Dt ______
Pg ______ B+

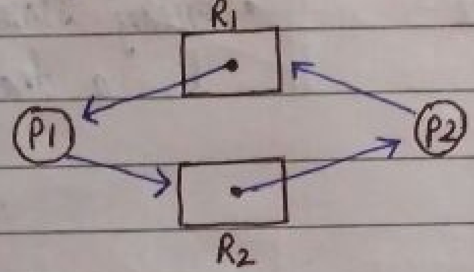
DEADLOCK

resource requested is held by some waiting process ....

It 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 waiting process

→ waiting time is infinite and no progress at all (for all the processes present in deadlock)

example :



Coffman Conditions
(necessary conditions for deadlock to occur) not sufficient

1) Mutual Exclusion Atleast one resource must be non-shareable.

2) Hold & Wait A process ~~can~~ holds a resource & waits for the other resources held by some waiting process.
(applies to all processes in the deadlock)

3) No preemption Resources can't be taken from a process unless it releases the resource.

4) Circular wait set of processes waits for each other in a circular form

Deadlock	Starvation
• no process proceeds	• high priority processes proceed while LPP are blocked
• infinite waiting	• long time waiting, not ∞
• All the resources that in deadlock are held by waiting processes & are not being used.	• Resources are ~~used~~ being used continuously by high priority processes.
• All deadlocks are starvations	• Every starvation need not be a deadlock

Resource Allocation Graph

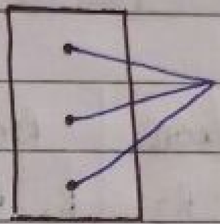
It is a set of

- Vertices
  - Processes $\{P_1, P_2, P_3, \ldots, P_n\}$
  - Resources $\{R_1, R_2, R_3, \ldots, R_n\}$
- Edges
  - $P_i \longrightarrow R_j$ (Request Edge)
  - $R_j \longrightarrow P_i$ (Assignment Edge)

Symbols:

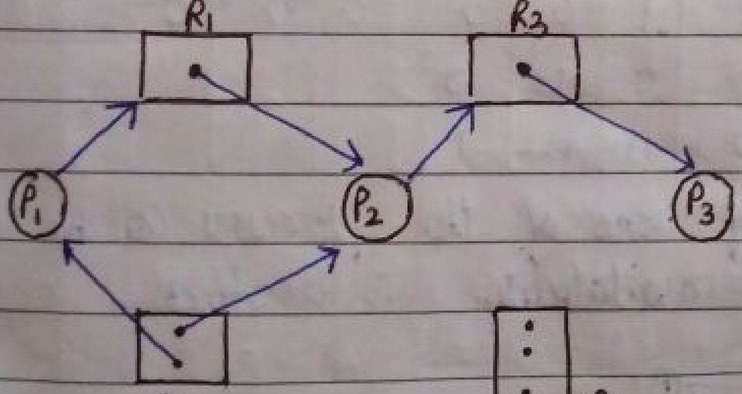
$\boxed{R_i}$ Resource

P_i Process



Instances of a resource

Example:



Here, $P = \{P_1, P_2, P_3\}$

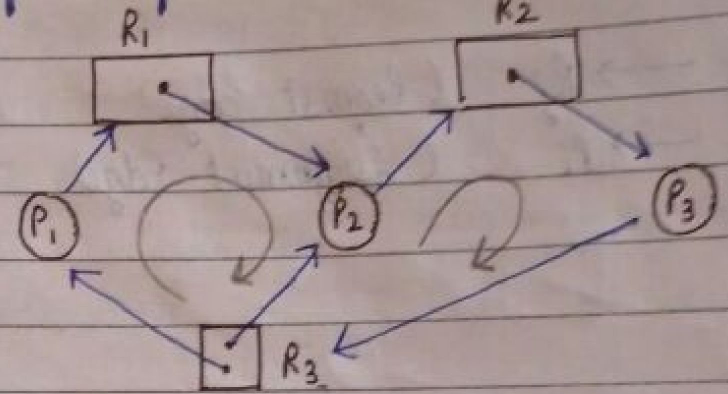
$R = \{R_1, R_2, R_3, R_4\}$

$E = \{P_1 \rightarrow R_1, P_2 \rightarrow R_3,$ Request edge

$R_1 \rightarrow P_2, R_2 \rightarrow P_1, R_2 \rightarrow P_2, R_3 \rightarrow P_3\}$ Assignment edge

Detection of Deadlock

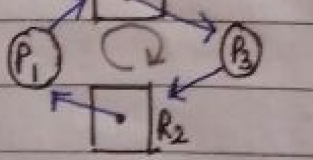
If any of the four coffman conditions is not fulfilled, deadlock doesn't occur
If all four conditions hold true, deadlock MAY occur

→ 

First check for cycle.
If cycle exists, check whether any process has progress

	Allocation			Request			Availability		
	R_1	R_2	R_3	R_1	R_2	R_3	R_1	R_2	R_3
P_1	0	0	1	1	0	0	0	0	0
P_2	1	0	1	0	1	0			
P_3	0	1	0	0	0	1			

Request of none of the processes can be fulfilled based on the availability ∴ Deadlock.

