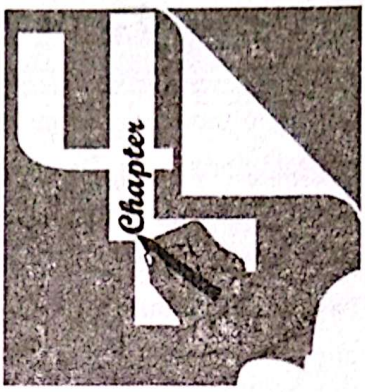


PROCESS DEADLOCKS

CHAPTER OUTLINE

After comprehensive study of this chapter, you will be able to:

- System Model, System Resources: Preemptable and Non-Preemptable
- Conditions for Resource Deadlocks
- Deadlock Modeling, The OSTRICH Algorithm,
- Method of Handling Deadlocks,
- Deadlock Prevention,
- Deadlock Avoidance: Banker's Algorithm, Deadlock Detection: Resource Allocation Graph,
- Recovery from Deadlock.

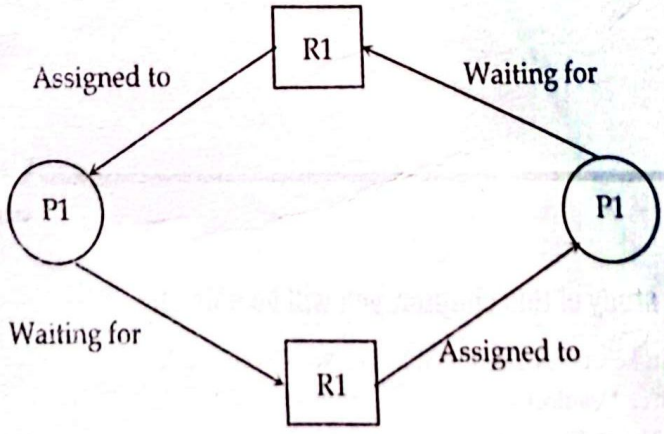
SYSTEM MODEL

A system model or structure consists of a fixed number of resources to be circulated among some opposing processes. The resources are then partitioned into numerous types, each consisting of some specific quantity of identical instances. Memory space, CPU cycles, directories and files, I/O devices like keyboards, printers and CD-DVD drives are prime examples of resource types. When a system has 2 CPUs, then the resource type CPU got two instances.Under the standard mode of operation, any process may use a resource in only the below-mentioned sequence:

- **Request**: When the request can't be approved immediately (where the case may be when another process is utilizing the resource), then the requesting job must remain waited until it can obtain the resource.
- **Use**: The process can run on the resource (like when the resource is a printer, its job/process is to print on the printer).
- **Release**: The process releases the resource (like, terminating or exiting any specific process)

INTRODUCTION TO DEADLOCK

Deadlock is a situation where a set of processes are blocked because each process is holding a resource and waiting for another resource acquired by some other process. Consider an example when two trains are coming toward each other on same track and there is only one track, none of the trains can move once they are in front of each other. Similar situation occurs in operating systems when there are two or more processes hold some resources and wait for resources held by other. For example, in the below diagram, Process 1 is holding Resource 1 and waiting for resource 2 which is acquired by process 2, and process 2 is waiting for resource 1.



Now neither program can proceed until the other program releases a resource. The operating syste cannot know what action to take. At this point the only alternative is to abort (stop) one of th programs.

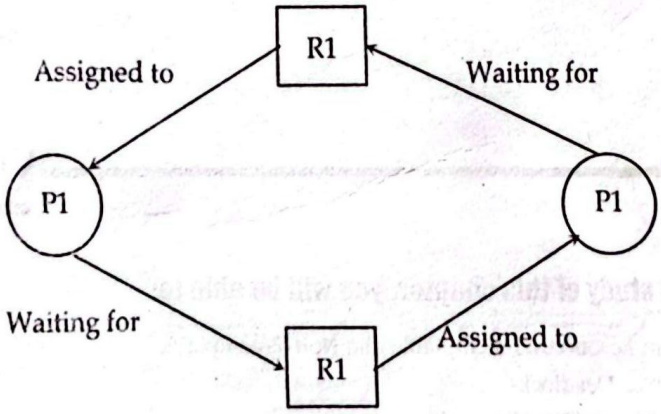
SYSTEM MODEL

A system model or structure consists of a fixed number of resources to be circulated among some opposing processes. The resources are then partitioned into numerous types, each consisting of some specific quantity of identical instances. Memory space, CPU cycles, directories and files, I/O devices like keyboards, printers and CD-DVD drives are prime examples of resource types. When a system has 2 CPUs, then the resource type CPU got two instances.Under the standard mode of operation, any process may use a resource in only the below-mentioned sequence:

- **Request**: When the request can't be approved immediately (where the case may be when another process is utilizing the resource), then the requesting job must remain waited until it can obtain the resource.
- **Use**: The process can run on the resource (like when the resource is a printer, its job/process is to print on the printer).
- **Release**: The process releases the resource (like, terminating or exiting any specific process)

INTRODUCTION TO DEADLOCK

Deadlock is a situation where a set of processes are blocked because each process is holding a resource and waiting for another resource acquired by some other process. Consider an example when two trains are coming toward each other on same track and there is only one track, none of the trains can move once they are in front of each other. Similar situation occurs in operating systems when there are two or more processes hold some resources and wait for resources held by other. For example, in the below diagram, Process 1 is holding Resource 1 and waiting for resource 2 which is acquired by process 2, and process 2 is waiting for resource 1.



Now neither program can proceed until the other program releases a resource. The operating system cannot know what action to take. At this point the only alternative is to abort (stop) one of the programs.

