Deadlocks

In a multiprogramming environment several processes may compete for a finite no. of resources. A process requests resources, and if the resources are not available at that time, the process enters a waiting state, because the resources it has requested are held by other waiting processes. This situation is called a deadlock.

There are four necessary conditions hold simultaneously in a system:

(i) Mutual exclusion :- Only one process at a time can use the resource. If another process requests that resource, the requesting process must be delayed until the resource has been released.

(ii) Hold and wait :- A process must be holding at least one resource and waiting to acquire additional resources that are currently being held by other processes.

(iii) No preemption :- Resources cannot be preempted; that is a resource can be released by the process

holding it, after that process has completed its task.

(iv) Circular wait :- One Process is waiting for a resource that is held by other Process, other Process is waiting for a resource that is held by next process & soon.

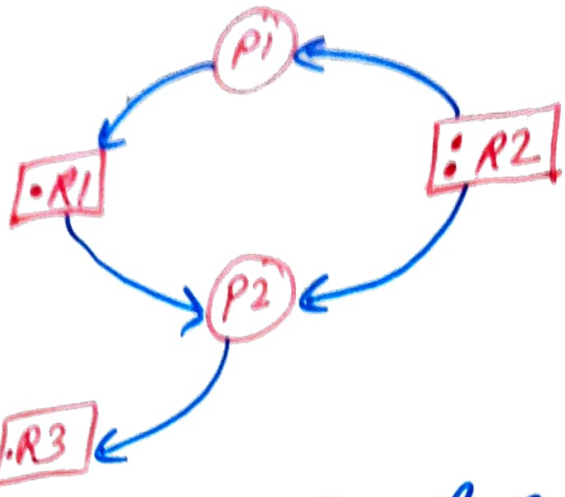
All four condition must hold for a deadlock to occur.

Resource Allocation Graph :- Deadlock are represented in terms of a directed graph called a system resource allocation graph.

* A directed edge $P_i \rightarrow R_j$ is called request edge
* A directed edge $P_i \leftarrow R_j$ is called an assignment edge or R_j has been allocates to Process P_i.
* Graphically each process P_i as a circle and each resource type R_j as a rectangle. since resource type R_j may have more than one instance, it represent each instance as a dot with in the rectangle.

single Instance

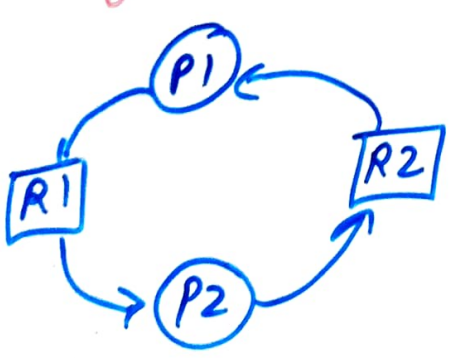
Two instances



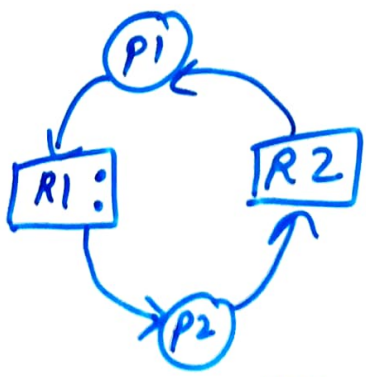
Given the definition of a resource allocation graph, it can be shown that if the graph contains no cycles, then no process in the system is deadlocked. If the graph does contain a cycle, then a deadlock may exist.

Note :- (1) If each resource type has exactly one instance, then a cycle implies that a deadlock has occurred.

