# Deadlock:

# Deadlock:

In a multiprogramming environment several processes may compete for a finite number of resources. A process request resources; if the resource is available at that time a process enters the wait state. Waiting process may never change its state because the resources requested are held by other waiting process. This situation is known as deadlock.

## Example

* System has 2 disk drives.
* P1 and P2 each hold one disk drive and each needs another one.
* 2 train approaches each other at crossing, both will come to full stop and neither shall start until other has gone.

* Traffic only in one direction.
* Each section of a bridge can be viewed as a resource.
* If a deadlock occurs, it can be resolved if one car backs up (preempt resources and rollback).
* Several cars may have to be backed up if a deadlock occurs.
* Starvation is possible

## System Model:

A system consists of a finite number of resources to be distributed among a number of competing processes. The resources are partitioned into several types each of which consists of a number of identical instances. A process may utilized a resources in the following sequence

* + **Request:** In this state one can request a resource.
  + **Use:** In this state the process operates on the resource.
  + **Release:** In this state the process releases the resources.

**Deadlock Characteristics:** In a deadlock process never finish executing and system resources are tied up. A deadlock situation can arise if the following four conditions hold simultaneously in a system.

* **Mutual Exclusion:** At a time only one process can use the resources. If another process requests that resource, requesting process must wait until the resource has been released.
* **Hold and wait:** A process must be holding at least one resource and waiting to additional resource that is currently held by other processes.
* **No Preemption:** Resources allocated to a process can’t be forcibly taken out from it unless it releases that resource after completing the task.
* **Circular Wait:** A set {P0, P1, …….Pn} of waiting state/ process must exists such that P0 is waiting for a resource that is held by P1, P1 is waiting for the resource that is held by P2 ….. P(n – 1) is waiting for the resource that is held by Pn and Pn is waiting for the resources that is held by P4.

## Resource Allocation Graph:

Deadlock can be described more clearly by directed graph which is called system resource allocation graph. The graph consists of a set of vertices ‘V’ and a set of edges ‘E’. The set of vertices ‘V’ is partitioned into two different types of nodes such as P = {P1, P2, …….Pn}, the set of all the active processes in the system and R = {R1, R2, …….Rm}, the set of all the resource type in the system. A directed edge from process Pi to resource type Rj is denoted by Pi → Rj. It signifies that process Pi is an instance of resource type Rj and waits for that resource. A directed edge from resource type Rj to the process Pi which signifies that an instance of resource type Rj has been allocated to process Pi. A directed edge Pi → Rj is called as request edge and Rj → Pi is called as assigned edge.

* Process
* Resource Type with 4 instances



* Pirequests instance of Rj

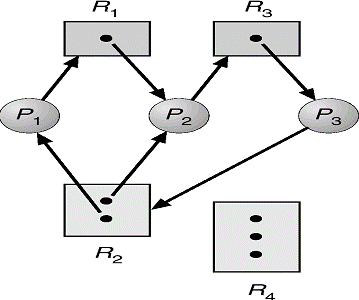
*Pi*

* Pi is holding an instance of Rj

*Pi*

When a process Pi requests an instance of resource type Rj then a request edge is inserted as resource allocation graph. When this request can be fulfilled, the request edge is transformed to an assignment edge. When the process no longer needs access to the resource it releases the resource and as a result the assignment edge is deleted. The resource allocation graph shown in below figure has the following situation.

* The sets P, R, E
* P = {P1, P2, P3}
* R = {R1, R2, R3, R4}
* E = {P1 → R1,P2 → R3,R1 → P2,R2 → P2,R2 → P1,R3 → P3} The resource instances are
  + Resource R1 has one instance
  + Resource R2 has two instances.
  + Resource R3 has one instance
  + Resource R4 has three instances.

