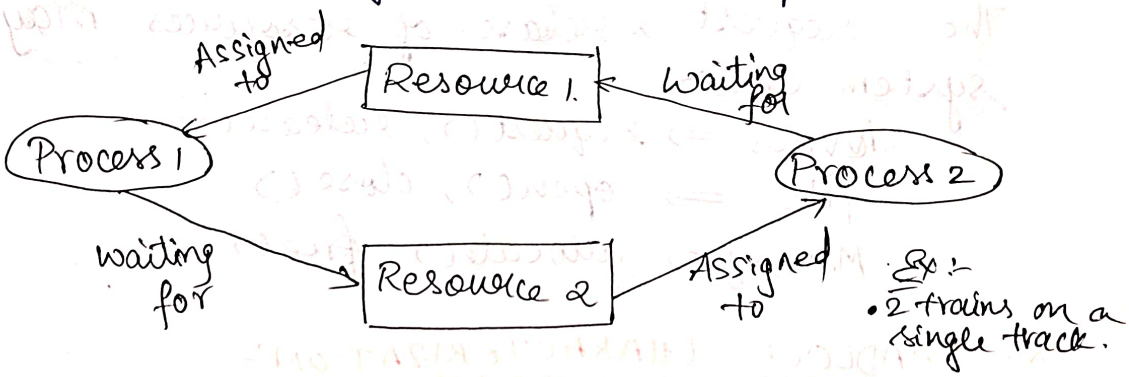
UNIT - III

DEADLOCKS & SYNCHRONIZATION TOOLS

⇒ Deadlocks :-

A deadlock is a situation in which one / more processes are blocked because it holds a resource and is waiting for another resource held by some other process.



1. SYSTEM MODEL :-

CPU cycles, files, I/O devices (such as network interfaces & DVD drives) are examples of resource types

If a system has 4 CPU's, then we say resource type CPU has 4 instances.

** If a thread / process requests an instance of a resource type, the allocation of any instance of the type should satisfy the request.

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

(i) REQUEST :-

The thread requests the resource. If

the request cannot be granted immediately, then the requesting thread must wait until it can acquire the resource.

(ii) USE :-
The thread can operate the resource

(iii) RELEASE :-
The thread releases the resource.

The request & release of resources may be system calls :-
devices ⇒ request(), release()
file ⇒ open(), close()
Memory ⇒ allocate(), free()

2. DEADLOCK CHARACTERIZATION :-

2.1. Necessary Conditions :-

A deadlock can arise if following 4 conditions hold simultaneously in a system :-

(i) MUTUAL EXCLUSION :-
"ATLEAST ONE RESOURCE MUST BE HELD IN A NON-SHARABLE MODE", i.e.,
only one thread at a time can use the resource. If another thread requests that resource, the requesting thread must be delayed until resource has been released.

(ii) HOLD AND WAIT :-
A thread must be holding atleast one resource and waiting to acquire additional resources that are currently being held by other threads.

(iii) NO PREEMPTION :-
Resources cannot be preempted; i.e.,

a resource can be released only voluntarily by the thread holding it, after the thread has completed its task.

(iv) CIRCULAR WAIT:-

A set $\{T_0, T_1, T_2, \ldots T_n\}$ of waiting threads must exist such that T_0 is waiting for a resource held by T_1, T_1 is waiting for resource held by $T_2, \ldots, T_{n-1}$ is waiting for a resource held by T_n & T_n is waiting for a resource held by T_1.

2.2 Resource Allocation Graph:- (RAG)

A resource allocation graph is a visual representation of how resources are assigned to various processes in an OS.

* It's a directed graph $G(V, E)$.

* Types of vertices:-

(i) Process Vertices:-

Every active process/thread is a process vertex & represented by a circle.

(ii) Resource Vertex:-

Every resource is represented as resource vertex & represented by a rectangle.