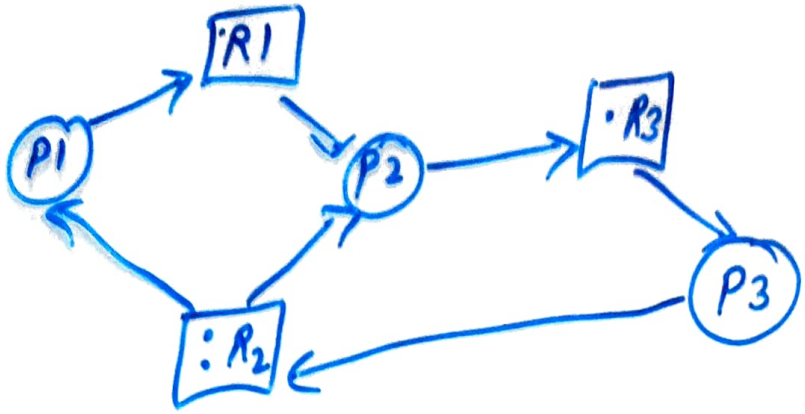
(2) If each resource type has several instances, then a cycle does not necessarily imply that a deadlock has occurred.



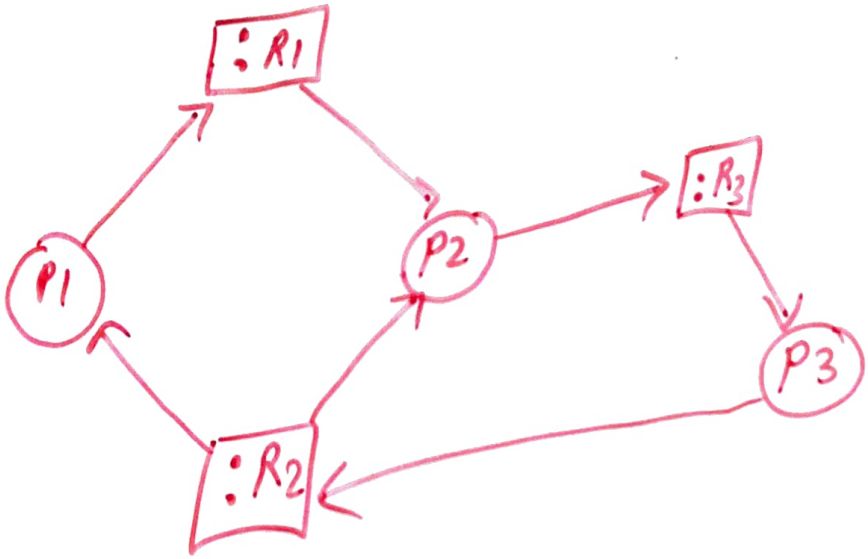
(a)



(b)



(a)



(b)

METHODS FOR HANDLING DEADLOCKS:-

We can deal with the deadlock problem in one of three ways:

(i) Ensure that the system will never enter a deadlock state. For this we have two Techniques :-

1.1 Deadlock Prevention

1.2 Deadlock Avoidance

(ii) Allow system to enter into the deadlock state, detect it and recover.

(iii) we can ignore the problem altogether and pretend that deadlocks never occur in the system. [It is used by most OS including UNIX and Windows]

(i) Deadlock Prevention :- is a set of methods for ensuring that at least one of the necessary conditions cannot hold.

Mutual Exclusion :- The mutual exclusion condition must hold for nonshareable resources. for example - printer. In general, we cannot prevent deadlocks by denying the mutual exclusion condition, because non sharable cannot be made to shareable.

Hold and wait :- ensure that hold & wait never occurs in the system, we must guarantee that, whenever a process requests a resource, it does not hold any other resources.

Two methods to break the condition:

I^st Method :-

Allocate all the resources to a process before it begins execution.

II^nd Method :-

Allow a process to request resources only when it should not be holding any resource.

Disadvantages :-

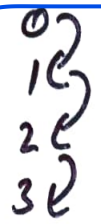
In first method, resource utilization may be low, since resources may be allocated but unused for a long period.

In second method, starvation is possible. A process that needs several popular resources may have to wait indefinitely, because at least one of the resources that it needs is always allocated to some other process.

NO PREEMPTION :- To ensure that this condition does not hold, we can use the following protocol. If a process is holding some resources and requests another resource that cannot be

immediately allocated to it, then all the resources currently being held are preempted. In other words these resources are implicitly released.

