Deadlocks

→ If a & more processes are waiting on happening of event, but that event doesn't happen

↓

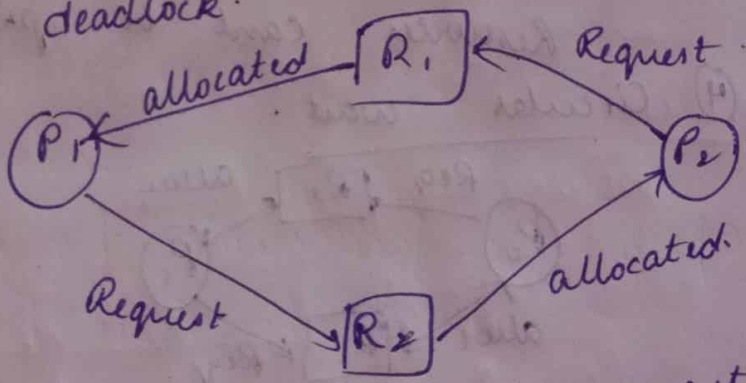
deadlock & processes are in deadlock state

Technically.

Process P_1 has semaphore S_1 waiting for S_2

Process P_2 has semaphore S_2 waiting for S_1

→



P_1 is having R_1 and P_1 will execute & release R_1 only if it gets R_2.

(But R_2 is already taken by P_2)

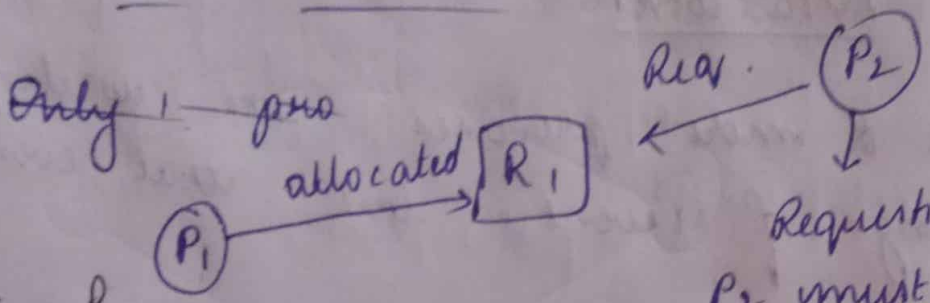
~~Solved~~ (Vice versa with P_2).

→ Deadlock Conditions

① Mutual exclusion
② no preemption
③ hold & wait
④ Circular wait.

} necessary & sufficient Conditions for deadlock

① mutual exclusion.



Requesting process P_2 must be delayed until the resource has been released.

No Sharing of ~~process~~ resources.

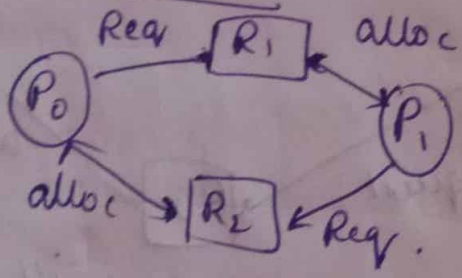
② Hold & wait.

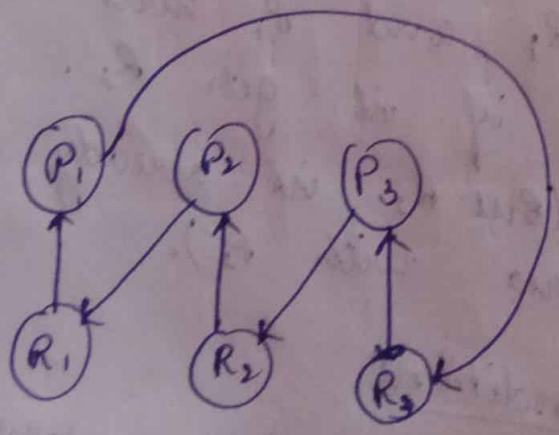
P_1	P_2
holds R_1	holds R_2
Req R_2	Req R_1

③ no preemption

Resources cant be preempted in middle

④ Circular wait.