Single instance resource

Multiple Instance resource

* Types of Edges:-

There are 2 types of edges in RAG

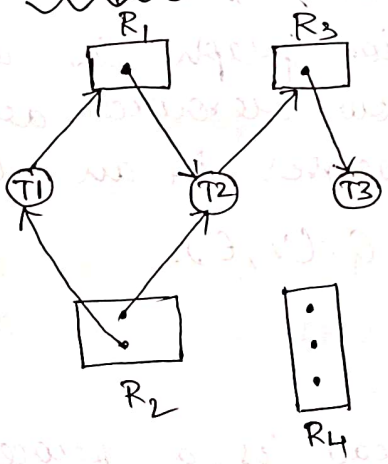
(i) Assign Edge:-

A directed eadge from resource type R_j to thread T_i is denoted by $R_j \rightarrow T_i$, is called an assignment edge (R_j allocated to T_i)

(ii) Request Edge:-

A directed edge from thread T_i to resource type R_j is denoted by $T_i \rightarrow R_j$, signifies that thread T_i requested an instance of resource type R_j. It is called request edge

example:- Resource Allocation Graph:-



G (V, E)

- Vertex Set:-

Process/Thread vertices $T = \{T_1, T_2, T_3\}$

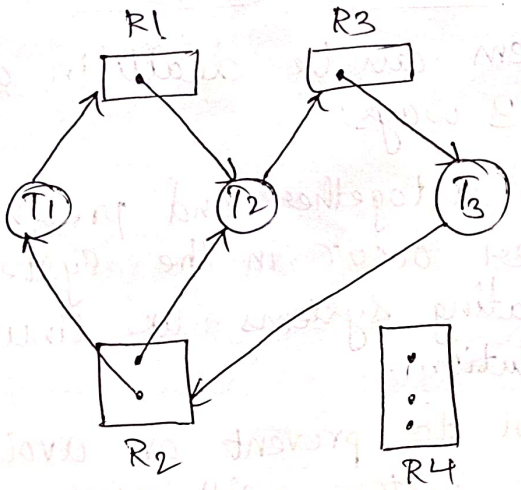
Resource Vertices, $R = \{R_1, R_2, R_3, R_4\}$

- Edges Set:-

$E = \{T_1 \rightarrow R_1, T_2 \rightarrow R_3, R_1 \rightarrow T_2, R_3 \rightarrow T_3, R_2 \rightarrow T_1, R_2 \rightarrow T_2\}$

- Resource Instances:-
  - 1 instance of resource type R_1.
  - 2 instances of resource type R_2.
  - 1 instance of resource type R_3
  - 3 instances of resource type R_4.
- Thread/Process States:-
  - Thread T_1 holding instance of R_2 & is waiting for instance of R_1.
  - Thread T_2 holding instance of R_2 & waiting for instance of R_3.
  - thread T_3 is holding instance of R_3.

2.2.1 :- Resource Allocation Graph with deadlock



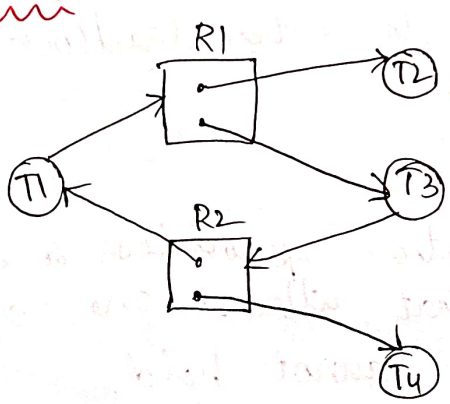
At this point, 2 minimal cycles exist in the system.

$T_1 \rightarrow R1 \rightarrow T2 \rightarrow R3 \rightarrow T3 \rightarrow R_2 \rightarrow T_1$.

$T_2 \rightarrow R3 \rightarrow T3 \rightarrow R_2 \rightarrow T2$.

Threads T_1, T_2, T_3 are deadlocked. T_2 is waiting for R_3, which is held by T3. T3 is waiting for R_2 held by T_1 & T2.

2.2.2. Resource Allocation Graph with no deadlock



Cycle :-

$T1 \rightarrow R1 \rightarrow T3 \rightarrow R_2 \rightarrow T_1$

Here, there is a cycle, but no deadlock. Thread T_4 may release its instance of resource type R_2. Resource can be then allocated to T3, breaking the cycle

NO CYCLE ⟹ No deadlock

CYCLE ⟹ May / may not be deadlock

3. Methods for Handling Deadlocks:-

Deadlock problem can be dealt in one of the following 3 ways:-

(i) Ignore problem altogether and pretend that deadlocks never occur in the system
Most operating systems like Linux & Windows use this solution.

