University Institute of Engineering

DEPARTMENT OF COMPUTER SCIENCE & ENGINEERING

Bachelor of Engineering (Computer Science & Engineering)

Subject Name : Operating System

Chapter : Deadlocks

DISCOVER . LEARN . EMPOWER

Outline

- Deadlock
- Conditions of Handling Deadlocks,
- Deadlock Prevention, Avoidance
- Banker's Algorithm,
- Resource Allocation Graph
- Deadlock Detection and Recovery

Introduction

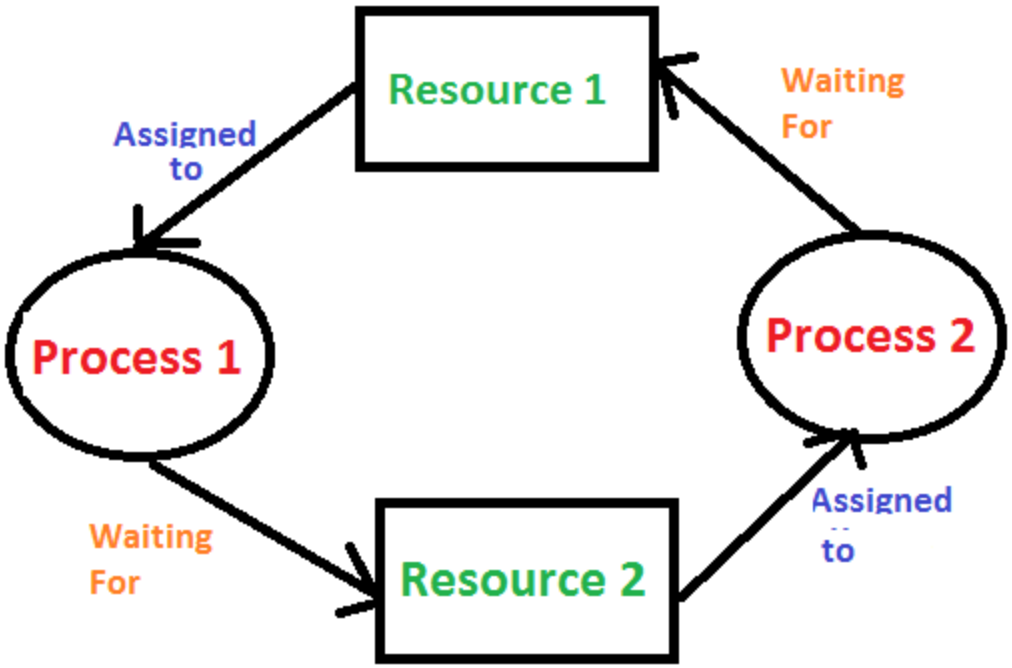
- Under the normal mode of operation, a process may utilize a resource in only the following sequence:

1. **Request:** The process requests the resource. If the request cannot be granted immediately (for example, if the resource is being used by another process), then the requesting process must wait until it can acquire the resource.
2. **Use:** The process can operate on the resource (for example, if the resource is a printer, the process can print on the printer).
3. **Release:** The process releases the resource.

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.

Deadlocks

- Deadlocks are a set of blocked processes each holding a resource and waiting to acquire a resource held by another process.



Necessary Conditions for Deadlock

- There are four conditions that are necessary to achieve deadlock:
  - **Mutual Exclusion** - At least one resource must be held in a non-sharable mode; If any other process requests this resource, then that process must wait for the resource to be released.
  - **Hold and Wait** - A process must be simultaneously holding at least one resource and waiting for at least one resource that is currently being held by some other process.
  - **No preemption** - Once a process is holding a resource (i.e. once its request has been granted), then that resource cannot be taken away from that process until the process voluntarily releases it.
  - **Circular Wait** - A set of processes { P0, P1, P2, . . ., PN } must exist such that every P[i] is waiting for P[(i + 1) % (N + 1)]. (Note that this condition implies the hold-and-wait condition, but it is easier to deal with the conditions if the four are considered separately.)

Methods for Handling Deadlocks

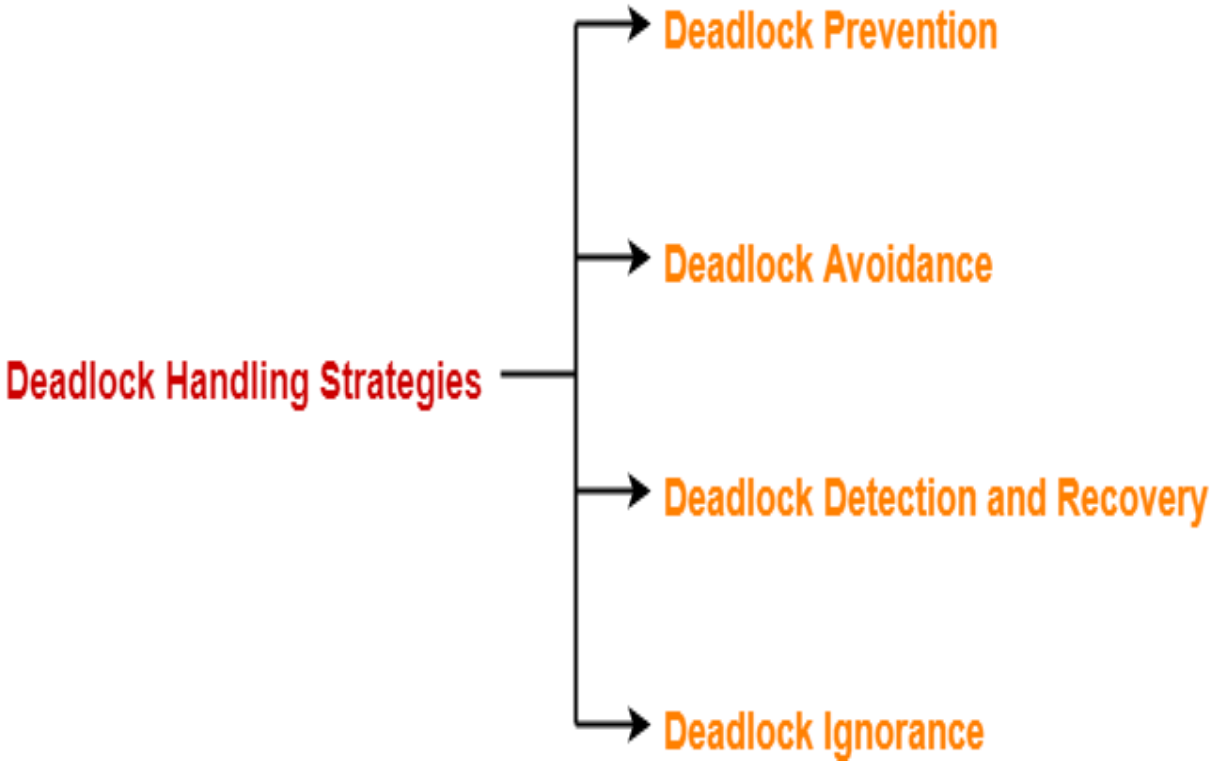
There are three ways to handle deadlock:

1) **Deadlock prevention or avoidance:** The idea is to not let the system into a deadlock state.
One can zoom into each category individually, Prevention is done by negating one of previous mentioned necessary conditions for deadlock.

Avoidance is kind of futuristic in nature. By using strategy of "Avoidance", we have to make an assumption. We need to ensure that all information about resources which process will need are known to us prior to execution of the process. We use Banker's algorithm (Which is in-turn a gift from Dijkstra) in order to avoid deadlock.

2) **Deadlock detection and recovery:** Let deadlock occur, then do preemption to handle it once occurred.
3) **Ignore the problem altogether:** If deadlock is very rare, then let it happen and reboot the system. This is the approach that both Windows and UNIX

Methods for Handling Deadlocks



Deadlock Prevention

- This strategy involves designing a system that violates one of the four necessary conditions required for the occurrence of deadlock.
- This ensures that the system remains free from the deadlock.
- The various conditions of deadlock occurrence may be violated as-

1. Mutual Exclusion-

- To violate this condition, all the system resources must be such that they can be used in a shareable mode.
- In a system, there are always some resources which are mutually exclusive by nature. So, this condition can not be violated.
- Mutual section from the resource point of view is the fact that a resource can never be used by more than one process simultaneously which is fair enough but that is the main reason behind the deadlock. If a resource could have been used by more than one process at the same time then the process would have never been waiting for any resource.
- However, if we can be able to violate resources behaving in the mutually exclusive manner then the deadlock can be prevented. **Therefore, we cannot violate mutual exclusion for a process practically.**

Deadlock Prevention

2. Hold and Wait-

- Allocate all required resources to the process before the start of its execution, this way hold and wait condition is eliminated but it will lead to low device utilization. for example, if a process requires printer at a later time and we have allocated printer before the start of its execution printer will remain blocked till it has completed its execution.
- The process will make a new request for resources after releasing the current set of resources. This solution may lead to starvation.

HOLD AND WAIT

The problem with the approach is:

- Practically **not possible.**
- Possibility of getting **starved** will be **increases** due to the fact that some process may hold a resource for a very long time.

Deadlock Prevention

3. No Preemption-

To ensure that this condition does not hold, we can use the following protocol

- Consider a process is holding some resources and request other resources that can not be immediately allocated to it.
- Then, by forcefully preempting the currently held resources, the condition can be violated.
- A process is allowed to forcefully preempt the resources possessed by some other process only if-
  - It is a high priority process or a system process.
  - The victim process is in the waiting state.

Deadlock Prevention

4. Circular Wait-

- This condition can be violated by not allowing the processes to wait for resources in a cyclic manner.

Approach-

- A natural number is assigned to every resource.
- Each process is allowed to request for the resources either in only increasing or only decreasing order of the resource number.
- In case increasing order is followed, if a process requires a lesser number resource, then it must release all the resources having larger number and vice versa.
- For Example, if P1 process is allocated R5 resources, now next time if P1 ask for R4, R3 lesser than R5 such request will not be granted, only request for resources more than R5 will be granted.
- This approach is the most practical approach and implementable.
- However, this approach may **cause starvation** but will **never lead to deadlock.**

Problem-01

A system is having 3 user processes each requiring 2 units of resource R. The minimum number of units of R such that no deadlock will occur-

Option 1. 3

Option 2. 5

Option 3. 4

Option 4. 6

Solution

In worst case,

The number of units that each process holds = One less than its maximum demand

So,

- Process P1 holds 1 unit of resource R
- Process P2 holds 1 unit of resource R
- Process P3 holds 1 unit of resource R

Thus,

- Maximum number of units of resource R that ensures deadlock = 1 + 1 + 1 = 3
- Minimum number of units of resource R that ensures no deadlock = 3 + 1 = 4

Problem-02

A system is having 10 user processes each requiring 3 units of resource R. The minimum number of units of R such that no deadlock will occur _____?

Solution

In worst case,

The number of units that each process holds = One less than its maximum demand

So,

- Process P1 holds 2 units of resource R
- Process P2 holds 2 units of resource R
- Process P3 holds 2 units of resource R and so on.
- Process P10 holds 2 units of resource R

Thus,

Maximum number of units of resource R that ensures deadlock = 10 x 2 = 20

Minimum number of units of resource R that ensures no deadlock = 20 + 1 = 21

Deadlock Avoidance

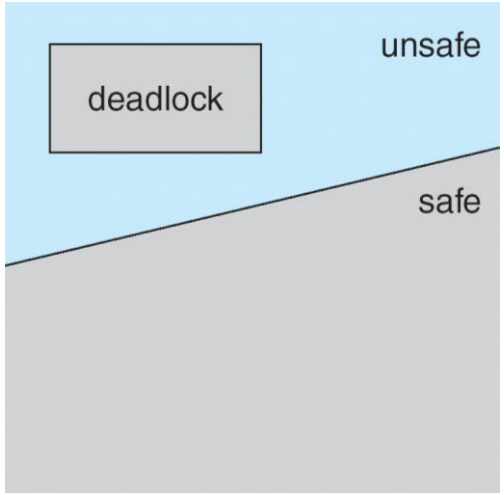
- This strategy involves maintaining a set of data using which a decision is made whether to entertain the new request or not.
- If entertaining the new request causes the system to move in an unsafe state, then it is discarded.
- This strategy requires that every process declares its maximum requirement of each resource type in the beginning.
- The main challenge with this approach is predicting the requirement of the processes before execution.
- Banker's Algorithm is an example of a deadlock avoidance strategy.