SYSTEM RESOURCES: PREEMPTABLE AND NONPREEMPTABLE RESOURCES

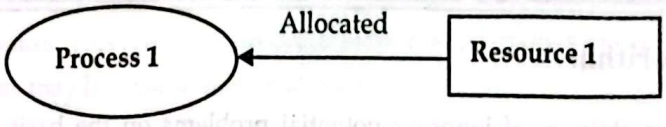
Resources come in two flavors: preemptable and nonpreemptable. A preemptable resource is one that can be taken away from the process with no ill effects. Memory is an example of a preemptable resource. On the other hand, a nonpreemptable resource is one that cannot be taken away from process (without causing ill effect). For example, CD resources are not preemptable at an arbitrary moment. Reallocating resources can resolve deadlocks that involve preemptable resources. Deadlocks that involve nonpreemptable resources are difficult to deal with

DEADLOCK CHARACTERIZATION (CONDITIONS FOR RESOURCE DEADLOCKS)

In a deadlock, processes never finish executing, and system resources are tied up, preventing other jobs from starting. Various features that characterize deadlock are listed below:

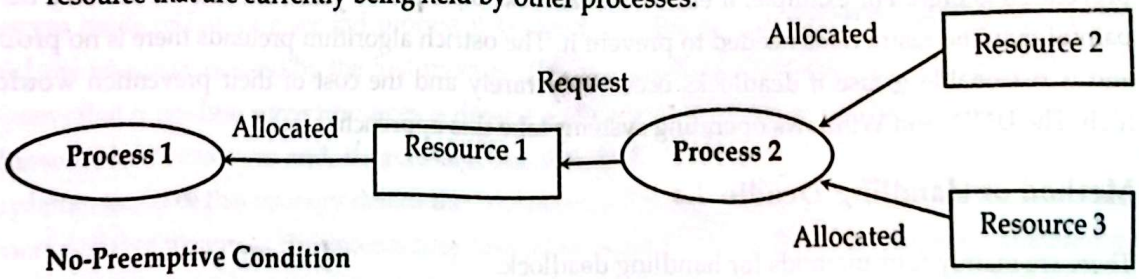
a. Mutual Exclusion Condition

The resources involved are non-shareable, At least one resource must be held in a non-shareable mode, i.e. only one process at a time claims exclusive control of the resource. If another process requests that resource, the requesting process must be delayed until the resource has been released.



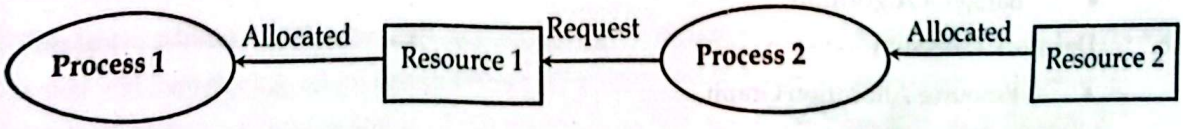
b. Hold and Wait Condition

Requesting process hold already, resources while waiting for requested resources, there must exist a process that is holding a resource already allocated to it while waiting for additional resource that are currently being held by other processes.



c. No-Preemptive Condition

Resources already allocated to a process cannot be preempted, Resources cannot be removed from the processes are used to completion or released voluntarily by the process holding it.



d. **Circular Wait Condition**

All the processes must wait for the resource in a cyclic manner where the last process waits for the resource held by the first process.

A set {P0 P1, P2... Pn} of waiting processes must exist such that P1 is waiting for a resource that is held by P1, P1 is waiting for a resource that is held by P2, ..., Pn-1 is waiting for a resource that is held by Pn and Pn is waiting for a resource that is held by P0. The processes in the system form a circular list or chain where each process in the list is waiting for a resource held by the next process in the list.

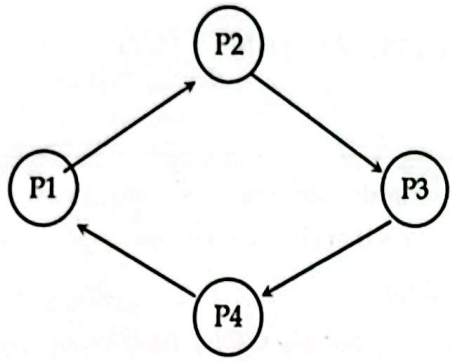


Fig: 4.1: Circular waits

DEADLOCK MODELING

The OSTRICH Algorithm

The ostrich algorithm is a strategy of ignoring potential problems on the basis that they may be exceedingly rare. It is named for the ostrich effect which is defined as "to stick one's head in the sand and pretend there is no problem". It is used when it is more cost-effective to allow the problem to occur than to attempt its prevention. This approach may be used in dealing with deadlocks in concurrent programming if they are believed to be very rare and the cost of detection or prevention is high. For example, if each PC deadlocks once per 10 years, the one reboot may be less painful than the restrictions needed to prevent it. The ostrich algorithm pretends there is no problem and is reasonable to use if deadlocks occur very rarely and the cost of their prevention would be high. The UNIX and Windows operating systems take this approach.

Method of Handling Deadlocks

There are mainly four methods for handling deadlock.

a. Deadlock Prevention

b. Deadlock Avoidance
   - Banker's Algorithm

c. Deadlock Detection
   - Resource Allocation Graph

d. Recovery from Deadlock

a. **Deadlock Prevention**

It means that we design such a system where there is no chance of having a deadlock. We can prevent deadlock by eliminating any of the above four condition.

Elimination of Mutual Exclusion Condition

The mutual exclusion condition must hold for non-sharable resources. That is, several processes cannot simultaneously share a single resource. This condition is difficult to eliminate because some resources, such as the tap drive and printer, are inherently non-shareable. Note that shareable resources like read-only-file do not require mutually exclusive access and thus cannot be involved in deadlock.

