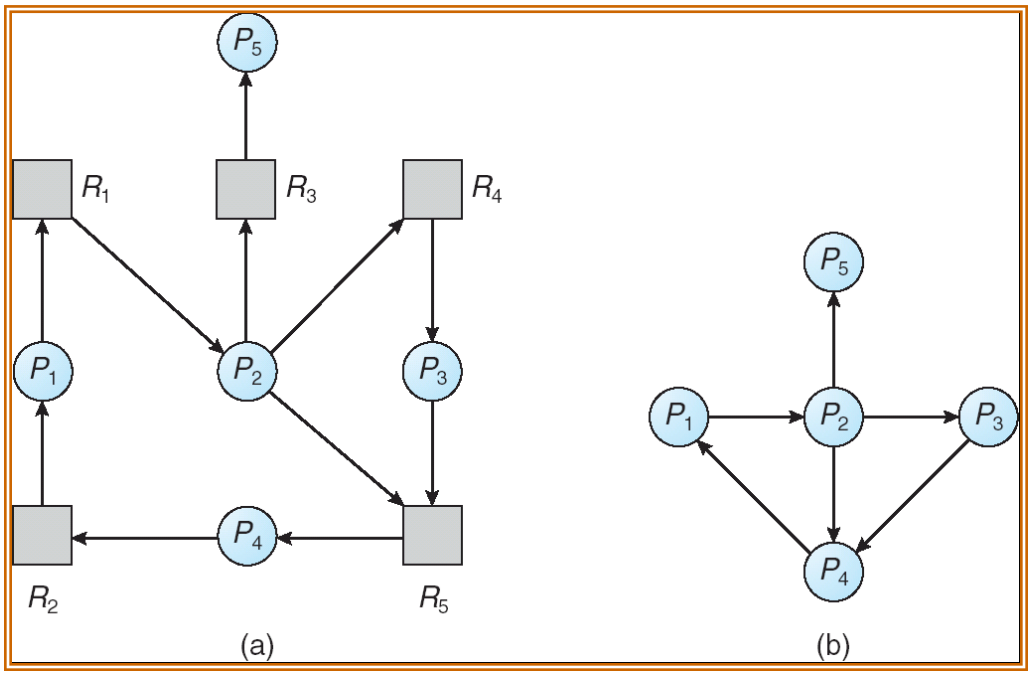
Deadlock Detection

If a system does not employ either a deadlock-prevention or a deadlock avoidance algorithm, then it should provide an algorithm that examines the state of the system to determine whether a deadlock has occurred and an algorithm to recover from the deadlock.

Single Instance of Each Resource Type

If all resources have only a single instance, then we can define a deadlock detection algorithm that uses a variant of the resource-allocation graph, called a *wait-for* graph. We obtain this graph from the resource-allocation graph by removing the resource nodes and collapsing the appropriate edges.

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



(a) Resource-allocation graph (b) Corresponding wait-for graph

If there exists a cycle in wait-for graph, there is a deadlock in the system, and the processes forming the part of cycle are blocked in the deadlock. To take appropriate action to recover from this situation, an algorithm needs to be called periodically to detect existence of cycle in wait-for graph.

Multiple Instances of a Resource Type

When multiple instances of a resource type exist, the wait-for graph becomes inefficient to detect the deadlock in the system. An algorithm which uses certain data structures similar to ones used in Banker's algorithm is applied.

The following data structures are used:

Available Resources - A vector of size m stores information about the number of available resources of each type.

Current Allocation - A matrix of order ***n x m*** stores information about the number of resources of each type allocated to each process.

Request - An nxm matrix indicates the current request of each process. If Request[i][j] equals k, then process P_i is requesting k more instances of resource type R_j

1. Let *Work* and *Finish* be vectors of length *m* and *n*, respectively. Initialize:

Work=*Available.* For *i*= 1,2,…,*n*, if $Allocation_i \neq 0$, then *Finish*[i] = false; otherwise, *Finish*[i] = *true*.

2. Find an index *i* such that both:

(a) *Finish*[*i*] == *false*

(b) $Request_i \leq Work$

If no such *i* exists, go to step 4.

3.Work=*Work*+$Allocation_i$

Finish[*i*]=*true*

go to step 2.

4. If *Finish*[*i*] == false, for some *i*, $1 \leq i \leq n$, then the system is in deadlock state. Moreover, if *Finish*[*i*] == *false*, then P_i is deadlocked.

Deadlock Recovery

When a detection algorithm determines that a deadlock exists, then,

- let the operator deal with the deadlock manually.
- let the system recover from the deadlock automatically.

There are two options for breaking a deadlock:

- to abort one or more processes to break the circular wait.
- to preempt some resources from one or more of the deadlocked processes.

Process Termination:

There are two methods that can be used for terminating the processes to recover from the deadlock:

- Terminating one process at a time until the circular-wait condition is eliminated-It involves an overhead of invoking a deadlock detection algorithm after terminating each process to detect whether circular-wait condition is eliminated or not, that is, whether any processes are still deadlocked.
- Terminating or Aborting all processes involved in the deadlock- This method will definitely ensure the recovery of a system from the deadlock. The disadvantage of this method is that many processes may have executed for a long time; close to their completion. As a result, the computations performed till the time of termination are discarded.

Resource Preemption:

An alternate method to recover system from the state of deadlock is to preempt the resources from the processes one by one and allocate them to other processes until the circular-wait condition is eliminated.

Steps involved in the preemption of resources from processes are:

1. Select a process for preemption - The choice of resources and processes must be such that they incur minimum cost to the system.
2. Roll back of the process - After preempting the resources, the corresponding process must be rolled back properly so that it does not leave the system in an inconsistent state. Process must be brought to some safe state from where it can be restarted later. In case no such safe state can be achieved, the process must be totally rolled back. Partial rollback is preferred.
3. Prevent starvation - In case of the selection of a process is based on the cost factor, it is quite possible that same process is selected repeatedly for the rollback leading to the situation of starvation. This can be avoided by including the number of rollbacks of a given process in the cost factor.

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 must be dealt with in any practical system.