CIRCULAR WAIT :- Each process can request resources only in an increasing order.



That is, a process can initially request any no. of instances of a resource type - R_i. After that, the process can request instances of resource type R_j if and only if $F(R_j) > F(R_i)$. If several instances of the same resource type are needed, a single request for all of them must be issued.

X — X — X — X — X — X

DEADLOCK AVOIDANCE :- A deadlock-avoidance algo. dynamically examines the resource allocation state to ensure that a circular-wait condition can never exist. The resource allocation state is defined by the number of available

and allocated resources and the maximum demands of the processes.

Safe State :- A state is safe if the system can allocate resources to each process in some order and still avoid a deadlock. More formally, a system is in a safe state only if there exists a safe sequence.

A safe state is not a deadlocked state. Conversely, a deadlocked state is an unsafe state. Not all unsafe states are deadlocks. An unsafe state may lead to a deadlock. As long as the state is safe, the OS cannot prevent processes from requesting resources such that a deadlock occurs. The behavior of the processes controls unsafe states.

Consider a system with 12 magnetic tape drives and 3 processes:

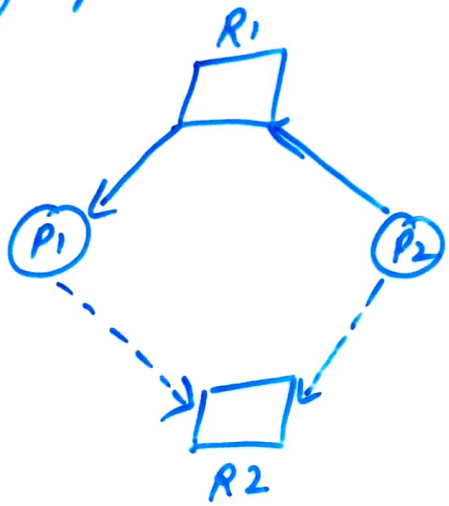
	Maximum Needs	Current Needs
P_0	10	5
P_1	4	2
P_2	9	2

given the concept of a safe state, we can define avoidance algorithms that ensure that the system will never deadlock. The idea is simple to ensure that the system will always remain in a safe state. Initially, the system is in a safe state. Whenever a process requests a resource that is currently available, the system must decide whether the resource can be allocated immediately or whether the process must wait. The request is granted only if the allocation leaves the system in a safe state.

Resource Allocation Graph Algorithm :- In addition it introduce a new type of edge, called a claim edge. A claim edge $P_i \rightarrow R_j$ indicates that process P_i may request resource R_j at some time in the future. claim edge represented by a dashed line. When a process P_i requests resource R_j the claim edge $P_i \rightarrow R_j$ is converted to a request edge. similarly, when a resource R_j is released by P_i, the assignment edge $R_j \rightarrow P_i$ is converted to a claim edge $P_i \rightarrow R_j$. The resources must be claimed a priori in the

in the system. That is, before process P_i starts executing all its claim edges must already appear in the resource allocation graph.

Suppose that process P_i requests resource R_j. The request can be granted only if converting the request edge $P_i \rightarrow R_j$ to an assignment edge $R_j \rightarrow P_i$ does not result in the formation of a cycle in the resource allocation graph.



Banker's Algorithm :- When a new process enters the system, it must declare the maximum number of instances of each resource type that it may need. This number may not exceed the total number of resources in the system. When a user requests a set of resources, the system must determine whether the allocation of these resources will leave the system in

a safe state.

Banker's Algorithm :-

1. Initialization
   set finish[i] = false for all i ranging from 1 to n
2. compute avail & need
   set need[i,j] = max[i,j] - alloc[i,j]
   for all process i & for all resources j
3. Find a process i such that
   finish[i] = false &
   need[i,j] ≤ avail[j] for all resources
   if no such i exists, goto step 5
4. Schedule process i for execution so that it can utilize resources, then free up its resources by doing following
   set Avail[j] = Avail[j] + alloc[i,j]
   set need[i,0] = 0 for all j
   set finish[i] = true
   Goto step 3
5. If finish[i] = True for all i return [safe]
   else return [unsafe]

Problems of Banker Algorithm :-

(i) The bankers algo assumes that total number of resources should remain constant. This cannot be made possible since a resource have to undergo maintenance number of times.