Elimination of Hold and Wait Condition

There are two possibilities for elimination of the second condition. The first alternative is that a process request be granted all of the resources it needs at once, prior to execution. The second alternative is to disallow a process from requesting resources whenever it has previously allocated resources. This strategy requires that all of the resources a process will need must be requested at once. The system must grant resources on "all or none" basis. If the complete set of resources needed by a process is not currently available, then the process must wait until the complete set is available. While the process waits, however, it may not hold any resources. Thus the "wait for" condition is denied and deadlocks simply cannot occur. This strategy can lead to serious waste of resources. For example, a program requiring ten tap drives must request and receive all ten derives before it begins executing. If the program needs only one tap drive to begin execution and then does not need the remaining tap drives for several hours. Then substantial computer resources (9 tape drives) will sit idle for several hours. This strategy can cause indefinite postponement (starvation). Since not all the required resources may become available at once.

Elimination of No-preemption Condition

The non-preemption condition can be alleviated by forcing a process waiting for a resource that cannot immediately be allocated to relinquish all of its currently held resources, so that other processes may use them to finish. Suppose a system does allow processes to hold resources while requesting additional resources. Consider what happens when a request cannot be satisfied. A process holds resources a second process may need in order to proceed while second process may hold the resources needed by the first process. This is a deadlock. This strategy requires that when a process that is holding some resources is denied a request for additional resources. The process must release its held resources and, if necessary, request them again together with additional resources. Implementation of this strategy denies the "no-preemptive" condition effectively **high Cost** when a process release resources the process may lose all its work to that point. One serious consequence of this strategy is the possibility of indefinite postponement (starvation). A process might be held off indefinitely as it repeatedly requests and releases the same resources.

Elimination of Circular Wait Condition

The last condition, the circular wait, can be denied by imposing a total ordering on all of the resource types and then forcing, all processes to request the resources in order (increasing or decreasing). This strategy imposes a total ordering of all resources types, and to require that each process requests

resources in a numerical order (increasing or decreasing) of enumeration. With this rule, the resource allocation graph can never have a cycle. For example, provide a global numbering of all the resources, as shown

1 ≡ Card reader

2 ≡ Printer

3 ≡ Plotter

4 ≡ Tape drive

5 ≡ Card punch

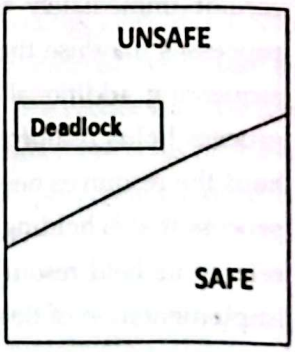
Now the rule is this: processes can request resources whenever they want to, but all requests must be made in numerical order. A process may request first printer and then a tape drive (order: 2, 4), but it may not request first a plotter and then a printer (order: 3, 2). The problem with this strategy is that it may be impossible to find an ordering that satisfies everyone.

b. **Deadlock Avoidance**

This approach to the deadlock problem anticipates deadlock before it actually occurs. This approach employs an algorithm to access the possibility that deadlock could occur and acting accordingly. This method differs from deadlock prevention, which guarantees that deadlock cannot occur by denying one of the necessary conditions of deadlock.If the necessary conditions for a deadlock are in place, it is still possible to avoid deadlock by being careful when resources are allocated. So named because the process is analogous to that used by a banker in deciding if a loan can be safely made.

A deadlock avoidance algorithm dynamically examines the resources allocation state to ensure that a circular wait condition case never exists. Where, the resources allocation state is defined by available and allocated resources and the maximum demand of the process. There are 3 states of the system:

- Safe state
- Unsafe state and
- Deadlock state



Safe and unsafe state

When a system can allocate the resources to the process in such a way so that they still avoid deadlock then the state is called safe state. When there is a safe sequence exit then we can say that the system is in the safe state.

A sequence is in the safe state only if there exists a safe sequence. A sequence of process P1, P2, Pn is a safe sequence for the current allocation state if for each Pi the resources request that Pi can still make can be satisfied by currently available resources pulls the resources held by all Pj with j<i.

If a safe sequence does not exist, then the system is in an unsafe state, which may lead to deadlock. All safe states are deadlock free, but not all unsafe states lead to deadlocks.

Example: Let's take four customers with available is 2

Customers	Used	Max	
A	1	6	Available
B	1	5	Units = 2
C	2	4	
D	4	7	

The key to a state being safe is that there is at least one way for all users to finish. In other analogy, the state of above table is safe because with 2 units left, the banker can delay any request except C's, thus letting C finish and release all four resources. With four units in hand, the banker can let either D or B have the necessary units and so on. Thus safe sequence is <C, D, B, A>

Consider what would happen if a request from B for one more unit were granted in above table.We would have following situation:

Customers	Used	Max	
A	1	6	Available
B	2	5	Units = 1
C	2	4	
D	4	7	

If all the customers namely *A, B, C,* and *D* asked for their maximum loans, then banker could not satisfy any of them and we would have a deadlock. Thus this is an unsafe state.

Methods for deadlock avoidance

There is mainly following Bankers algorithm is used for deadlock avoidance:

Banker's Algorithm

The Banker algorithm, sometimes referred to as the detection algorithm, is a resource allocation and deadlock avoidance algorithm that tests for safety by simulating the allocation of predetermined maximum possible amounts of all resources, and then makes an "s-state" check to test for possible deadlock conditions for all other pending activities, before deciding whether allocation should be allowed to continue. Following 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.

