) $i = P_2$, $Need[P_2] \leq Work$

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

P_2 executes.

$Work = Work + Allocation[P_2]$

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

$Finish[P_2] = true$.

$Finish[P_0 - P_4] = true$, All the processes in the s
can be executed. So the system is in safe s
The safe sequence is $< P_1, P_3, P_4, P_0, P_2 >$

ii) P_1 request $\rightarrow [1,0,2]$

1) $Request_{[P_1]} \leq Need_{[P_1]}$

$[1,0,2] \leq [1,2,2]$ ✓

Requested resource is needed.

2) $Request_{[P_1]} \leq Available$

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

Requested resource is available.

3) Assume that resources is allocated to P_1, t

the changes are -

$Allocation_{[P_1]} = Allocation_{[P_1]} + Request_{[P_1]}$

$Allocation_{[P_1]} = [2,0,0] + [1,0,2] = [3,0,2]$

$Need_{[P_1]} = Need_{[P_1]} - Request_{[P_1]}$

$Need_{[P_1]} = [1,2,2] - [1,0,2] = [0,2,0]$

$Available = Available - Request_{[P_1]}$

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

Now, with these changes safety algorithm is ch

to find if the system is in safe state.

	Allocation			Maximum			Need		
	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

Available $[2,3,0]$

Work = [2, 3, 0] Finish [P0 - P4] = false

1) i = P0, Need[P0] ≤ Work
[7, 4, 3] ≤ [2, 3, 0] ✗
P0 cannot execute now.

2) i = P1, Need[P1] ≤ Work
[0, 2, 0] ≤ [2, 3, 0] ✓
P1 executes.
Work = Work + Allocation[P1]
= [2, 3, 0] + [3, 0, 2] = [5, 3, 2]
Finish[P1] = true

3) i = P2, Need[P2] ≤ Work
[6, 0, 0] ≤ [5, 3, 2] ✗
P2 cannot execute now.

4) i = P3, Need[P3] ≤ Work
[0, 1, 1] ≤ [5, 3, 2] ✓
P3 executes.
Work = Work + Allocation[P3]
= [5, 3, 2] + [2, 1, 1] = [7, 4, 3]
Finish[P3] = true.

5) i = P4, Need[P4] ≤ Work
[4, 3, 1] ≤ [7, 4, 3] ✓
P4 executes
Work = Work + Allocation[P4]
= [7, 4, 3] + [0, 0, 2] = [7, 4, 5].
Finish[P4] = true

6) i = P0, Need[P0] ≤ Work
[7, 4, 3] ≤ [7, 4, 5] ✓
Work = [7, 4, 5] + [0, 1, 0] = [7, 5, 5]
Finish[P0] = true.