(ii) It assumes that total no. of process in a system should remain constant. In case of multiprogramming environment it is quite difficult to maintenance constant flow of Processes.

(iii) The banker algorithm assumes that each & every process should declare the maximum amount of resources, that requires during its life time, but it is not possible in dynamic environment. So it is quite difficult for a process to declare its resources in advance.

x - x - x - x - x - x - x - x - x

Deadlock Detection & Recovery :- In this environment the system must provide :

⇒ An algorithm that examines the state of the system to determine whether a deadlock has occured.

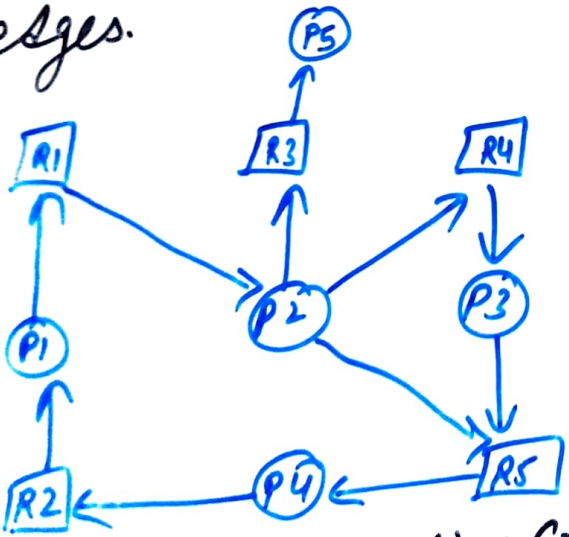
⇒ An algorithm to recover from the deadlock.

We elaborate on these two requirements:

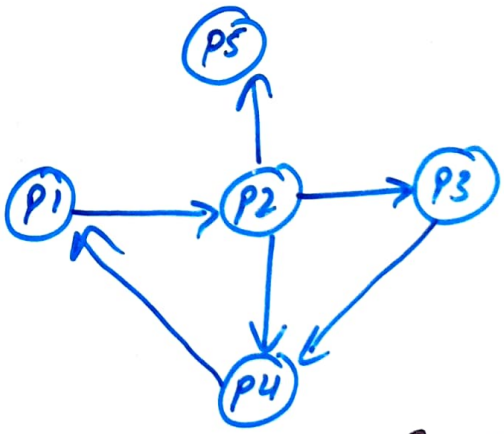
→ When the System with only single instance of each resource type

⇒ System with several instances of each resource type.

(i) Resource have single instance :- If all resources have only a single instance then we have to make a graph known as wait for graph. It is used to determine whether the system is in deadlock or not. In wait for graph resources are not displayed, only processes are displayed. So we obtain this graph by removing the resources nodes & collapsing the appropriate edges.



Resource Allocation Graph



Wait for Graph

A deadlock exists in the System if and only if the wait for graph contains a cycle.

(ii) Resources having several instances :- The wait for graph scheme is not applicable to a resource allocation system with multiple instances of each resource type. The Algorithm employs several time varying data structures that are similar to those used in the banker's algorithm.

(i) If alloc[i,j] ≠ 0 then
set finish[i] = false;

else
set finish[i] = True;

for all process i = 1 to n
for all resource j = 1 to n

(ii) Find a process i such that the conditions
finish[i] = false and
Request[i,j] ≤ avail[j]
If no such i exists go to step 4

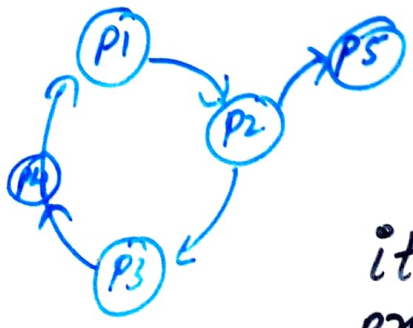
(iii) set avail[j] = avail[j] + alloc[i,j]
set finish[i] = True;
goto step 2

(iv) If finish[i] = false for some process then system is in deadlock state.