Safe State

- When a process requests an available resource, system must decide if immediate allocation leaves the system in a safe state
- System is in **safe state** if there exists a sequence $<P_1, P_2, \ldots, P_n>$ of ALL the processes in the systems such that for each P_i, the resources that P_i can still request can be satisfied by currently available resources + resources held by all the P_j, with $j < I$
- That is:
  - If P_i resource needs are not immediately available, then P_i can wait until all P_j have finished
  - When P_j is finished, P_i can obtain needed resources, execute, return allocated resources, and terminate
  - When P_i terminates, P_{i+1} can obtain its needed resources, and so on

Basic Facts

- If a system is in safe state ⇒ no deadlocks
- If a system is in unsafe state ⇒ possibility of deadlock
- Avoidance ⇒ ensure that a system will never enter an unsafe state.



Deadlock Avoidance Algorithms

- Single instance of a resource type
  - **Use a resource-allocation graph**

- Multiple instances of a resource type
  - **Use the banker's algorithm**

Resource-Allocation Graph

Resource Allocation Graph (RAG)

- Resource Allocation Graph (RAG) is a graph that represents the state of a system pictorially.

In some cases deadlocks can be understood more clearly through the use of **Resource-Allocation Graphs**, having the following properties:

- A set of resource categories, { R1, R2, R3, . . ., RN }, which appear as square nodes on the graph. Dots inside the resource nodes indicate specific instances of the resource. (E.g. two dots might represent two laser printers.)

- A set of processes, { P1, P2, P3, . . ., PN }

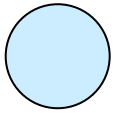
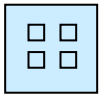
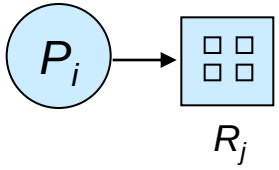
Resource-Allocation Graph

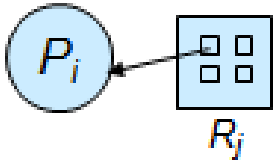
- **Request Edges -** A set of directed arcs from Pi to Rj, indicating that process Pi has requested Rj, and is currently waiting for that resource to become available.

- **Assignment Edges -** A set of directed arcs from Rj to Pi indicating that resource Rj has been allocated to process Pi, and that Pi is currently holding resource Rj.

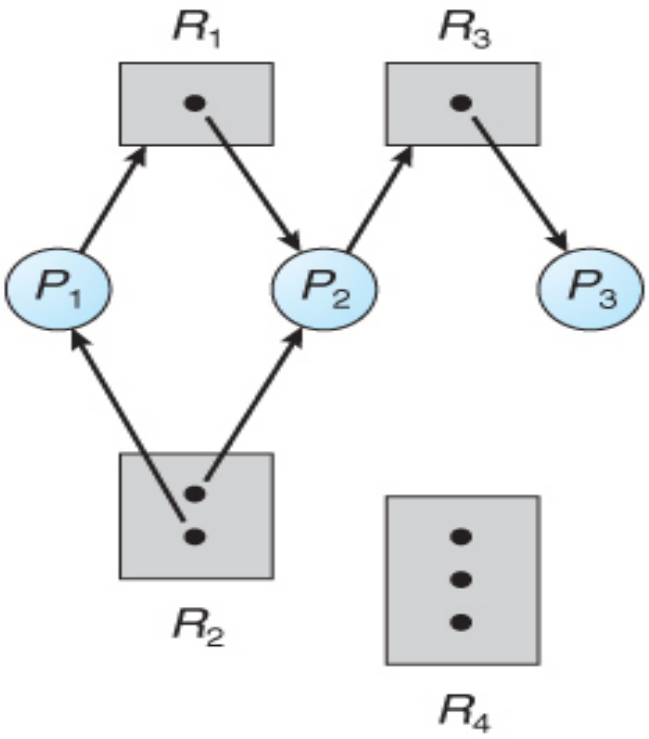
- Note that a **request edge** can be converted into an **assignment edge** by reversing the direction of the arc when the request is granted. (However note also that request edges point to the category box, whereas assignment edges emanate from a particular instance dot within the box.)

Resource-Allocation Graph (Cont.)

- Process 
- Resource Type with 4 instances 
- P_i requests instance of R_j 

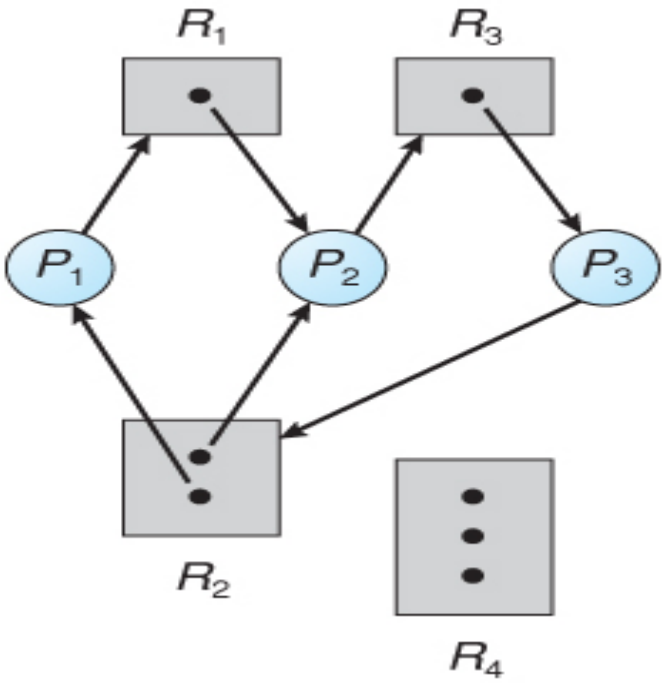
- P_i is holding an instance of R_j 

Resource-Allocation Graph (Cont.)