(ii) Use a protocol to prevent or avoid deadlocks, ensuring that system will never enter a deadlocked state
deadlock prevention:-
Provides a set of methods to ensure that atleast one of the necessary conditions cannot hold.
deadlock avoidance:-
Additional knowledge in advance, concerning which resources a thread will request ~~or~~ & use in its lifetime, OS will avoid a deadlock

(iii) Allowing the system to enter deadlock, detect it, and recover.

(A) Deadlock Prevention:-

Deadlock prevention provides a set of methods to ensure that atleast one of the necessary conditions cannot hold.

(i) Elimination Mutual Exclusion:-
Sharable resources donot require mutually exclusive access & thus cannot be involved in a deadlock. Read-only files are example of a sharable resource.

But in general, we cannot prevent deadlocks by denying mutual exclusion condition, because some resources are intrinsically non-shareable

(ii) Eliminate Hold & Wait:-

There are 2 ways to eliminate hold & wait.

- By eliminating wait:-
  The process specifies the resources it requires in advance so that it doesnot have to wait for allocation after execution starts
- By eliminating hold:-
  The process can make a new resource request only when it is holding none of the resources (All resources have to be released before request)

(iii) Eliminate No preemption:-

Two ways to eliminate no preemption.

- Processes must release resources voluntarily:-
  If the process is holding some resources & requests another resource that cannot be immediately allocated, then all the resources the thread is currently holding are preempted.
- Avoid partial allocation:-
  Allocate all required resources to a process at once before it begins execution. If not all resources are available, the process must wait.

(iv) Eliminate Circular-wait:-

To eliminate circular wait for deadlock prevention:-

- Assign a unique number to each resource.
- Processes can only request resources in

increasing order of their numbers

Ex: Printer - 1
CPU - 5
Memory - 6
CD drive - 7

P_1	P_2
1.	5

P_1 can request for 5/6/7.
P_2 can request for 6/7 only

Here As P_2 holds 5 & cannot request 1, then there is no circular wait.

(5) Deadlock Avoidance

An alternative method for avoiding deadlocks is to require additional information about how resources are to be requested.

A deadlock avoidance algorithm dynamically examines the resource-allocation state to ensure that a circular wait condition never exists.

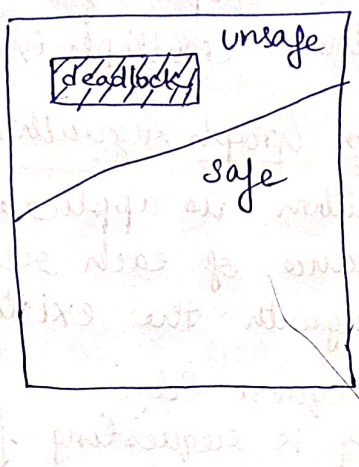
The resource allocation state is defined by the number of available and allocated resources and the maximum demands of the threads/processes

SAFE STATE:

A sequence of processes $<P_1, P_2, \ldots P_n>$ is a safe sequence for the current allocation state, if, for each P_i, the resource requests that P_i can still make can be satisfied by the currently available resources plus the resources held by all P_j, with $j<i$.

A system is in safe state only if there exists a safe sequence. & if the system can allocate resources to each thread in some

·order & still can avoid a deadlock.



Safe, unsafe & deadlocked state spaces.

Example:-

Consider a system with 12 magnetic tapes and 3 processes P_0, P_1, P_2.

	Max. Needs	Current Needs
P_0	10	5
P_1	4	2
P_2	9	2

At time t_0, all process P_0, P_1, P_2 are holding $5+2+2 = 9$ tapes. So, there are $12-9 = 3$ tape drives that are free.

The system is in a safe state with safe sequence $\langle P_1, P_0, P_2 \rangle$.

Process P_1 can immediately be allocated all its tape drives i.e., 2 current needs + 2 additional requests (less than 3). After P_1 completes execution, it will return all its held resources i.e., 4.

Now, system will have 5 free tape drives.

Then P_0 can be allocated its extra 5 drives and P_0 returns them, once its terminated.