Available

- It is a one dimensional array of size 'm' indicating the number of available resources of each type.
- Available[j] = k means there are 'k' instances of resource type R_j

Max

- It is a 2-dimensional array of size 'n*m' that defines the maximum demand of each process in a system.
- Max[i, j] = k means process P_i may request at most 'k' instances of resource type R_j

Allocation

- It is a 2-dimensional array of size 'n*m' that defines the number of resources of each type currently allocated to each process.
- Allocation[i, j] = k means process Pi is currently allocated 'k' instances of resource type Rj

Need

- It is a 2-dimensional array of size 'n*m' that indicates the remaining resource need of each process.
- Need[i, j] = k means process Pi currently need 'k' instances of resource type Rjfor its execution.
- Need[i, j] = Max[i, j] – Allocation[i, j]

Banker's resource request Algorithm

Step 1: If request ≤ need then go to step 2

Else error

Step 2: if request ≤available then go to step 3

Else wait

Step 3: Available = available – Request

Allocation = Allocation + Request

Need = Need – Request

Step 4: Check new state is safe or not?

Bankers Safety Algorithm

Step 1: If need ≤ Available then

Execute Process

Calculate new available as,

Available= Available + Allocation

Step 2: Otherwise

Do not execute and go forward

Numerical problem 1: Assume that there are 5 processes, P0 through P4, and 4 types of resources. At T0 we have the following system state:

	Allocation				Max Need				Available			
	A	B	C	D	A	B	C	D	A	B	C	D
P0	0	1	1	0	0	2	1	0	1	5	2	0
P1	1	2	3	1	1	6	5	2				
P2	1	3	6	5	2	3	6	6				
P3	0	4	3	2	0	6	5	2				
P4	0	0	1	4	0	6	5	6				
	2	12	18	12								

Create the need matrix (max-allocation)

	Need matrix(max-allocation)			
	A	B	C	D
P0	0	1	0	0
P1	0	4	2	1
P2	1	0	0	1
P3	0	0	2	0
P4	0	6	4	2

Use the safety algorithm to test if the system is in a safe state.

We will first define work and finish:

Work vector	Finish matrix	
1	P0	FALSE
5	P1	FALSE
2	P2	FALSE
0	P3	FALSE
	P4	FALSE

Check to see if need0 for process P0 is (0, 1, 0, 0) is less than or equal to work. It is, so let's set finish to true for that process and also update work by adding the allocated resources (0, 1, 1, 0) for that process to work.

Work vector	Finish matrix	
1	P0	TRUE
6	P1	FALSE
3	P2	FALSE
0	P3	FALSE
	P4	FALSE

Now, let's check to see if need1 (0, 4, 2, 1) <= work. Remember that we have to check each element of the vector need1 against the corresponding element in work. Because 1 is not less than 0 (the fourth element), we need to move on to P2.

Need2 (1, 0, 0, 1) is not less than work, so must move on to P3.

Need3 (0, 0, 2, 0) is less than work, so we can update work and finish.

Work vector	Finish matrix	
1	P0	TRUE
12	P1	FALSE
6	P2	FALSE
2	P3	TRUE
	P4	FALSE

Next, let's look at P4. Need4 (0, 6, 4, 2) is less than work, so we can update work and finish as follows:

Work vector	Finish matrix	
1	P0	TRUE
12	P1	FALSE
7	P2	FALSE
6	P3	TRUE
	P4	TRUE

Now we can go back up to P1. Need1 (0, 4, 2, 1) is less than work, so let's update work and finish.

Work vector	Finish matrix	
2	P0	TRUE
14	P1	TRUE
10	P2	FALSE
7	P3	TRUE
	P4	TRUE

Finally, let's look at P2. Need2 (1, 0, 0, 1) is less than work, so we can then say that the system is in a safe state and the processes will be executed in the following order:

{P0, P3, P4, P1, P2}

1. If the system is in a safe state, can the following requests be granted, why or why not? Please also run the safety algorithm on each request as necessary.

   a) P1 requests (2, 1, 1, 0)

   We cannot grant this request, because we do not have enough available instances of resource A.

   b) P1 requests (0, 2, 1, 0)

   There are enough available instances of the requested resources, so first let's pretend to accommodate the request and see what the system looks like:

	Allocation				Max				Available			
	A	B	C	D	A	B	C	D	A	B	C	D
P0	0	1	1	0	0	2	1	0	1	3	1	0
P1	1	4	4	1	1	6	5	2				
P2	1	3	6	5	2	3	6	6				
P3	0	6	3	2	0	6	5	2				
P4	0	0	1	4	0	6	5	6				

Similarly calculate need matrix as

	Need matrix(max-allocation)			
	A	B	C	D
P0	0	1	0	0
P1	0	2	1	1
P2	1	0	0	1
P3	0	0	2	0
P4	0	6	4	2

Now we need to run the safety algorithm:

Work vector	Finish matrix	
1	P0	FALSE
3	P1	FALSE
1	P2	FALSE
0	P3	FALSE
	P4	FALSE

Let's first look at P0. Need0 (0, 1, 0, 0) is less than work, so we change the work vector and finish matrix as follows:

Work vector	Finish matrix	
1	P0	TRUE
4	P1	FALSE
2	P2	FALSE
0	P3	FALSE
	P4	FALSE

Need1 (0, 2, 1, 1) is not less than work, so we need to move on to P2.

Need2 (1, 0, 0, 1) is not less than work, so we need to move on to P3.

Need3 (0, 0, 2, 0) is less than or equal to work. Let's update work and finish:

Work vector	Finish matrix	
1	P0	TRUE
10	P1	FALSE
5	P2	FALSE
2	P3	TRUE
	P4	FALSE

Let's take a look at Need4 (0, 6, 4, 2). This is less than work, so we can update work and finish:

Work vector	Finish matrix	
1	P0	TRUE
10	P1	FALSE
6	P2	FALSE
6	P3	TRUE
	P4	TRUE

We can now go back to P1. Need1 (0, 2, 1, 1) is less than work, so work and finish can be updated:

Work vector	Finish matrix	
1	P0	TRUE
14	P1	FALSE
10	P2	FALSE
7	P3	TRUE
	P4	TRUE

Finally, Need2 (1, 0, 0, 1) is less than work, so we can also accommodate this. Thus, the system is in a safe state when the processes are run in the following order: {P0, P3, P4, P1, P2}. We therefore can grant the resource request.