Recovery from Deadlock :- There are two options for breaking a deadlock. one is simply to abort one or more processes to break the circular wait. The other is to preempt some resources from one or more of the deadlocked processes.

(1) PROCESS TERMINATION :- To eliminate deadlocks by aborting a process, we use one of two methods.

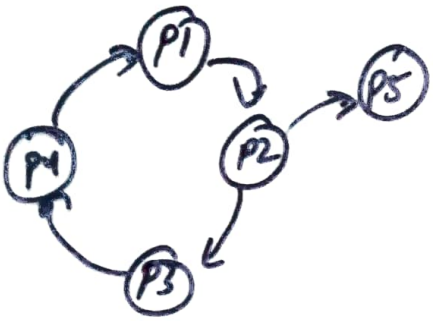
(1.1) Abort one process at a time until the deadlock cycle is eliminated :- This method incurs considerable overhead, since after each process is aborted, a deadlock detection algorithm must be invoked to determine whether any processes are still deadlocked.



Choose 1 process for for P2 to abort, check system is deadlock state or not. If it is then we move further process to abort.

(1.2) Abort all deadlocked processes :- This method clearly with break the deadlock cycle, but at 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 all processes at a same time. All processes need to be restart again & CPU time they have used till now will get wasted.

Best to use :- Abort 1 process at a time.

2] RESOURCE PREEMPTION :- To eliminate deadlocks using resource preemption, we can successively preempt some resources from processes and give these resources to other processes until the deadlock cycle is broken.

If preemption is required to deal with deadlocks, then three issues need to be addressed :

(2.1) Selecting a victim :- which resources and which processes are to be preempted? As in process termination, we must determine the order of preemption to minimize cost. Cost factors may include such parameters as the number of resources a deadlocked process is holding

and the amount of time the process has thus far consumed during its execution.

2.2) Roll Back :- If we preempt a resource from a process, what should be done with that process? Clearly, it cannot continue with its normal execution; it is missing some needed resource. we must roll back the process to some safe state & restart it from that state.

Since, in general, it is difficult to determine what a safe state is, the simplest solution is a total roll back: Abort the process and then restart it.

2.3) Starvation :- How do we ensure that starvation will not occur? In a system where victim selection is based primarily on cost factors, it may happen that the same process is always picked as a victim. As a result, this process never completes its designated task, a starvation situation that needs to be dealt with in any practical system. Clearly, we must ensure that a process can be picked as a victim only a finite number of times.

Combined Approach to Deadlock handling :-

It has been argued that none of the presented approaches is suitable for handling of deadlocks in a complex system. Instead, deadlock prevention, avoidance and detection can be combined for maximum effectiveness. So combine the three basic approaches :-

⇒ prevention, avoidance, detection allowing the use of the optimal approach for each of resources in the system.

⇒ Partition resources into hierarchically ordered classes.

⇒ Use most appropriate technique for handling deadlocks within each class.

Consider a system with the following classes of resources ordered in a sequence.

(i) Swapping space an area of secondary storage for backing up blocks of main memory.

(ii) Job resources such as printers & drives with removable media

(iii) Main Memory such as pages or segments

(iv) Internal resources such as I/O channels & slots of the pool of dynamic memory

The following strategies may be applied to individual resource classes.

Swapping space :- Prevention, Avoidance is also possible but deadlock detection is not, since there is no backup of the swapping store.

Job resource :- Avoidance, prevention is also a possibility, but detection & recovery are undesirable due to the possibility of modification of files that belong to this class of resources.

Main Memory :- with swapping, prevention by means of preemption is a reasonable choice. Avoidance is undesirable because of its runtime overhead. Deadlock detection is possible.

Internal system resources :- Prevention is probably the best as due to runtime overhead of deadlock avoidance, even of detection can hardly be tolerated.