Resource allocation graph

Resource-Allocation Graph (Cont.)



Resource allocation graph with a deadlock

Suppose that process P3 requests an instance of resource type R2.

Since no resource instance is currently available, we add a request edge P3 →R2 to the graph. At this point, two minimal cycles exist in the system:

$$P_1 \rightarrow R_1 \rightarrow P_2 \rightarrow R_3 \rightarrow P_3 \rightarrow R_2 \rightarrow P_1$$
$$P_2 \rightarrow R_3 \rightarrow P_3 \rightarrow R_2 \rightarrow P_2$$

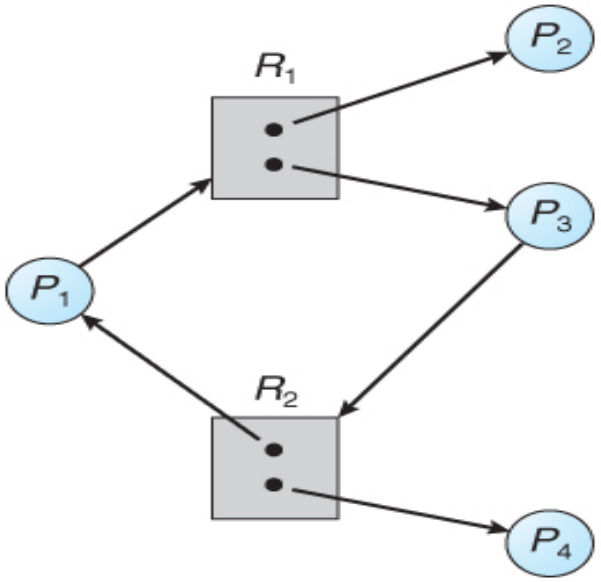
Processes P1, P2 ,and P3 are deadlocked.

Process P2 is waiting for the resource R3, which is held by process P3.

Process P3 is waiting for either process P1 or process P2 to release resource R2.

In addition, process P1 is waiting for process P2 to release resource R1

Resource-Allocation Graph (Cont.)



Resource allocation graph with a cycle but no deadlock

Now consider the resource-allocation graph in Figure 7.3. In this example, we also have a cycle:

$$P_1 \rightarrow R_1 \rightarrow P_3 \rightarrow R_2 \rightarrow P_1$$

However, there is no deadlock. Observe that process P4 may release its instance of resource type R2 . That resource can then be allocated to P3 , breaking the cycle.

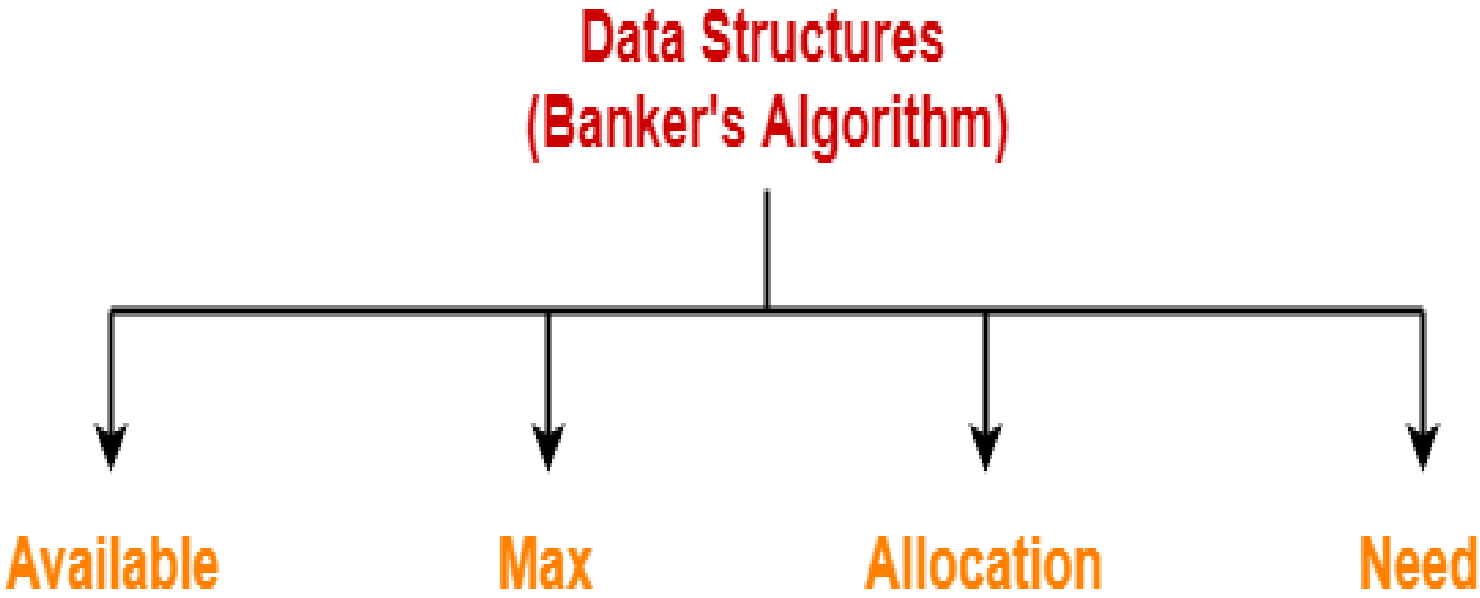
In summary, if a resource-allocation graph does not have a cycle, then the system is not in a deadlocked state. If there is a cycle, then the system may or may not be in a deadlocked state.