Numerical problem 2: Assume that there are three resources, A, B, and C. There are 4 processes P0 to P3. At T0 we have the following snapshot of the system:

	Allocation			Max			Available		
	A	B	C	A	B	C	A	B	C
P0	1	0	1	2	1	1	2	1	1
P1	2	1	2	5	4	4			
P2	3	0	0	3	1	1			
P3	1	0	1	1	1	1			

1. Create the need matrix (max-allocation)

	Need matrix(max-allocation)		
	A	B	C
P0	1	1	0
P1	3	3	2
P2	0	1	1
P3	0	1	0

Is the system in a safe state? Why or why not?

In order to check this, we should run the safety algorithm. Let's create the work vector and finish matrix:

Work vector	Finish matrix	
2	P0	FALSE
1	P1	FALSE
1	P2	FALSE
	P3	FALSE

Need0 (1, 1, 0) is less than work, so let's go ahead and update work and finish:

Work vector	Finish matrix	
3	P0	TRUE
1	P1	FALSE
2	P2	FALSE
	P3	FALSE

Need1 (3, 3, 2) is not less than work, so we have to move on to P2.

Need2 (0, 1, 1) is less than work, let's update work and finish:

Work vector	Finish matrix	
6	P0	TRUE
1	P1	FALSE
2	P2	TRUE
	P3	FALSE

Need3 (0, 1, 0) is less than work, we can update work and finish:

Work vector	Finish matrix	
7	P0	TRUE
1	P1	FALSE
3	P2	TRUE
	P3	TRUE

We now need to go back to P1. Need1 (3, 3, 2) is not less than work, so we cannot continue. Thus, the system is not in a safe state.

Problem with Banker's Algorithm

- An algorithm requires fixed number of resources; some processes dynamically change the number of resources.
- An algorithm requires the number of resources in advance; it is very difficult to predict the resources in advanced.
- Algorithms predict all process returns within finite time, but the system does not guarantee it.

It is important to note that an unsafe state does not imply the existence or even the eventual existence a deadlock. What an unsafe state does imply is simply that some unfortunate sequence of events might lead to a deadlock. The Banker's algorithm is thus to consider each request as it occurs, and see if granting it leads to a safe state. If it does, the request is granted, otherwise, it postponed until later.

c. **Deadlock Detection**

Deadlock detection is the process of actually determining that a deadlock exists and identifying the processes and resources involved in the deadlock. The basic idea is to check allocation against resource availability for all possible allocation sequences to determine if the system is in deadlocked state a. Of course, the deadlock detection algorithm is only half of this strategy. Once a deadlock is detected, there needs to be a way to recover several alternatives exists:

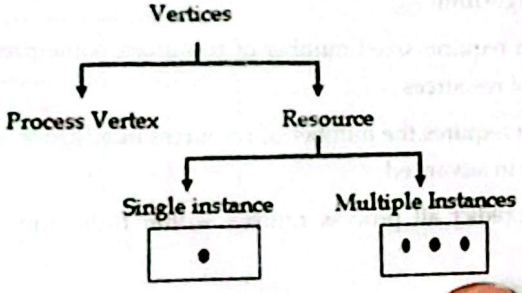
- Temporarily prevent resources from deadlocked processes.
- Back off a process to some check point allowing preemption of a needed resource and restarting the process at the checkpoint later.
- Successively kill processes until the system is deadlock free.

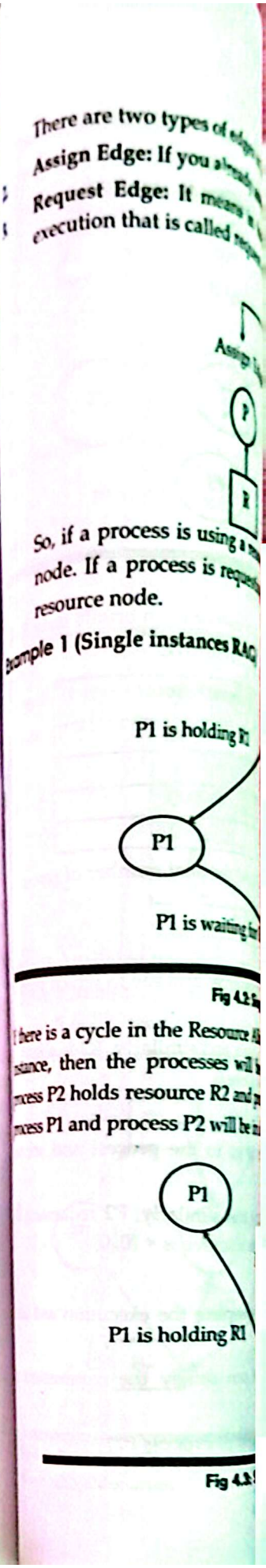
These methods are expensive in the sense that each iteration calls the detection algorithm until the system proves to be deadlock free. The complexity of algorithm is $O(N^2)$ where N is the number of proceeds. Another potential problem is starvation; same process killed repeatedly.