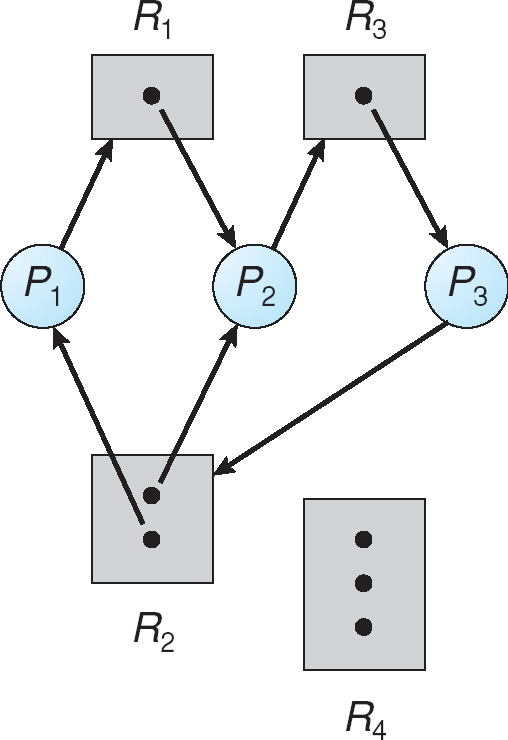


The process states are:

* + Process P1 is holding an instance of R2 and waiting for an instance of R1.
  + Process P2 is holding an instance of R1 and R2 and waiting for an instance R3.
  + Process P3 is holding an instance of R3.

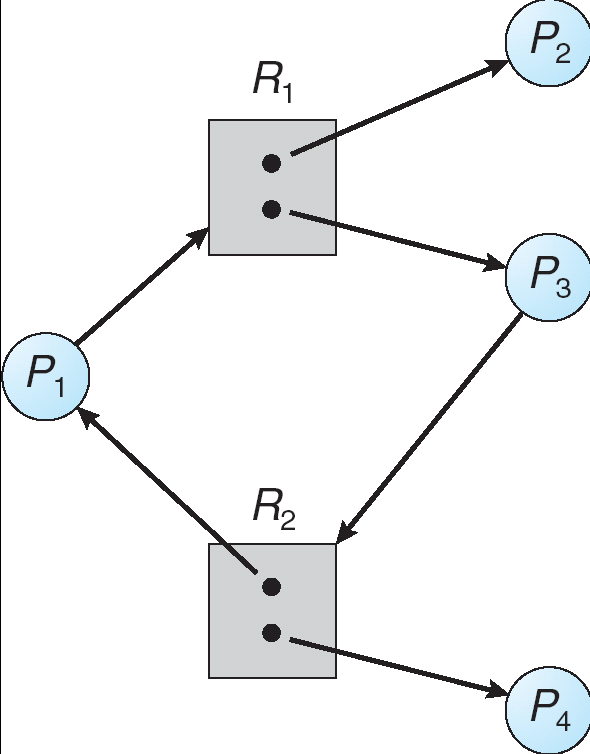
The following example shows the resource allocation graph with a deadlock.

* + P1 -> R1 -> P2 -> R3 -> P3 -> R2 -> P1
  + P2 -> R3 -> P3 -> R2 -> P1



The following example shows the resource allocation graph with a cycle but no deadlock.

* + P1 -> R1 -> P3 -> R2 -> P1
  + No deadlock
  + P4 may release its instance of resource R2
  + Then it can be allocated to P3



# Methods for Handling Deadlocks

The problem of deadlock can deal with the following 3 ways.

* We can use a protocol to prevent or avoid deadlock ensuring that the system will never enter to a deadlock state.
* We can allow the system to enter a deadlock state, detect it and recover.
* We can ignore the problem all together.

To ensure that deadlock never occur the system can use either a deadlock prevention or deadlock avoidance scheme.

## Deadlock Prevention:

Deadlock prevention is a set of methods for ensuring that at least one of these necessary conditions cannot hold.