Resource allocation graph

→ Convinient & efficient way to represent the state of the system.
(how resources are allocated to processes & how processes are have assigned the multiple resources)

→ To represent if in our system there is a deadlock & not ↓ RAG is most suitable.

RAG contains → Vertex
→ Edge.

Vertex → (All processes running in our system Represented using vertices) circled (process vertex)
↓
(Process vertex Rep:- 0)

Process Vertex

Resources Vertex.
(All Resources available in our sys (Res) ( ))

↓
Single instance
Eg:- CPU, monitor
[.]

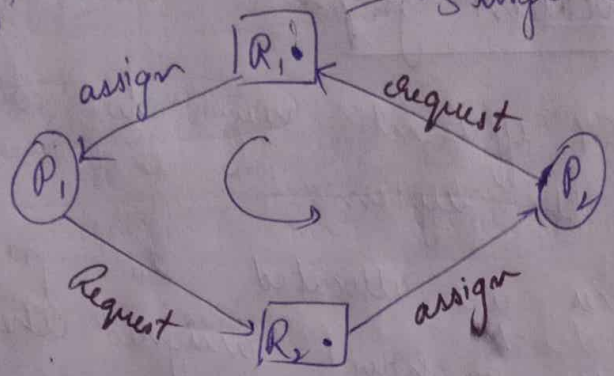
Multiple instance
Eg:- Register, printer etc
[. . . .]

Edges :-

Edges
↓
Assigned edge
(edge towards process)
(P) ↑ [R]

Request edge.
(towards resource)
(P) ↓ [R]

Eg:-



→ Single instance resources (R_1, R_2 single)

Advantage of this representation

We can check if the system has a deadlock or not. (by looking RAG)

Circular wait ✓ (cycle)

✓✓

Deadlock exists. (Graph is simple so we can analyse.

Method

Processes → (2) P_1, P_2

	Allocate		Request	
	R_1	R_2	R_1	R_2
P_1	1	0	0	1
P_2	0	1	1	0

Availability:-(currently available) R_1 R_2 (0,0)

↓

By seeing this availability we need to check if we can fulfill either P_1 or P_2's Request

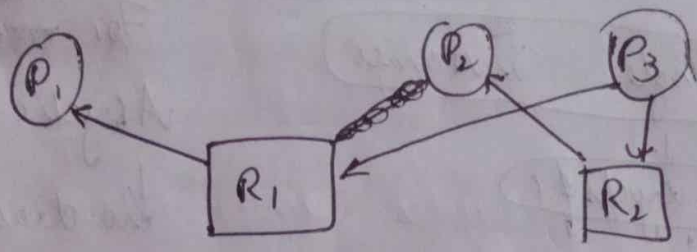
↓

We cant fulfill P_1 } So as we cant
" " " P_2. } fulfill anyone

↓

Deadlock is there.

Q



no cycle here
(if we start from
1 point we can't
reach that pnt

~~Acyclic~~
Acyclic

	Allocate		Req	
	R_1	R_2	R_1	R_2
P_1	1	0	0	0
P_2	0	1	0	0
P_3	0	0	1	1

Availability (0,0)

↓

P_1 will execute (no Req)

↓

P_1 terminates → R_1 is taken back

Availability (1,0)

P_2 will also be executed (no Req)

↓

P_2 executes & terminates → R_2 taken back.

Availability (1,1)

↓ (So all processes are executed)
They can be used to fulfill P_3.

So No Deadlock

(no infinite wait)

↳ (event happens)

↳ (finite wait only)

Finite wait: Starvation

Infinite wait: Deadlock

For Single Instance resources,

→ If RAG has a cycle (i.e has a circular wait → cyclic → has a deadlock

→ For single instance

↓

if cyclic

↓

deadlock exists for sure

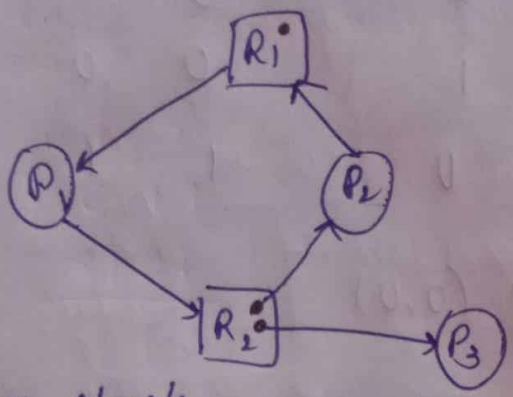
For ~~multi~~ Instance

Acyclic

↓ no deadlock (for sure)