Now, system will have 10 free tape drives, with which P_2 can be served with its addition requests & P_2 returns them once the task is over. System will have its all 12 tape drives

Deadlock avoidance algorithms are:-
(i) Resource-Allocation Graph Algorithm. (Single instance)
(ii) Banker's Algorithm. (Multiple instances)

(i) Resource-Allocation Graph Algorithm:-

This algorithm is applicable when there is only 1 instance of each resource type in the system. Alongwith the existing edges in RAG

$P_i \rightarrow R_j \Rightarrow$ Request Edge
P_i is requesting for resource R_j

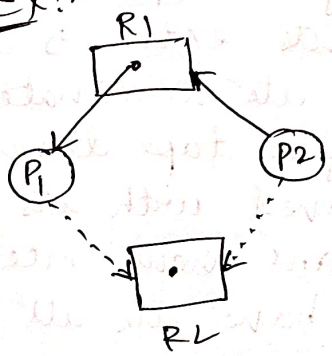
$R_j \rightarrow P_i \Rightarrow$ Assignment Edge
Resource R_j is assigned to P_i

an extra edge called claim edge is added

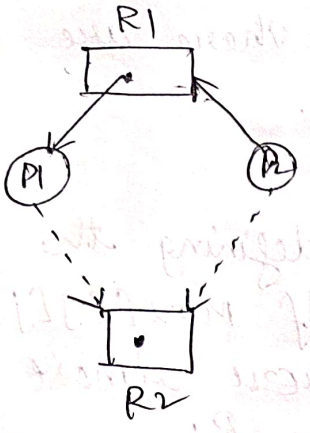
$P_i \dashrightarrow R_j \Rightarrow$ Claim edge.
Process P_i may request resource R_j at sometime in future. Represented using dashed line.

RULE:- Now suppose process P_i requests resource R_j, the request can only be granted if converting request edge $P_i \rightarrow R_j$ to an assignment edge $R_j \rightarrow P_i$ doesnot result in cycle & eventually a deadlock. (Unsafe state).
If no cycle exists, then the allocation of resource results in safe state.

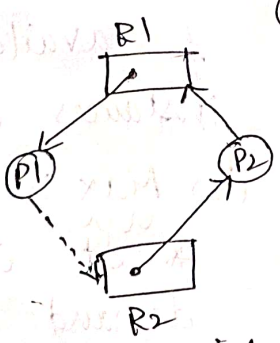
Ex:-



Considering the RAG, Suppose that P_2 requests R_2. Although R_2 is free currently, we cannot allocate it to P_2, ∵ this allocation creates deadlock



If R_2 is allocated to P2 and P1 is requesting R2 at the same time



Circular wait

This results in a circular wait, so, allocating R2 to P2 itself results in unsafe state. Thus R2 is not allocated though R2 is free

(ii) Bankers Algorithm:-

Bankers algorithm is a deadlock avoidance algorithm applicable to a resource allocation system with multiple instances of each resource type.

Name was chosen because the algorithm can be used in banking system to ensure that the bank never allocated its available cash in such a way that it could no longer satisfy the needs of all its customers.

Components / data structures Used in algorithm:-

The following datastructures are needed, where $n \rightarrow$ processes
$m \rightarrow$ number of resources.

(a) Available:-

* It is an array/vector of length "m" indicating no. of available resources of each type.

if available[j] = k, means there are k instances of resource type R_j.

(b) Max :-

* It is a n×m matrix defining the maximum demand of each process. If Max[i][j] = k, then process P_i may request atmost k instances of resource type R_j.

(c) Allocation:-

* It is a n×m matrix defining the no. of ~~each~~ resources of each type currently allocated to each process. If Allocation[i][j] = k, then Process P_i is currently allocated k instances of resource type R_j.

(d) Need:-

* It is a n×m matrix indicating remaining resource need of each process. If Need[i][j] = k, then Process P_i may need k more instances of resource type R_j to complete its task.

Need[i][j] = Max[i][j] - Allocation[i][j].

SAFETY ALGORITHM:-

The following algorithm is for finding out whether or not a system is in safe state.

Step 1:- Let work and finish be arrays of length m and n, respectively. Initialize work = Available & Finish[i] = False for i = 0, ... n-1.

Step 2:- Find an index i such that both.

a) Finish[i] == false.

b) $Need_i \leq work$.

If no such i exists, go to step 4.

Step 3:-

work = work + Allocation$_i$.

Finish[i] = true.

Go to step 2.

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

Ex:- Consider a system with 5 process P_0, P_1, P_2, P_3, P_4 and 3 resource types A, B, C. Resource type A has 10 instances, B has 5 instances, C has 7 instances. Suppose the following snapshot represents the current state of the system, identify the safe sequence.