* **Mutual Exclusion:** The mutual exclusion condition holds for non sharable. The example is a printer cannot be simultaneously shared by several processes. Sharable resources do not require mutual exclusive access and thus cannot be involved in a dead lock. The example is read only files which are in sharing condition. If several processes attempt to open the read only file at the same time they can be guaranteed simultaneous access.
* **Hold and wait:**To ensure that the hold and wait condition never occurs in the system, we must guaranty that whenever a process requests a resource it does not hold any other resources. There are two protocols to handle these problems such as one protocol that can be used requires each process to request and be allocated all its resources before it begins execution. The other protocol allows a process to request resources only when the process has no resource. These protocols have two main disadvantages. First, resource utilization may be low, since many of the resources may be allocated but unused for a long period. Second, 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, a protocol is used. If a process is holding some resources and request another resource that cannot be immediately allocated to it. The preempted one added to a list of resources for which the process is waiting. The process will restart only when it can regain its old resources, as well as the new ones that it is requesting. Alternatively if a process requests some resources, we first check whether they are available. If they are, we allocate them. If they are not available, we check whether they are allocated to some other process that is waiting for additional resources. If so, we preempt the desired resources from the waiting process and allocate them to the requesting process. If the resources are not either available or held by a waiting process, the requesting process must wait.
* **Circular Wait:**We can ensure that this condition never holds by ordering of all resource type and to require that each process requests resource in an increasing order of enumeration. Let R

= {R1, R2, …….Rn}be the set of resource types. We assign to each resource type a unique integer number, which allows us to compare two resources and to determine whether one precedes another in our ordering. Formally, we define a one to one function F: R  N, where N is the set of natural numbers. For example, if the set of resource types R includes tape drives, disk drives and printers, then the function F might be defined as follows:

F (Tape Drive) = 1, F (Disk Drive) = 5, F (Printer) = 12.

We can now consider the following protocol to prevent deadlocks: Each process can request resources only in an increasing order of enumeration. That is, a process can initially request any number of instances of a resource type, say Ri. After that, the process can request instances of resource type Rj if and only if F (Rj) > F (Ri). If several instances of the same resource type are needed, defined previously, a process that wants to use the tape drive and printer at the same time must first request the tape drive and then request the printer.

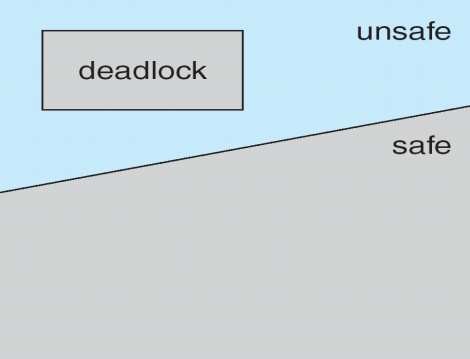
# Deadlock Avoidance

Requires additional information about how resources are to be used.Simplest and most useful model requires that each process declare the maximum number of resources of each type that it may need.The deadlock-avoidance algorithm dynamically examines the resource-allocation state to ensure that there can never be a circular-wait condition.Resource-allocation state is defined by the number of available and allocated resources, and the maximum demands of the processes.

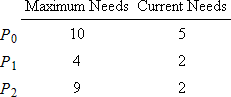
## Safe State

When a process requests an available resource, system must decide if immediate allocation leaves the system in a safe state.Systems are in safe state if there exists a safe sequence of all process. A sequence <P1, P2, …, Pn> of ALL the processes is the system such that for each Pi, the resources that Pi can still request can be satisfied by currently available resources + resources held by all the Pj, withj <i.That is:

* If Pi resource needs are not immediately available, then Pi can wait until all Pjhave finished.
* When Pj is finished, Pi can obtain needed resources, execute, return allocated resources, and terminate.
* When Pi terminates, Pi +1 can obtain its needed resources, and so on.
* If system is in safe state => No deadlock
* If system in not in safe state => possibility of deadlock
* OS cannot prevent processes from requesting resources in a sequence that leads to deadlock
* Avoidance => ensue that system will never enter an unsafe state, prevent getting into deadlock

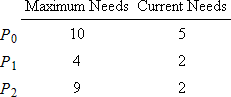


## Example:



* + Suppose processes P0, P1, and P2 share 12 magnetic tape drives
  + Currently 9 drives are held among the processes and 3 are available
  + Question: Is this system currently in a safe state?
  + Answer: Yes!