→ 

	Allocation		Request		Availability	
	R_1	R_2	R_1	R_2	R_1	R_2
✓✓ P_1	0	1	1	0		
	1	1			0	0
✓ P_2	1	0	0	0		
					1	0
✓✓ P_3	1	0	0	1	0	0
	1	1			1	1
					1	0 → 2 1

★ If cycle exists & each resource have only one instance, deadlock occurs for sure but if it has multiple instances, deadlock may occur

Deadlock Handling

i) Deadlock Prevention
ii) Deadlock Avoidance
iii) Deadlock detection & Recovery
iv) Deadlock ignorance (Ostrich method)

1) Deadlock Prevention

deadlock can be prevented by violating any of the 4 necessary conditions

→ Removal of mutual exclusion
- shared resources must be used.
- can't be implemented

→ Removal of hold and wait either hold or wait, not both

Three ways:

wait X - A process must acquire all the necessary resources before execution starts (not implementable) - inefficient

hold X - If a process is holding some resources & requesting for additional resource, then it must release the acquired resources first (may lead to starvation)

hold X wait for limited time - A process holding some resources must wait for another resource for a limited time only after which it must release all the resources held by it.

→ Removal of No-Preemption for high priority processes - system processes

Two ways:

- If a process is holding some resources & requesting for another resource that can't be immediately allocated then all the acquired resources will be preempted.

- If a process is ~~is~~ holding some resources & request a resource,
  - If resource is available (allocated)
  - not available (resources ~~held~~ allocated to some other process that is waiting) - preempted from the waiting process & allocated to it.

→ Removal of Circular Wait — implementable

~~To Request Resource~~

All the resources are associated with a no

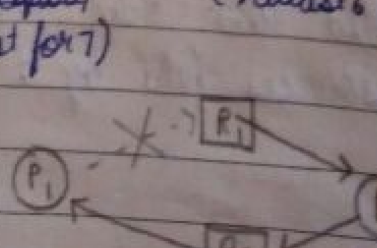
To request resource R_j, a process must first release all R_i s.t. $i >= j$

A process requests & acquires resources in order

eg: Printer (1), CPU (5), Memory (6), CD drive (7)

P_1 : 1, 6 $\xrightarrow{\text{Request(5)}}$ 1 ⟶ 1, 5 ⟶ 1, 5, 6

(can ~~acquire~~/ request for 7) (release 6) (acquire 5) (acquire 6)



cycle never forms

prior info like
- no of res. allocated to a process
- max res required by a process
- order in which resources are required

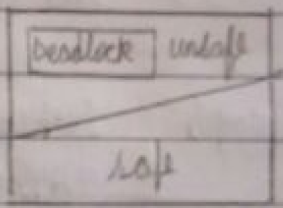
Dt.____ Pg.____ B+

2) Deadlock Avoidance

→ System maintains some database using which it can take decision whether to entertain a request or not s.t it remains in safe state

→ System (kernel) analyse the database to determine whether granting a request can lead to deadlock in future
- if not, then request is granted
- otherwise keep pending until they can be granted (process may face long delay for obtaining a res.)

A safe seq. is possible ← Safe state: if system can allocate all the resources requested by all the processes without entering into deadlock otherwise → unsafe state (cycle +nt)



Initially system is in safe state ~~A request~~

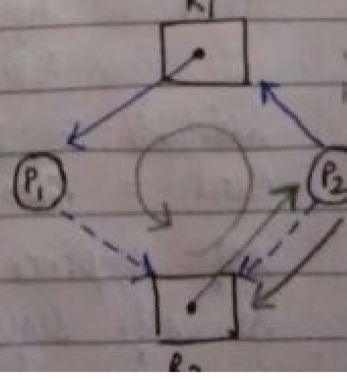
→ Deadlock avoidance using RAG (used when all resources have only 1 instance)
- $P_i \rightarrow R_j$ (request edge)
- $R_j \rightarrow P_i$ (assignment edge)
- $P_i \dashrightarrow R_j$: P_i may request R_j in future (claim edge)

if granted

after completion

Resources must be claimed in advance So, in the beg. all edges in RAG are claim edges

Example: If P_i requests R_j then request edge can only be converted into assignment edge if it doesn't form a cycle
Here, request of P_2 can't be granted



unsafe state may enter in deadlock

For safe state, atleast one safe sequence must exist.