Banker's Algorithm

- Banker's Algorithm is a deadlock avoidance strategy.
- The banker's algorithm is **a resource allocation** and **deadlock avoidance** algorithm that tests for safety by simulating the allocation for predetermined maximum possible amounts of all resources, then makes an "**s-state" check** to test for possible activities, before deciding whether allocation should be allowed to continue.
- It is called so because it is used in **banking systems** to decide whether a loan can be granted or not. Suppose there are n number of account holders in a bank and the total sum of their money is S. If a person applies for a loan then the bank first subtracts the loan amount from the total money that bank has and if the remaining amount is greater than S then only the loan is sanctioned. It is done because if all the account holders comes to withdraw their money then the bank can easily do it.

Banker's Algorithm

- To implement banker's algorithm, following four data structures are used-



Data Structures for the Banker's Algorithm

Suppose n is the number of processes, and m is the number of each type of resource used in a computer system.

1. **Available**: It is an array of length 'm' that defines each type of resource available in the system. When Available[j] = K, means that 'K' instances of Resources type R[j] are available in the system.
2. **Max:** It is a [n x m] matrix that indicates each process P[i] can store the maximum number of resources R[j] (each type) in a system.
3. **Allocation:** It is a matrix of m x n orders that indicates the type of resources currently allocated to each process in the system. When Allocation [i, j] = K, it means that process P[i] is currently allocated K instances of Resources type R[j] in the system.
4. **Need:** It is an M x N matrix sequence representing the number of remaining resources for each process. When the Need[i] [j] = k, then process P[i] may require K more instances of resources type Rj to complete the assigned work. **Need[i][j] = Max[i][j] - Allocation[i][j].**

Working

Banker's Algorithm checks whether the request made by the process is valid or not.

Step-01:

- **(Request ≤ Need)-** The number of requested instances of each resource type is less than the need declared by the process in the beginning. If the request is invalid, it aborts the request.
- If the request is valid, it follows step-02.

Step-02:

- **(Request ≤ Available)** Banker's Algorithm checks if the number of requested instances of each resource type is less than the number of available instances of each type
- If the sufficient number of instances are not available, it asks the process to wait longer.
- If the sufficient number of instances are available, it follows step-03.

Working

Step-03:

- Banker's Algorithm makes an assumption that the requested resources have been allocated to the process.
- Then, it modifies its data structures accordingly and moves from one state to the other state.
  - **Available = Available - Request(i)**
  - **Allocation(i) = Allocation(i) + Request(i)**
  - **Need(i) = Need(i) - Request(i)**
- Now, Banker's Algorithm follows the safety algorithm to check whether the resulting state it has entered in is a safe state or not.
- If it is a safe state, then it allocates the requested resources to the process in actual.
- If it is an unsafe state, then it rollbacks to its previous state and asks the process to wait longer.

Safety Algorithm

Safety Algorithm is executed to check whether the resultant state after allocating the resources is safe or not.

1. When a process gets all its resources it must return them in a finite amount of time. Let *Work* and *Finish* be vectors of length *m* and *n*, respectively. Initialize:
   a) *Work* = *Available*
   b) *Finish* [*i*] = *false* for *i* – 0, 1,2 …, *n*.
2. Find an index *i* such that both:
   (a) *Finish* [*i*] = *false*
   (b) $Need_i <= Work$

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

Safety Algorithm

3. $Work = Work + Allocation_i$

   $Finish[i] = true$

   go to step 2.

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

Example of Banker's Algorithm

- Considering a system with five processes from P_0 to $P_4$and three resource types A, B, C. Resource type A has 10 instances, B has 5 instances and type C has 7 instances. Suppose at time t_0 following snapshot of the system has been taken:

	Allocation	Max	Available
	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	

Example (Cont.)

Question 1) What will be the content of need matrix.

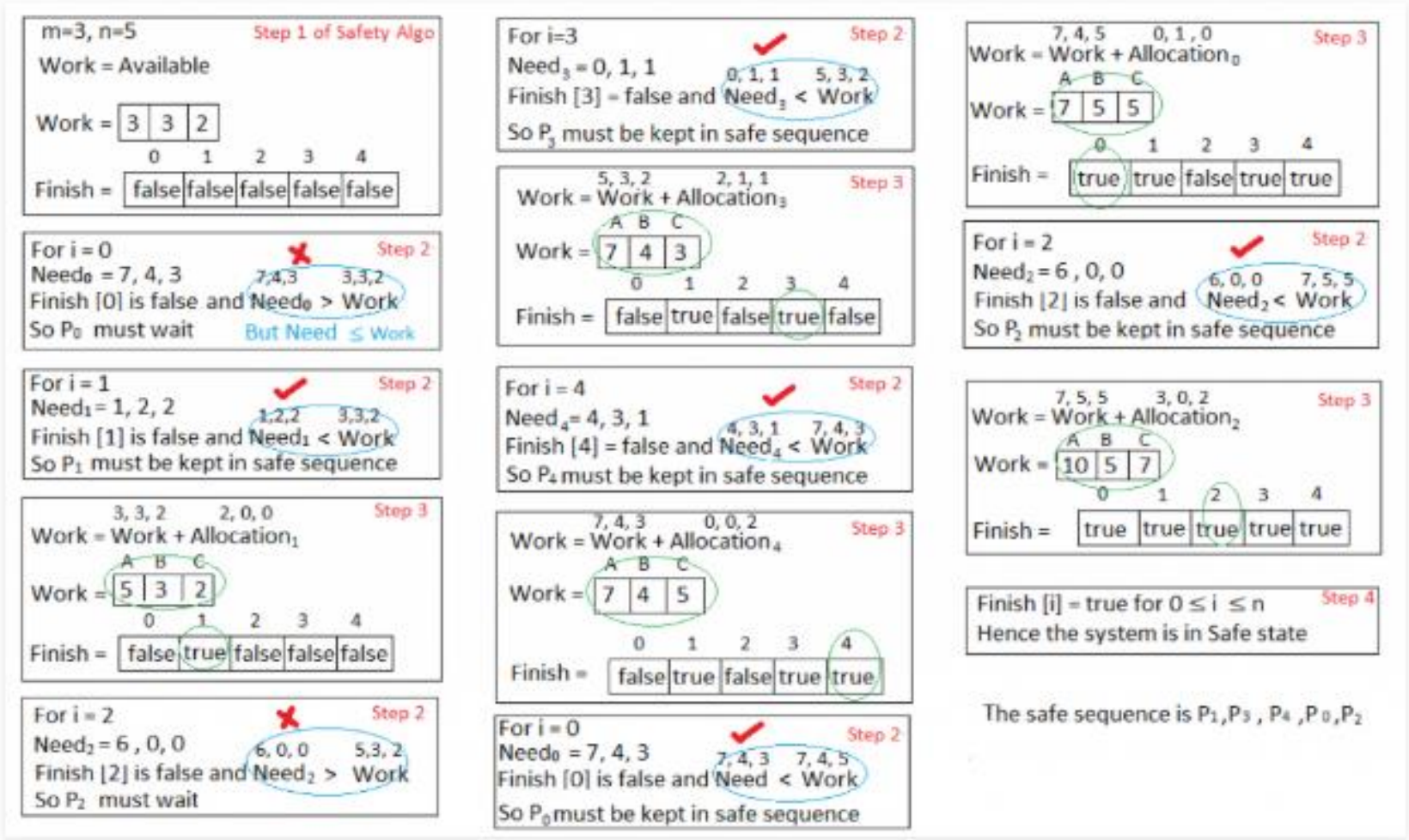
- Need [i,j] is defined to be Max[i,j] – Allocation[i,j].