	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			

Sol:- Total Resources

A	B	C
10	5	7

Need Matrix is defined as:-

Need[i][j] = 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

	A	B	C
Available	3	3	2
work	3	3	2
~~Finish~~	~~F~~	~~F~~	~~F~~

	0	1	2	3	4
Finish	F	F	F	F	F

i=0 Process P_0:- $Need_0$ is (7,4,3) & work is (3,3,2)

$Need_0 \not\leq work$

i=1 Process P_1:- $Need_1$ is (1,2,2)

$Need_1 \leq work$

i.e., (1,2,2) ≤ (3,3,2) true. ✓

Finish[1] == false

So, P_1 is a correct choice

Work = Work + Allocation$_1$.

Work = (3,3,2) + (2,0,0).

= (5,3,2)

	0	1	2	3	4
Finish	F	T	F	F	F

& Finish[1] = true

i=2 Process P_2 :- Need$_2$ = (6,0,0)

Finish[2] is false ⇒ true

Need$_2$ ≤ Work

(6,0,0) ≤ (5,3,2) ⇒ False

i=3 Process P_3 :- Need$_3$ = (0,1,1).

Finish[3] is false ⇒ true

Need$_3$ ≤ Work.

(0,1,1) ≤ (5,3,2) ⇒ true

Work = Work + Allocation$_3$

= (5,3,2) + (2,1,1).

Work = (7,4,3)

Finish[3] = true

	0	1	2	3	4
Finish	F	T	F	T	F

i=4 :- Process P_4 :- Need$_4$ = (4,3,1)

Finish[4] is false ⇒ true

Need$_4$ ≤ Work

(4,3,1) ≤ (7,4,3) ⇒ true

Work = Work + Allocation$_4$

= (7,4,3) + (0,0,2)

Work = (7,4,5)

Finish[4] = true

	0	1	2	3	4
Finish	F	T	F	T	T

i=0 :

Process P_0 :- Need$_0$ = (7,4,3)

Finish[0] = False ⇒ true

Need$_0$ ≤ Work

~~(7,4,3) ≤ (7,4,5) ⇒ false~~

(7,4,3) ≤ (7,4,5) ⇒ true

Work = Work + Allocation$_0$

= (7,4,5) + (0,1,0)

Work = (7,5,5)

	0	1	2	3	4
finish	T	T	F	T	T

i=2 Process P_2 :- Need$_2$ = (6,0,0)

Finish[2] is false ⇒ true

Need$_2$ ≤ Work

(6,0,0) ≤ (7,5,5)

Work = Work + Allocation$_2$

= (7,5,5) + (3,0,2)

= (10,5,7)

All processes can be allocated requests without entering a deadlock. The safe sequence is $< P_1, P_3, P_4, P_0, P_2 >$.

(Q). Consider the following resource allocation state 5 processes P_0, P_1, P_2, P_3, P_4. Using Banker's Algorithm, Illustrate that the system is in a safe state by demonstrating an order in which the processes may complete.

	Allocation				Max				Available			
	A	B	C	D	A	B	C	D	A	B	C	D
P_0	0	0	1	2	0	0	1	2	1	5	2	0
P_1	1	0	0	0	1	7	5	0				
P_2	1	3	5	4	2	3	5	6				
P_3	0	6	3	2	0	6	5	2				
P_4	0	0	1	4	0	6	5	6				

$< P_0, P_2, P_3, P_4, P_1 >$.

Resource Request Algorithm:

Let $Request_i$ be the request vector for process P_i. If $Request_i[j] == k$, then process P_i wants k instances of resource type R_j. When a request for resources is made by Process P_i, the following actions are taken

(i) If $Request_i \leq Need_i$, go to step 2. Otherwise raise an error condition, since the process has exceeded its maximum claim.

(ii) If $Request_i \leq Available$, go to step 3. Otherwise P_i must wait, since the resources are not available

(iii) Have the system pretend to have allocated the requested resources to process P_i by modifying the state as follows:-

$Available = Available - Request_i$;

$Allocation_i = Allocation_i + Request_i$;

$Need_i = Need_i - Request_i$;

If the resulting resource-allocation state is safe, the transaction is complete, & Process P_i is allocated its resources.