Banker's Algorithm
used when resources have multiple instances.

It requires:
- Max requirement of instances of each resource by each process. (2D array)
- Instances of each resource held by each process (2D array)
- Total instances of each resource available in the system (1D array)
- More instances required by each process (2D array)

Example: Snapshot at T_0

	Allocation				Max				Available				Need			
	A	B	C	D	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	0	0	0	0
✓ P_1	1	0	0	0	1	7	5	0	1	5	3	2	0	7	5	0
✓ P_2	1	3	5	4	2	3	5	6	2	8	8	6	1	0	0	2
✓ P_3	0	6	3	2	0	6	5	2	2	14	12	8	0	0	2	0
✓ P_4	0	0	1	4	0	6	5	6	2	14	12	12	0	6	4	2
	2	3	10	12					3	14	12	12				

Need matrix? Available res.?
Is system in safe state? If yes find safe seq.

Total :

A	B	C	D
3	14	12	12

Available = Total - allocated
Need = ~~Alloc~~ Max - allocated
safe sequence : P_0 P_2 P_3 P_4 P_1
As safe sequence exists ∴ It is in safe state

All requirements of all the processes are fulfilled
↳ (no cycle test)
↳ no deadlock

Example :

Snapshot at time T_0

	Allocation				Max				Available				Total			
	A	B	C	D	A	B	C	D	A	B	C	D	A	B	C	D
P_0	2	0	0	1	4	2	1	2	3	3	2	1	12	12	8	10
P_1	3	1	2	1	5	2	5	2								
P_2	2	1	0	3	2	3	1	6								
P_3	1	3	1	2	1	4	2	4								
P_4	1	4	3	2	3	6	6	5								
	9	9	6	9												

Available Resource instances = Total - Total (Allocation)
Need = Max - Allocation

	Allocation				Max				Available				Need			
	A	B	C	D	A	B	C	D	A	B	C	D	A	B	C	D
✓ P_0	2	0	0	1	4	2	1	2	3	3	2	1	2	2	1	1
✓ P_1	3	1	2	1	5	2	5	2	5	3	2	2	2	1	3	1
✓ P_2	2	1	0	3	2	3	1	6	6	6	3	4	0	2	1	3
✓ P_3	1	3	1	2	1	4	2	4	7	10	6	6	0	1	1	2
✓ P_4	1	4	3	2	3	6	6	5	10	11	8	7	2	2	3	3
									12	12	8	10				

safe sequence : $P_0 \rightarrow P_3 \rightarrow P_4 \rightarrow P_1 \rightarrow P_2$
As safe sequence exists , system is in safe state

If request from P_1 arrives for (1, 1, 0, 0), can this request be granted immediately?

Request must not exceed the need.

(1, 1, 0, 0) ≤ (2, 1, 3, 1)

Check if Request ≤ Available

(1, 1, 0, 0) ≤ (3, 3, 2, 1) => True

∴ System pretends to grant the request & checks whether system remains in the safe state after granting the request.

	Allocation				Max				Available				Need			
	A	B	C	D	A	B	C	D	A	B	C	D	A	B	C	D
✓ P_0	2	0	0	1	4	2	1	2	2	2	2	1	2	2	1	1
✓ P_1	4	2	2	1	5	2	5	2	4	2	2	2	1	0	3	1
✓ P_2	2	1	0	3	2	3	1	6	5	5	3	4	0	2	1	3
✓ P_3	1	3	1	2	1	4	2	4	6	9	6	6	0	1	1	2
✓ P_4	1	4	3	2	3	6	6	5	10	11	8	7	2	2	3	3
									12	12	8	10				

safe sequence : $P_0 \rightarrow P_3 \rightarrow P_4 \rightarrow P_1 \rightarrow P_2$.

As safe sequence exists, system remains in safe state after granting the request

Therefore, request can be granted immediately.

If request from P_4 arrives for (0, 0, 2, 0), can it be granted immediately?