	Need
	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

The system is in a safe state since the sequence < P1, P3, P4, P2, P0> satisfies safety criteria.

Question2. Is the system in a safe state? If Yes, then what is the safe sequence?

Applying the Safety algorithm on the given system,



Question3. What will happen if process P_1 requests one additional instance of resource type A and two instances of resource type C?

A B C

$Request_1 = 1, 0, 2$

To decide whether the request is granted we use Resource Request algorithm

Step 1

$Request_1 \; (1, 0, 2) < Need_1 \; (1, 2, 2)$ ✓

Step 2

$Request_1 \; (1, 0, 2) < Available \; (3, 3, 2)$ ✓

	Allocation	*Max*	*Available*
	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	

Step 3

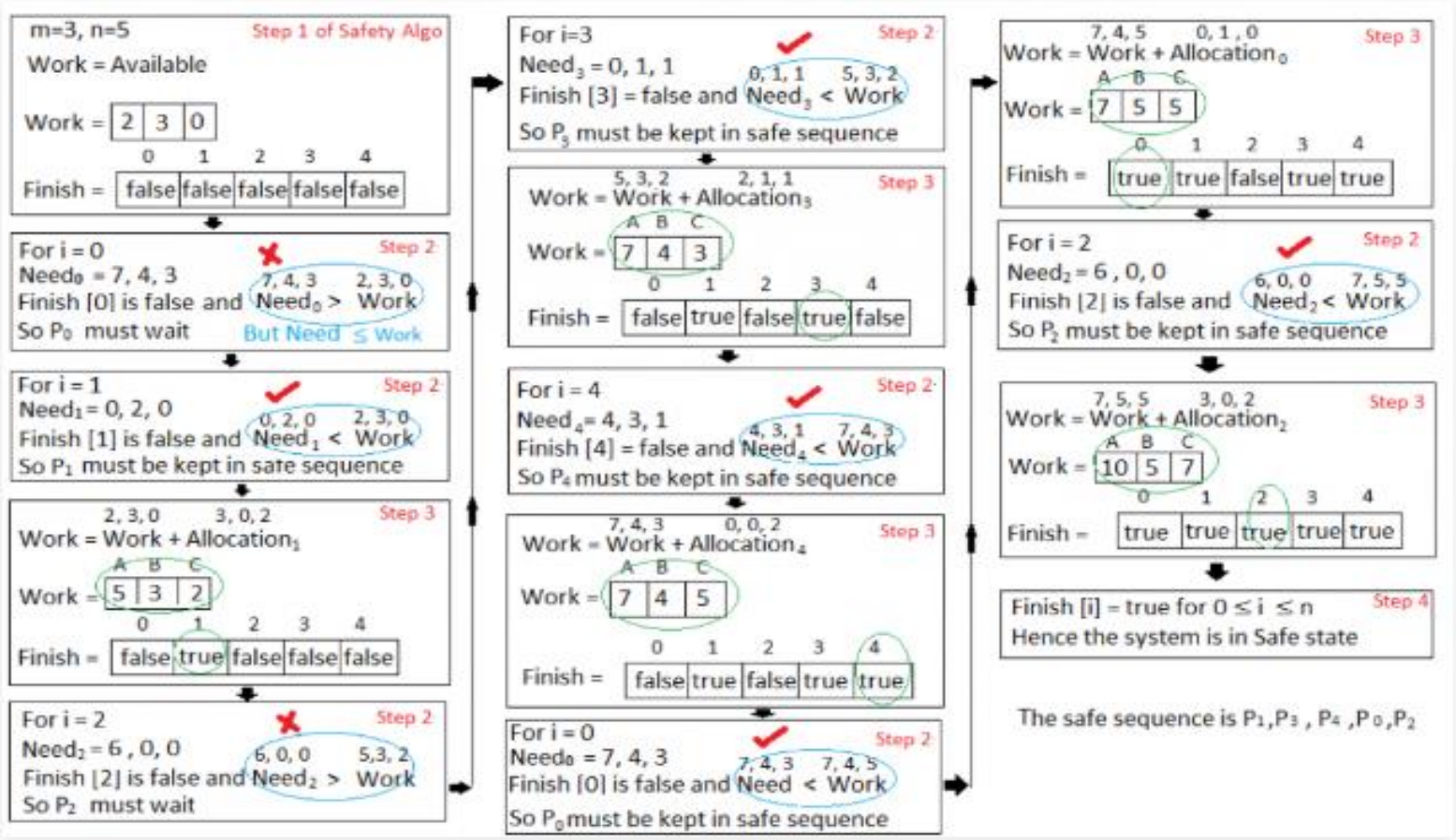
$Available = Available - Request_1$

$Allocation_1 = Allocation_1 + Request_1$

$Need_1 = Need_1 - Request_1$

Process	Allocation	Need	Available
	A B C	A B C	A B C
P_0	0 1 0	7 4 3	2 3 0
P_1	3 0 2	0 2 0	
P_2	3 0 2	6 0 0	
P_3	2 1 1	0 1 1	
P_4	0 0 2	4 3 1	

We must determine whether this new system state is safe. To do so, we again execute Safety algorithm on the above data structures.