Resource allocation graph (RAG)

Resource allocation graph is explained as what is the state of the system in terms of processes and resources. Like how many resources are available, how many are allocated and what is the request of each process. Everything can be represented in terms of the diagram. One of the advantages of having a diagram is, sometimes it is possible to see a deadlock directly by using RAG, but then it might not be able to know that by looking at the table. But the tables are better if the system contains lots of process and resource and Graph is better if the system contains less number of process and resource. We know that any graph contains vertices and edges. So RAG also contains vertices and edges. In RAG vertices are two types:

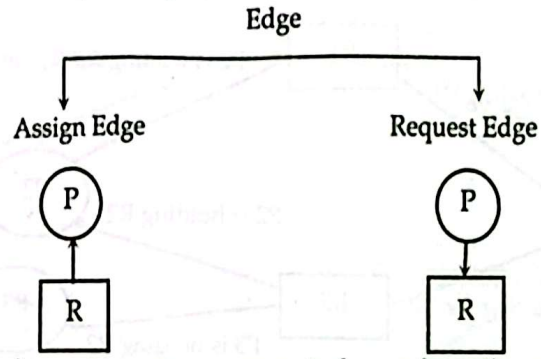
1. **Process vertex** – Every process will be represented as a process vertex. Generally, the process will be represented with a circle.
2. **Resource vertex** – Every resource will be represented as a resource vertex. It is also two type
   - **Single instance type resource** – It represents as a box, inside the box, there will be one dot. So the number of dots indicates how many instances are present of each resource type.
   - **Multi-resource instance type resource** – It also represents as a box, inside the box, there will be many dots present.





There are two types of edges in RAG

2. **Assign Edge:** If you already assign a resource to a process then it is called Assign edge.
3. **Request Edge:** It means in future the process might want some resource to complete the execution that is called request edge.



So, if a process is using a resource, an arrow is drawn from the resource node to the process node. If a process is requesting a resource, an arrow is drawn from the process node to the resource node.

Example 1 (Single instances RAG)

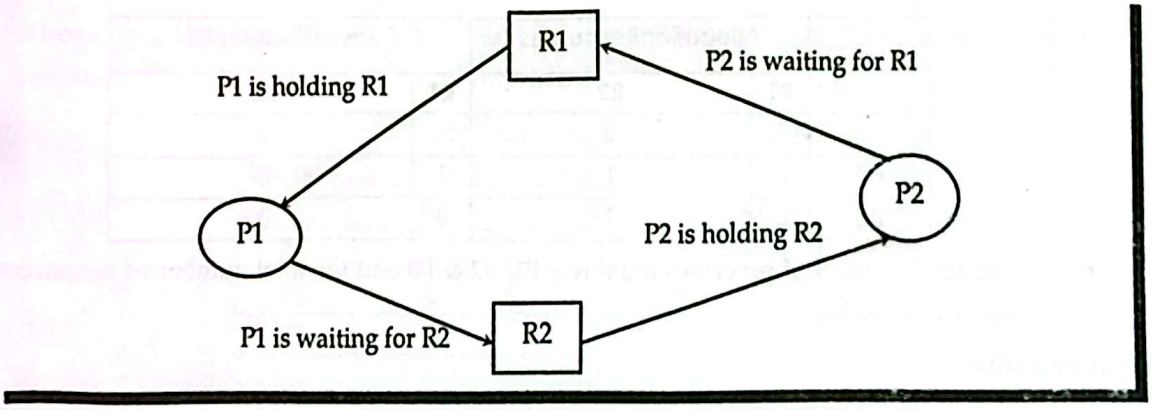


Fig 4.2: Single Instance Resource Type with Deadlock

If there is a cycle in the Resource Allocation Graph and each resource in the cycle provides only one instance, then the processes will be in deadlock. For example, if process P1 holds resource R1, process P2 holds resource R2 and process P1 is waiting for R2 and process P2 is waiting for R1, then process P1 and process P2 will be in deadlock.

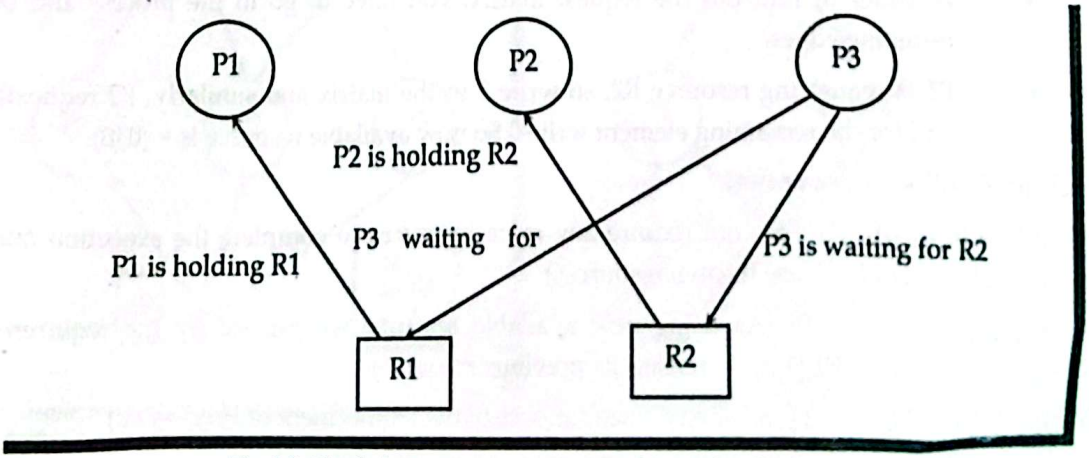


Fig 4.3: Single instance resource type without deadlock

Here is another example that shows Processes P1 and P2 acquiring resources R1 and R2 while process P3 is waiting to acquire both resources. In this example, there is no deadlock because there is no circular dependency. So cycle in single-instance resource type is the sufficient condition for deadlock.