If resulting allocation is unsafe, the old resource allocation state is restored

(8) Consider the following snapshot of the resource allocation state of a system with 5 processes:-

(6) Deadlock Detection:-

If a system doesnot employ deadlock prevention or avoidance algorithm, then a deadlock occurs. So, an algorithm that examines the state of the system to determine whether a deadlock has occured.

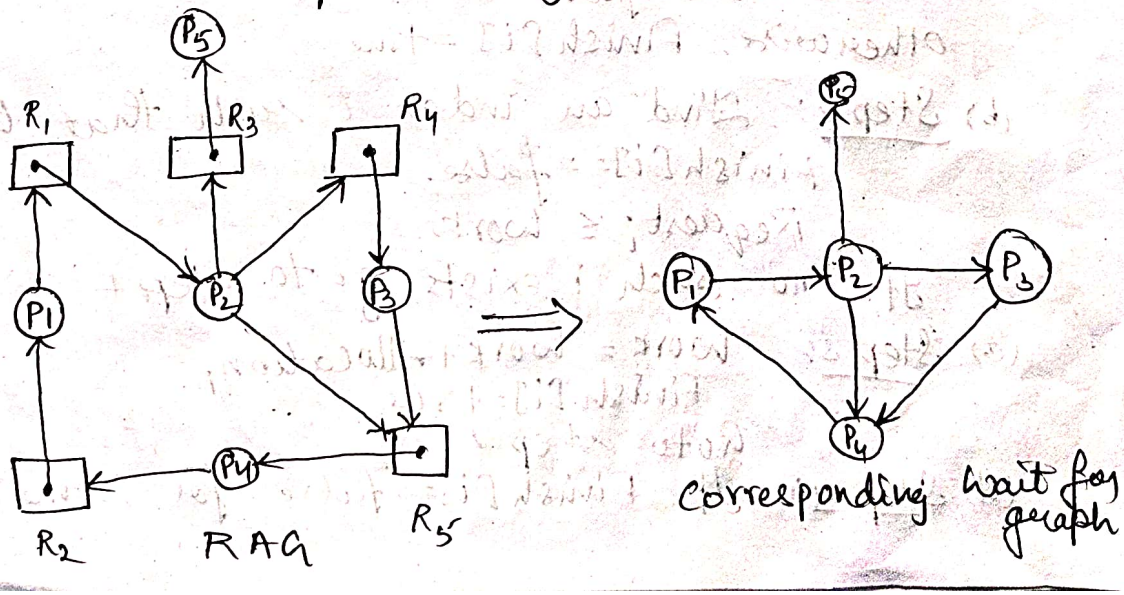
6.1) Single Instance of Each Resource Type:-

WAIT-FOR Graph:

"wait-for" graph is a variant of "resource allocation" graph if used when all resources in the system have only a single instance.

An edge $P_i \rightarrow P_j$ exists in a wait for graph if and only if the corresponding resource allocation graph contains 2 edges $P_i \rightarrow R_q$ and $R_q \rightarrow P_j$ for some resource R_q.

A deadlock exists in the system if and only if the wait-for graph contains a cycle. An algorithm to detect cycle in a graph has to be periodically invoked.



(6.2) Several instances of a Resource type

This algorithm has several data structures like:-

(i) Available: A vector of length 'm' indicates the no. of available resources of each type.

(ii) Allocation:- A $n \times m$ matrix defines the no. of resources of each type currently allocated to each process

(iii) Request:- A $n \times m$ matrix indicates the current request of each process. If Request[i][j] = k, then process P_i is requesting k more instances of resource type R_j.

The following detection algorithm investigates every possible allocation sequence for the processes that remain to be completed.

(a) Step 1:- Let Work & Finish be arrays or vectors of length m and n respectively. Initialize

Work = Available

If $Allocation_i \neq 0$ $\forall i = 0, \ldots n-1$, then

Finish[i] = false.

otherwise, Finish[i] = true.

(b) Step 2:- Find an index i such that both

Finish[i] == false

$Request_i \leq Work$.

If no such i exists, go to step 4.

(c) Step 3:- Work = Work + $Allocation_i$

Finish[i] = true

Goto step 2.