Hence the new system state is safe, so we can immediately grant the request for process $\mathbf{P_1}$.

Exercise: Banker's Algorithm

Exercise 1: Assume that there are 5 processes, P0through P4, and 4 types of resources. At T0 we have the following system state:

- Max Instances of Resource Type A = 3 (2 allocated + 1 Available)
- Max Instances of Resource Type B = 17 (12 allocated + 5 Available)
- Max Instances of Resource Type C = 16 (14 allocated + 2 Available)
- Max Instances of Resource Type D = 12 (12 allocated + 0 Available)

Given Matrices												
	Allocation Matrix (N0 of the allocated resources By a process)				Max Matrix Max resources that may be used by a process				Available Matrix Not Allocated Resources			
	A	B	C	D	A	B	C	D	A	B	C	D
P_0	0	1	1	0	0	2	1	0	1	5	2	0
P_1	1	2	3	1	1	6	5	2				
P_2	1	3	6	5	2	3	6	6				
P_3	0	6	3	2	0	6	5	2				
P_4	0	0	1	4	0	6	5	6				
Total	2	12	14	12								

Solution: Banker's Algorithm

Given Matrices												
	Allocation Matrix (N0 of the allocated resources By a process)				Max Matrix Max resources that may be used by a process				Available Matrix Not Allocated Resources			
	A	B	C	D	A	B	C	D	A	B	C	D
P_0	0	1	1	0	0	2	1	0	1	5	2	0
P_1	1	2	3	1	1	6	5	2				
P_2	1	3	6	5	2	3	6	6				
P_3	0	6	3	2	0	6	5	2				
P_4	0	0	1	4	0	6	5	6				
Total	2	12	14	12								

1. Create the need matrix (max-allocation)

Need(i) = Max(i) – Allocated(i)

(i=0) (0,2,1,0) - (0,1,1,0) = (0,1,0,0)

(i=1) (1,6,5,2) - (1,2,3,1) = (0,4,2,1)

(i=2) (2,3,6,6) - (1,3,6,5) = (1,0,0,1)

(i=3) (0,6,5,2) - (0,6,3,2) = (0,0,2,0)

(i=4) (0,6,5,6) - (0,0,1,4) = (0,6,4,2)

Ex. Process P1 has max of (1,6,5,2) and allocated by (1,2,3,1)

Need(p1) = max(p1)- allocated(p1) = (1,6,5,2) – (1,2,3,1) = (0,4,2,1)

Need Matrix = Max Matrix – Allocation Matrix				
	A	B	C	D
P_0	0	1	0	0
P_1	0	4	2	1
P_2	1	0	0	1
P_3	0	0	2	0
P_4	0	6	4	2

Solution: Banker's Algorithm

Given Matrices												
	Allocation Matrix (N0 of the allocated resources By a process)				Max Matrix Max resources that may be used by a process				Available Matrix Not Allocated Resources			
	A	B	C	D	A	B	C	D	A	B	C	D
P_0	0	1	1	0	0	2	1	0	1	5	2	0
P_1	1	2	3	1	1	6	5	2				
P_2	1	3	6	5	2	3	6	6				
P_3	0	6	3	2	0	6	5	2				
P_4	0	0	1	4	0	6	5	6				
Total	2	12	14	12								

Need Matrix = Max Matrix – Allocation Matrix				
	A	B	C	D
P_0	0	1	0	0
P_1	0	4	2	1
P_2	1	0	0	1
P_3	0	0	2	0
P_4	0	6	4	2

Use the safety algorithm to test if the system is in a safe state or not?

Solution: Banker's Algorithm

Given Matrices												
	Allocation Matrix (N0 of the allocated resources By a process)				Max Matrix Max resources that may be used by a process				Available Matrix Not Allocated Resources			
	A	B	C	D	A	B	C	D	A	B	C	D
P_0	0	1	1	0	0	2	1	0	1	5	2	0
P_1	1	2	3	1	1	6	5	2				
P_2	1	3	6	5	2	3	6	6				
P_3	0	6	3	2	0	6	5	2				
P_4	0	0	1	4	0	6	5	6				
Total	2	12	14	12								

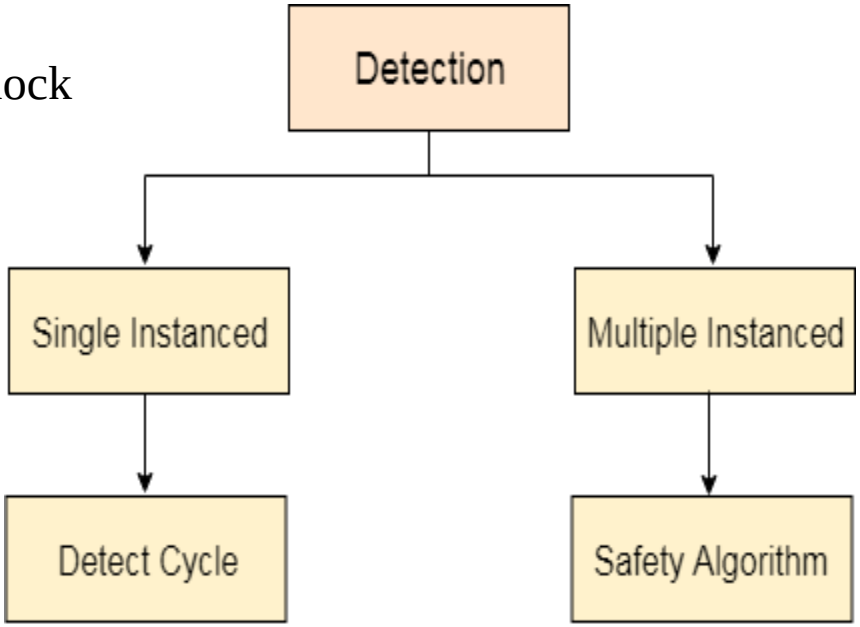
Need Matrix = Max Matrix – Allocation Matrix				
	A	B	C	D
P_0	0	1	0	0
P_1	0	4	2	1
P_2	1	0	0	1
P_3	0	0	2	0
P_4	0	6	4	2

Use the safety algorithm to test if the system is in a safe state or not?