Example 2 (Multi-instances RAG)

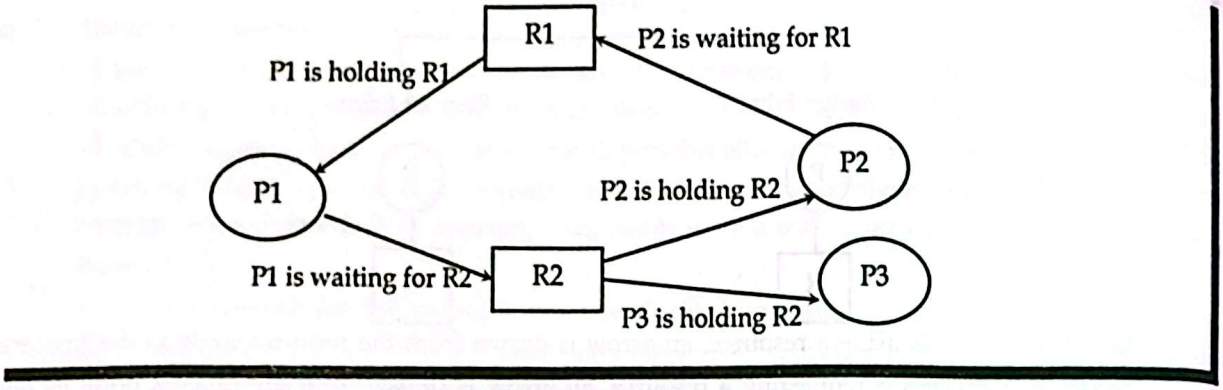


Fig 4.4: Multi Instances without Deadlock

From the above example, it is not possible to say the RAG is in a safe state or in an unsafe state. So to see the state of this RAG, let's construct the allocation matrix and request matrix.

Process	AllocationResources		RequestResources	
	R1	R2	R1	R2
P1	1	0	0	1
P2	0	1	1	0
P3	0	1	0	0

- The total number of processes are three; P1, P2 & P3 and the total number of resources are two; R1 & R2.

Allocation matrix

- For constructing the allocation matrix, just go to the resources and see to which process it is allocated.
- R1 is allocated to P1, therefore write 1 in allocation matrix and similarly, R2 is allocated to P2 as well as P3 and for the remaining element just write 0.

Request matrix

- In order to find out the request matrix, you have to go to the process and see the outgoing edges.
- P1 is requesting resource R2, so write 1 in the matrix and similarly, P2 requesting R1 and for the remaining element write 0.So now available resource is = (0, 0).

Checking deadlock (safe or not)

Available = [0 0] (As P3 does not require any extra resources to complete the execution and after completion P3 [0 1] P3 release its own resources)

New Available = [0 0] (As using new available resource we can satisfy the requirement of process P1 and P1 also P1 [1 0] release its previous resource)

New Available = [1 1] (Now easily we can satisfy the requirement of Process P2)

P2 [0 1]

New Available = [1 2]

So, there is no deadlock in this RAG. Even though there is a cycle, still there is no deadlock. Therefore, in multi-instance resource cycle is not sufficient condition for deadlock.

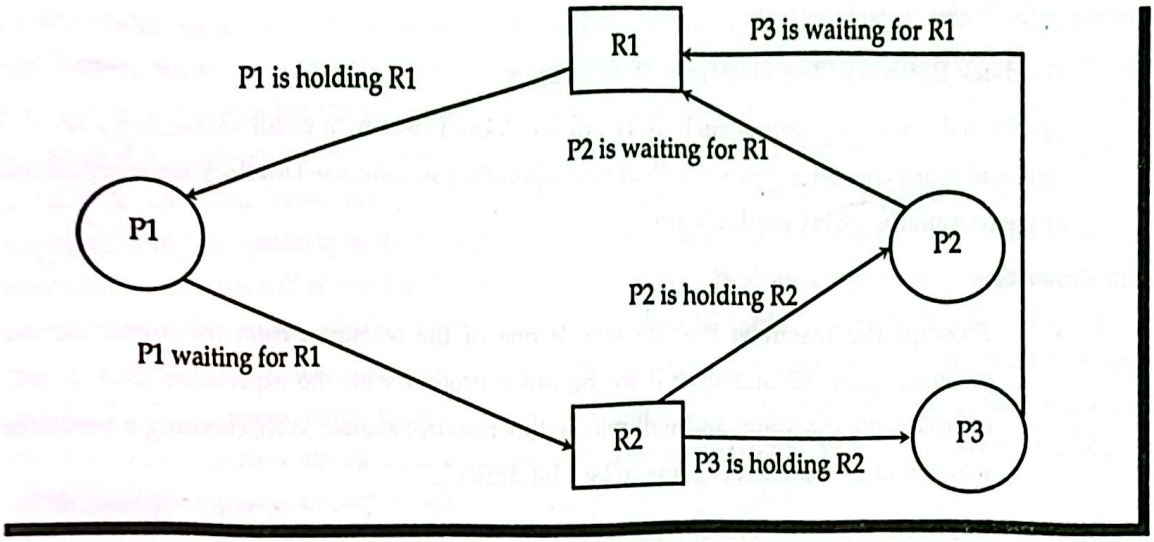


Fig 4.5: Multi Instances with deadlock

Above example is the same as the previous example except that, the process P3 requesting for resource R1.So the table becomes as shown in below.

Process	AllocationResources		RequestResources	
	R1	R2	R1	R2
P1	1	0	0	1
P2	0	1	1	0
P3	0	1	1	0

So, the Available resource is = (0, 0), but requirement are (0, 1), (1, 0) and (1, 0).So you can't fulfill any one requirement. Therefore, it is in deadlock. Therefore, every cycle in a multi-instance resource type graph is not a deadlock, if there has to be a deadlock, there has to be a cycle. So, in case of RAG with multi-instance resource type, the cycle is a necessary condition for deadlock, but not sufficient.