(d) Step 4:- If Finish[i] == false for some i

then the system is in a deadlocked state. Moreover, if Finish[i] == false, then P_i is is deadlocked

(Q) Consider Processes P_0, P_1, P_2, P_3, P_4 & 3 resource types A, B, C. If A has 7, B has 2, & C has 6 instances each. The following snapshot represents the current state of system:-

	Allocation			Request			Available		
	A	B	C	A	B	C	A	B	C
P_0	0	1	0	0	0	0	0	0	0
P_1	2	0	0	2	0	2			
P_2	3	0	3	0	0	0			
P_3	2	1	1	1	0	0			
P_4	0	0	2	0	0	2			

Sol^n:-

Finish

0	1	2	3	4
F	F	F	F	F

Work

0	1	2
0	0	0

$i=0 \Rightarrow$ Process P_0 :- $Request_0 = (0,0,0)$

Finish[0] = false $\Rightarrow$ TRUE

$Request_0 \leq Finish[0]$

$(0,0,0) \leq (0,0,0)$ TRUE

So, $Work = Work + Allocation_0$

$= (0,0,0) + (0,1,0)$

$= (0,1,0)$

Finish[0] = TRUE

Finish

0	1	2	3	4
T	F	F	F	F

Work

0	1	0

$i=1 \Rightarrow$ Process P_1 :- $Request_1 = (2,0,2)$

Finish[1] = False $\Rightarrow$ TRUE

$Request_1 \leq Work$

$(2,0,2) \leq (0,1,0) \Rightarrow$ FALSE

$i=2 \Rightarrow$ Process P_2 :-

Finish[2] is false

$Request_2 \leq Work \Rightarrow (0,0,0) \leq (0,1,0)$ TRUE

So, Work = (0,1,0) + (3,0,3)
= (3,1,3)
Finish[2] = TRUE

	0	1	2	3	4
Finish	F	F	T	F	F

i = 3 ⇒ Process P_3 :-
finish[3] is false ⇒ TRUE
$Request_3 \leq (3,1,3)$
(1,0,0) ≤ (3,1,3)
TRUE

	0	1	2	3	4
Finish	T	F	T	T	F

Work = (3,1,3) + (2,1,1)
= (5,2,4)

i = 4; ⇒ Process P_4 :-
Finish[4] is false
Request[4] ≤ (5,2,4)
(0,0,2) ≤ (5,2,4) ⇒ true

	0	1	2	3	4
Finish	T	F	T	T	T

Finish[4] is true
Work = (5,2,4) + (0,0,2)
= (5,2,6)

i = 0; Process P_1 :-
Request[0] ≤ (5,2,6)
(2,0,0) ≤ (5,2,6)
Work = (5,2,6) + (2,0,0)
= (7,2,6)

We claim that the system is not in deadlocked state. The safe sequence is $<P_0, P_2, P_3, P_4, P_1>$.

Suppose if P_2 makes an addition request for an instance of type C. Then the system is in deadlocked state. Although resources held by P_0 can be reclaimed, the no. of available resources is not sufficient to fulfil the requests of other processes.

7. Deadlock Recovery:-

When a deadlock detection algorithm determines a deadlock exists, there are several alternatives available to recover the system.

7.1 Process Termination:-

To eliminate deadlocks by aborting a process, we can use 2 methods.

(i) ABORT ALL DEADLOCKED PROCESSES:-

* This method aborts all the processes involved in deadlock.
* All the aborted deadlocked processes which have been computed for a long time, their results are to be discarded & have to be recomputed later

(ii) ABORT ONE PROCESS AT A TIME UNTIL THE DEADLOCK CYCE IS ELIMINATED:-

* Each process in cycle has to be aborted and deadlock detection algorithm must be invoked to determine whether any process are deadlocked. This incurs considerable overhead.

7.2 Resource Preemption:-

To eliminate deadlocks, we successfully preempt some resources from processes & give them to other waiting processes till the deadlock breaks. There are 3 issues ~~here~~ to be addressed here:-

(i) SELECTING A VICTIM:-

Which resources & processes are to be preempted? The no. of ~~processes~~ resources a deadlocked process is holding & amount of time the process has thus far consumed are to be considered to minimize the cost.

(ii) ROLLBACK:-

If we preempt a resource from a process, we must rollback the process to some safe state & restart it from that state.

If its difficult to determine the safe state, then simplest solution is ~~com~~ total rollback.

(iii) STARVATION:-

In a system where victim is selected on cost factors, it may happen that same process is always picked as victim. So, the process never completes its task & is starved.

So, make sure that a process can be picked as a victim only a finite (small) no. of times.