The system is in a safe state and the processes will be executed in the following order: P0,P3,P4,P1,P2

Deadlock Detection and Recovery

- If a system does not employ either a deadlock-prevention or a deadlock-avoidance algorithm, then a deadlock situation may occur. In this environment, the system may provide:
- An algorithm that examines the state of the system to determine whether a deadlock has occurred
- •An algorithm to recover from the deadlock



Deadlock Detection and Recovery

- The OS can detect the deadlocks with the help of Resource allocation graph.
- In **single instanced resource types**, if a cycle is being formed in the system then there will definitely be a deadlock can be confirmed from **wait-for graph**.
- On the other hand, in **multiple instanced resource type graph**, detecting a cycle is not just enough.
- We have to apply **the safety algorithm** on the system by converting the resource allocation graph into the allocation matrix and request matrix.

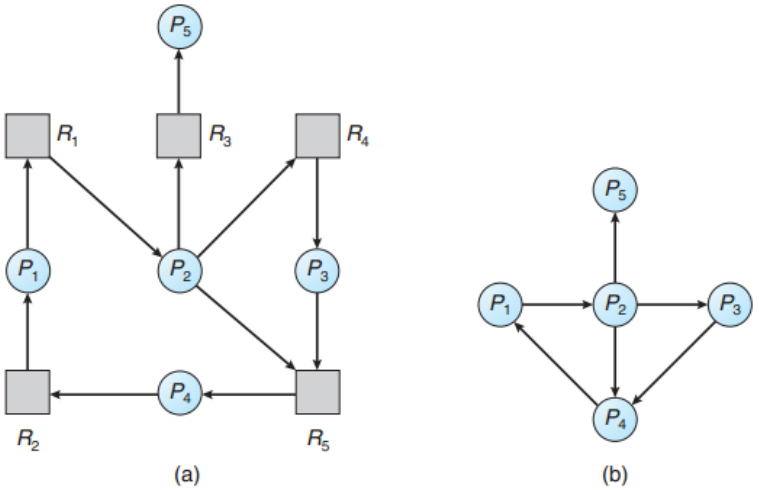
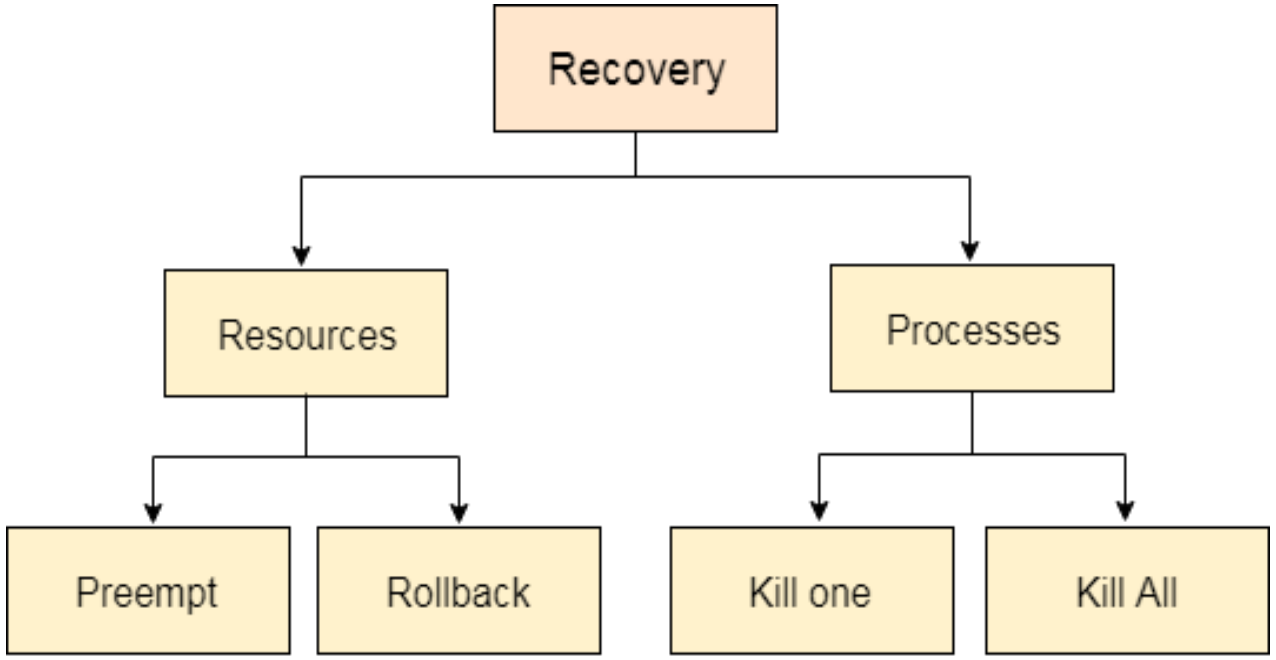


Figure 7.9 (a) Resource-allocation graph. (b) Corresponding wait-for graph.

Deadlock Recovery

In order to recover the system from deadlocks, either OS considers resources or processes.



Deadlock Recovery

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 rollback 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.

Deadlock Recovery

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.

Deadlock Ignorance

- This strategy involves ignoring the concept of deadlock and assuming as if it does not exist.
- This strategy helps to avoid the extra overhead of handling deadlock.
- Windows and Linux use this strategy and it is the most widely used method.
- It is also called as **Ostrich approach**.

References

- "Operating System Concepts" by Avi Silberschatz and Peter Galvin
- "Operating Systems: Internals and Design Principles" by William Stallings
- www.tutorialpoint.com
- www.javapoint.com