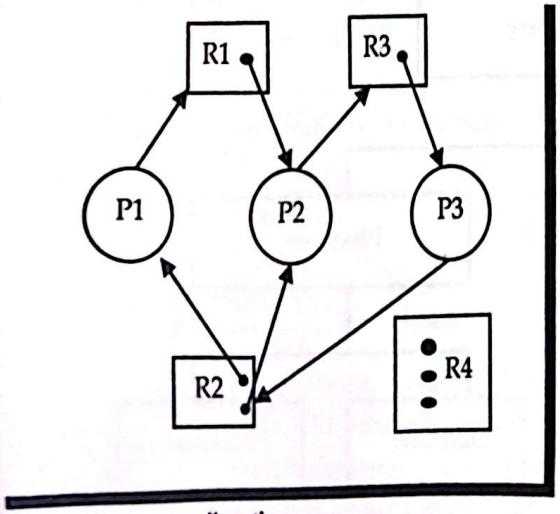


Fig 4.6: Resource allocation Graph with a deadlock

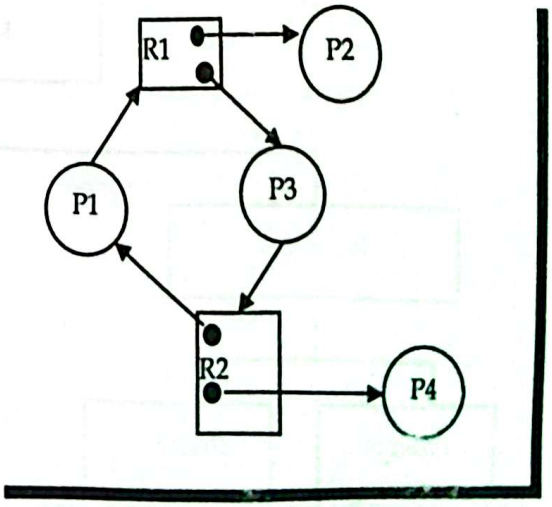


Fig 4.7: Resource allocation Graph with a cycle but no deadlock

Note:If each resource has only one instance, the cycle always causes the deadlock, if each resource type has several instances, then a cycle does not necessarily imply that a deadlock has occurred. In the graph represented on right P4 may release its instance of resource type R2 and P3 is allowed breaking the cycle.

d. Deadlock Recovery (Recovery from Deadlock)

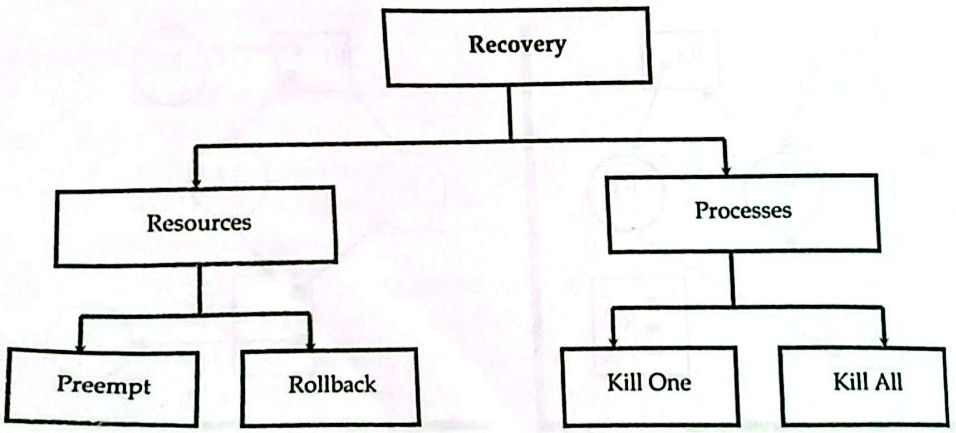
Traditional operating system such as Windows doesn't deal with deadlock recovery as it is time and space consuming process. Real time operating systems use Deadlock recovery. Some of the common recovery methods are

For Resource

- **Preempt the resource:** We can snatch one of the resources from the owner of the resource (process) and give it to the other process with the expectation that it will complete the execution and will release this resource sooner. Well, choosing a resource which will be snatched is going to be a bit difficult.
- **Rollback to a safe state:** System passes through various states to get into the deadlock state. The operating system can roll back the system to the previous safe state. For this purpose, OS needs to implement check pointing at every state. The moment, we get into deadlock, we will rollback all the allocations to get into the previous safe state.

For Process

- **Kill a process:** Killing a process can solve our problem but the bigger concern is to decide which process to kill. Generally, operating system kills a process which has done least amount of work until now.
- **Kill all process:** This is not a suggestible approach but can be implemented if the problem becomes very serious. Killing all process will lead to inefficiency in the system because all the processes will execute again from starting.



Laboratory Works

Program 1: Write a C program to simu avoidance.

Description: In a multiprogramming env number of resources. A process requests re process enters a waiting state. Sometime because the resources it has requested are deadlock. Deadlock avoidance is one o requires that the operating system be gi resources a process will request and use decide for each request whether or not request can be satisfied or must be de available, the resources currently alloca each process. Banker's algorithm is a d with multiple instances of each resource

Program

```
#include<stdio.h>
struct file
{
        int all[10];
        int max[10];
        int need[10];
        int flag;
};
void main()
{
        struct file f[10];
        intfl;
        inti, j, k, p, b, n, r, g, cnt=0, ic
        int avail[10],seq[10];
        clrscr();
        printf("Enter number of pr
        scanf("%d",&n);
        printf("Enter number of re
        scanf("%d",&r);
        for(i=0;i<n;i++)
        {
                printf("Enter de
                printf("\nEnter
                for(j=0;j<r;j++)
                scanf("%d",&f
```