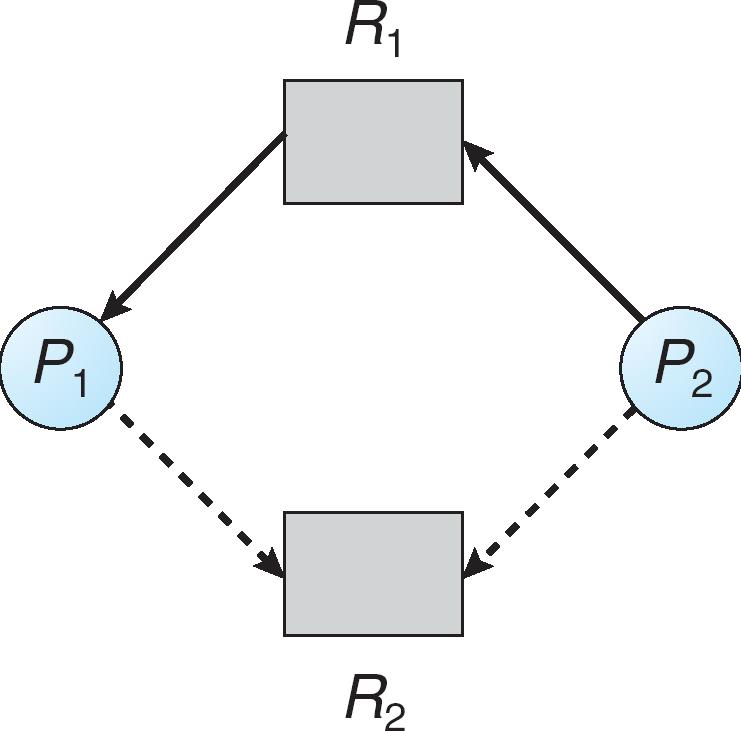
o Safe Sequence: <P1, P0, P2>



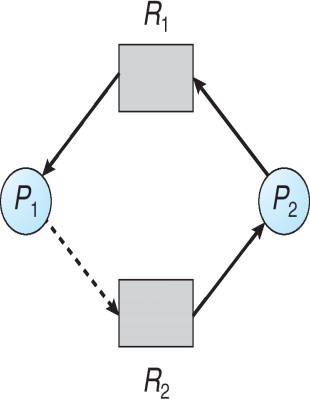
* + Suppose process P2 requests and is allocated 1 more tape drive.
  + Question: Is the resulting state still safe?
  + Answer: No! Because there does not exist a safe sequence anymore.
    - Only P1 can be allocated its maximum needs.
    - IFP0 and P2 request 5 more drives and 6 more drives, respectively, then the resulting state will be deadlocked.

## Resource Allocation Graph Algorithm

In this graph a new type of edge has been introduced is known as claim edge. Claim edge PiRj indicates that process Pj may request resource Rj; represented by a dashed line.Claim edge converts to request edge when a process requests a resource.Request edge converted to an assignment edge when the resource is allocated to the process.When a resource is released by a process, assignment edge reconverts to a claim edge.Resources must be claimed a priori in the system.



* P2 requesting R1, but R1 is already allocated to P1.
* Both processes have a claim on resource R2
* What happens if P2 now requests resource R2?



* Cannot allocate resource R2 to process P2
* Why? Because resulting state is unsafe
  + P1 could request R2, thereby creating deadlock!

Use only when there is a single instance of each resource type

* + Suppose that process Pi requests a resource Rj
  + The request can be granted only if converting the request edge to an assignment edge does not result in the formation of a cycle in the resource allocation graph.
  + Here we check for safety by using cycle-detection algorithm.