Request i.e (0, 0, 2, 0) ≤ Need i.e. (2, 2, 3, 3)

Check if Request ≤ Available
(0, 0, 2, 0) ≤ (3, 3, 2, 1) ⇒ True

System pretends to grant the request:

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

safe sequence: Since, need of none of the processes can be fulfilled with available resources, safe sequence doesn't exist ⇒ system is in unsafe state

Therefore, request can't be granted immediately

Algorithm

Input : - Processes
- any 2 out of (Max, Need, Allocation matrix)
- any 1 out of (Available, Total)

Safety Algorithm :

1. Let Work & Finish be vectors of length m & n.
   Work = Available
   Finish [i] = false for $i = 0, 1, \ldots, n-1$
   n = no. of processes

2. Find i such that :
   Finish [i] = false
   $Need_i \leq Work$
   If no such i exists, go to step 4.

3. Work = Work + Allocation
   Finish [i] = true
   go to step 2.

4. If finish [i] = true for all i, then system is in safe state
   else, it is in unsafe state.

Time Complexity = $O(mn^2)$
m = no. of resources
n = no. of processes.

Dt.______
Pg.______ B+

Resource - Request Algorithm for Process P_i

1. Request = request vector for process P_i
If Request $\leq$ Need,
go to step 2
Otherwise,
raise error as process has exceeded its max claim.

2. If Request $\leq$ Available,
go to step 3
Otherwise,
P_i must wait as resources are not available.

3. Pretend to allocate requested resources to P_i by modifying:
   - Available = Available − Request;
   - $Allocation_i$ = $Allocation_i$ + $Request_i$;
   - $Need_i$ = $Need_i$ − $Request_i$;

4. Find whether system is in safe state after allocation
   - If safe $\Rightarrow$ resources are allocated to P_i
   - If unsafe $\Rightarrow$ P_i must wait & the old resource-allocation state is restored.

3) Deadlock Detection & Recovery

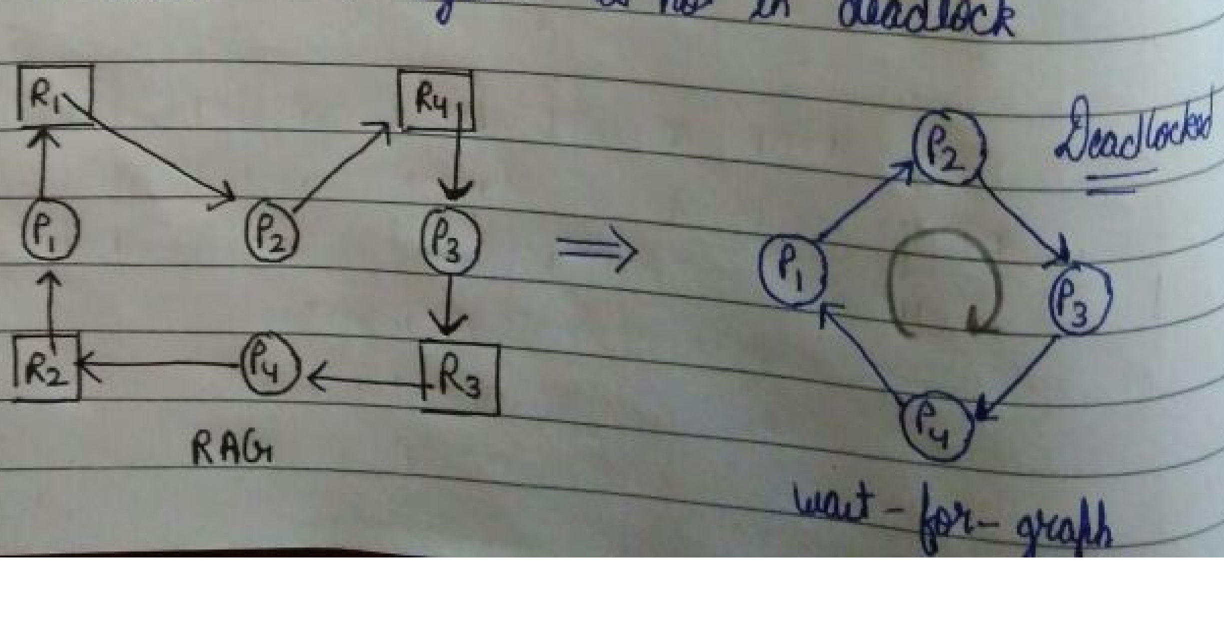
→ Allow the system to enter into deadlocked state
→ System checks for deadlock periodically.
→ Deadlock is detected using deadlock detection algorithms & recover the system using some recovery techniques