## Banker’s Algorithm

This algorithm can be used in banking system to ensure that the bank never allocates all its available cash such that it can no longer satisfy the needs of all its customer. This algorithm is applicable to a system with multiple instances of each resource type. When a new process enter in to 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. Several data structure must be maintained to implement the banker’s algorithm.

Let,

* + - n = number of processes
    - m = number of resources types
* **Available**: Vector of length m. If Available[j] = k, there are k instances of resource type Rjavailable.
* **Max**: n x m matrix. If Max [i,j] = k, then process Pimay request at most k instances of resource type Rj.
* **Allocation**: n x m matrix. If Allocation[i,j] = k then Pi is currently allocated k instances of Rj.
* **Need**: n x m matrix. If Need[i,j] = k, then Pi may need k more instances of Rjto complete its task.

Need [i,j] = Max[i,j] – Allocation [i,j].

## Safety Algorithm

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

Finish [i] = false for i = 0, 1, …,n- 1.

1. Find and i such that both:
2. Finish [i] = false
3. NeediWork

If no such i exists, go to step 4.

1. Work = Work + Allocationi Finish[i] = true

go to step 2.

1. If Finish [i] == true for all i, then the system is in a safe state.

## Resource Allocation Algorithm

Request = request vector for process Pi. If Requesti[j] = k then process Pi wants k instances of resource type Rj.

1. If RequestiNeedigo to step 2. Otherwise, raise error condition, since process has exceeded its maximum claim.
2. If RequestiAvailable, go to step 3. Otherwise Pi must wait, since resources are not available.
3. Pretend to allocate requested resources to Pi by modifying the state as follows: Available = Available – Request;

Allocationi= Allocationi + Requesti; Needi=Needi – Requesti;

* + If safe  the resources are allocated to Pi.
  + If unsafe  Pi must wait, and the old resource-allocation state is restored Example
  + 5 processes P0 through P4;
  + 3 resource types:
* A (10 instances), B (5instances), and C (7 instances).
  + Snapshot at time T0:

|  |  |  |  |
| --- | --- | --- | --- |
|  | 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 2 | 9 0 2 |  |
| P3 | 2 1 1 | 2 2 2 |  |
| P4 | 0 0 2 | 4 3 3 |  |

* + - The content of the matrix Need is defined to be Max – Allocation. Need

A B C

|  |  |
| --- | --- |
| P0 | 7 4 3 |
| P1 | 1 2 2 |
| P2 | 6 0 0 |
| P3 | 0 1 1 |

P4 4 3 1

* + - The system is in a safe state since the sequence <P1, P3, P4, P2, P0> satisfies safety criteria. P1 requests (1, 0, 2)
    - Check that Request  Available (that is, (1,0,2)  (3,3,2)  true.

|  |  |  |  |
| --- | --- | --- | --- |
| Allocation | Need | Available | |
|  | A B C | A B C A B C | |
| P0 | 0 1 0 | 7 4 3 | 2 3 0 |
| P1 | 3 0 2 | 0 2 0 |  |
| P2 | 3 0 1 | 6 0 0 |  |
| P3 | 2 1 1 | 0 1 1 |  |
| P4 | 0 0 2 | 4 3 1 |  |

* + - Executing safety algorithm shows that sequence <P1, P3, P4, P0, P2> satisfies safety requirement.
    - Can request for (3,3,0) by P4 be granted? –NO
    - Can request for (0,2,0) by P0 be granted? –NO (Results Unsafe)

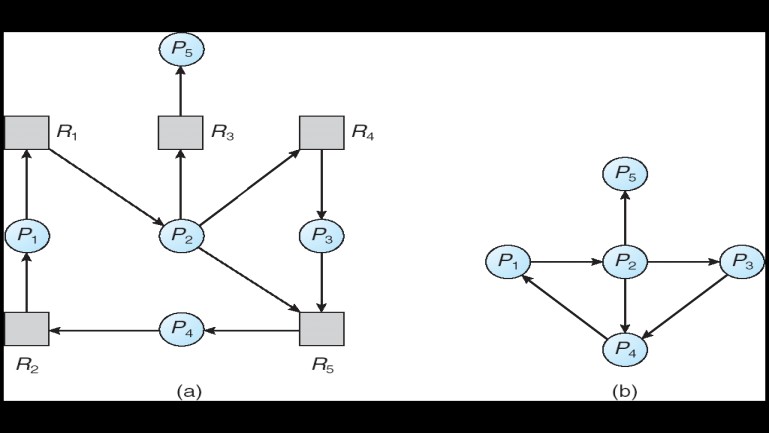
# Deadlock Detection

If a system doesn’t employ either a deadlock prevention or deadlock avoidance, then deadlock situation may occur. In this environment the system must provide

* An algorithm to recover from the deadlock.
* An algorithm to remove the deadlock is applied either to a system which pertains single in instance each resource type or a system which pertains several instances of a resource type.

## Single Instance of each Resource type

If all resources only a single instance then we can define a deadlock detection algorithm which uses a new form of resource allocation graph called “Wait for graph”. We obtain this graph from the resource allocation graph by removing the nodes of type resource and collapsing the appropriate edges. The below figure describes the resource allocation graph and corresponding wait for graph.



Resource-Allocation Graph

* + For single instance

Correspondin wait-for graph

* + Pi ->Pj(Pi is waiting for Pj to release a resource that Pi needs)
  + Pi->Pj exist if and only if RAG contains 2 edges Pi ->Rq and Rq ->Pj for some resource Rq

## Several Instances of a Resource type

The wait for graph scheme is not applicable to a resource allocation system with multiple instances of reach resource type. For this case the algorithm employs several data structures which are similar to those used in the banker’s algorithm like available, allocation and request.

* + **Available**: A vector of length m indicates the number of available resources of each type.
  + **Allocation**: An n x m matrix defines the number of resources of each type currently allocated to each process.
  + **Request**: An n x m matrix indicates the current request of each process. If Request [ij] = k, then process Pi is requesting k more instances of resource type. Rj.

1. Let Work and Finish be vectors of length m and n, respectively Initialize:
2. Work = Available
3. For i = 1,2, …, n, if Allocationi 0, then Finish[i] = false;otherwise, Finish[i] = true.
4. Find an index i such that both:
5. Finish[i] == false
6. RequestiWork

If no such i exists, go to step 4.

1. Work = Work + Allocation Finish [i] = true

Go to step 2

1. If Finish [i] = false, for some i, 1 i n, then the system is in a deadlock state. Moreover, if Finish

[i] = false, then process Pi is deadlocked.

# Recovery from Deadlock

When a detection algorithm determines that a deadlock exists, several alternatives exist. One possibility is to inform the operator that a deadlock has occurred, and to let the operator deal with the deadlock manually. The other possibility is to let the system recover from the deadlock automatically. There are two options for breaking a deadlock. One solution is simply to abort one or more processes to break the circular wait. The second option is to preempt some resources from one or more of the deadlocked processes.

## Process Termination:

To eliminate deadlocks by aborting a process, we use one of two methods. In both methods, the system reclaims all resources allocated to the terminated processes.

* + **Abort all deadlocked processes:** This method clearly will break the deadlock cycle, but at a great expense; these processes may have computed for a long time, and the results of these partial computations must be discarded and probably 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.

## Resource Preemption:

To eliminate deadlocks using resource preemption, we 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.

* + **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 numbers of resources a deadlock process is holding, and the amount of time a deadlocked process has thus far consumed during its execution.
  + **Rollback:** 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 rollback the process to some safe state, and restart it from that state.
  + **Starvation:** 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 small finite number of times. The most common solution is to include the number of rollbacks in the cost factor.