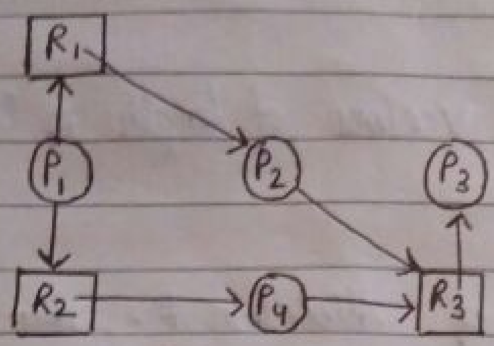
Deadlock detection algorithms

i) Wait-for-graph (for single instances)
- It is a RAG which consists of only processes as its nodes.
- set of
  - nodes - Processes (P_i)
  - edges - $P_i \rightarrow P_j$
    (P_i is waiting for the resource held by P_j)

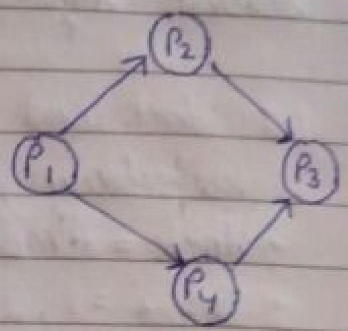
This algorithm checks for a cycle in the wait-for-graph
- If cycle is detected ⇒ system is in deadlock
- otherwise ⇒ system is not in deadlock

Example:





RAG



wait-for-graph
(no cycle, no deadlock)

2) For multiple Instances

It requires:
- Total no. of instances available for each resource in the system (1D array)
- Instances of each resource held by each process (2D array)
- Current request of ~~es~~ instances of each resource by each process (2D array)

Example

	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	0	1	0
✓ P_2	3	0	3	0	0	0	3	1	3
✓ P_3	2	1	1	1	0	0	5	2	4
✓ P_4	0	0	2	0	0	2	5	2	6
							7	2	6

safe sequence : $P_0 \to P_2 \to P_3 \to P_4 \to P_1$
System is not in deadlock

	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	0	1	0
P_2	3	0	3	0	0	1			
P_3	2	1	1	1	0	0			
P_4	0	0	2	0	0	2			

safe seq. : $P_0 \to$ further no request can be granted
P_1, P_2, P_3, P_4 are in deadlock

Detection Algorithm

1) Let Work and Finish be vectors of length m & n respectively, Initialize
   - Work = Available
   - For $i = 1, 2, \dots n$, if Allocation $\neq 0$, then Finish [i] = false, else Finish [i] = true

2) Find an index i s.t. both
   - Finish [i] == false
   - $Request_i \leq Work$

   If no such i exists, go to step 4

3) Work = Work + $Allocation_i$
   Finish [i] = true
   go to step 2

4) If Finish [i] == false for some $1 \leq i \leq n$, system is in deadlock state & P_i is deadlocked

Time Complexity = $O(mn^2)$

Recovery Techniques

Recovery Techniques are used to recover a system from deadlock when detected by some deadlock detection algo.

Two approaches:

1) Process Termination (Pessimistic Approach)

i) Abort all deadlocked processes.
- All the partially completed processes ~~not~~ in deadlock are aborted even if they have computed for a long time
- This leads to high expenses

ii) Abort one process at a time
- Abort one process at a time and run deadlock detection algorithm. If ~~proc~~ system is still in deadlocked state, abort another process and continue till deadlock is removed.
- Overhead of executing deadlock detection algo multiple times
- Process to be aborted is decided on the basis of:
  - priority of processes.
  - time for which the process has computed
  - time required for completion.
  - No. & type of resources utilised
  - No. & type of resources needed.

Pg

3) Resource Preemption (optimistic approach)
Pre-empt some resources from processes & give those resources to other processes until deadlock cycle is broken.
Three issues:

i) Selecting a victim
resources to be preempted must be decided in accordance with the cost minimization.

ii) Rollback
The process from which resources are preempted must be either
- rollbacked completely & restarted again or
- rollbacked to a safe state

iii) Starvation
If a process is picked as a victim multiple times, it may lead to starvation.
Therefore a process must be selected as a victim ~~for~~ only a finite no of times.

4) Deadlock Ignorance

- It is used in the systems in which deadlock occurs rarely. Deadlock is completely ignored and rather than spending resources on it, more frequently occurring issues are handled such as compiler errors, system crashes due to hardware failure, OS bugs.
- This approach is used in UNIX & Windows
- If deadlock occurs, it is handled by rebooting