Multi Instance RAG

→ Qsn



R_1 → 1 instance

R_2 → 2 "

We can check manually,

↓

P_3 has no Req so it can execute

↓

One inst. of R_2 is released

↓

which can be given to P_1

↓

P_1 has R_1, R_2 → it executes & Releases R_1

↓

P_2 executes

So no Deadlock

Method

	Alloc		Request	
	R_1	R_2	R_1	R_2
P_1	1	0	0	1
P_2	0	1	1	0
P_3	0	1	0	0

R_1 R_2

Availability (0, 0)

↓

P_3 is fulfilled

↓ Availability (0, 1)

then R_2 is assigned to P_1

↓ (Availability (0,0)

then finally

Availability (1, 1)

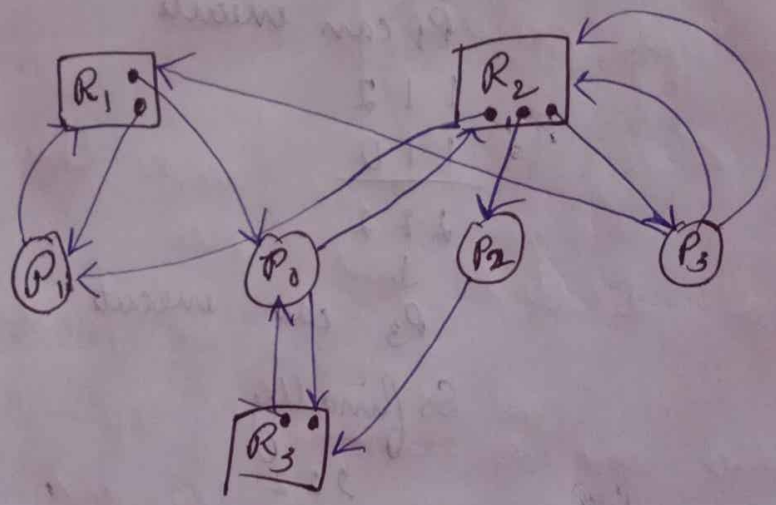
↓

R_1 given to P_2.

(P_3 P_1 P_2)

→ Here there is a circular wait, But no deadlock, as it is multi instance (so no guarenty for deadlock.) ↳ if circular wait exists

→ (a)



method.

	Alloc			Req		
	R_1	R_2	R_3	R_1	R_2	R_3
P_1	1	1	0	1	0	0
P_2	0	1	0	0	0	1
P_3	0	1	0	1	2	0
P_0	1	0	1	0	0	1

Availability (0 0 1)

(R_1 R_2 R_3)

with Availability, we can fulfill P_2

↓

So P_2 executes and releases

```
  0 0 1
  0 1 0
 -------
  0 1 1
```

↓

P_0 can execute

↓

```
  0 1 1
  0 1 0 1
 ---------
  1 1 2
```

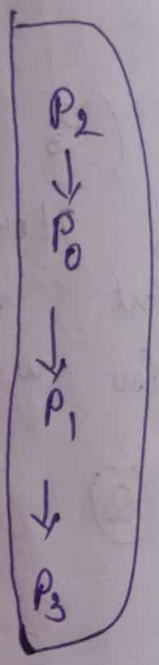
↓

P_1 can execute

```
  1 1 2
  1 1 0
 -------
  2 2 2
```

↓

P_3 can execute.



So finally

```
  2 2 2
  0 1 0
 -------
  2 3 2
 -------
```

There is no deadlock in sys.

Sequence :- $P_2 \rightarrow P_0 \rightarrow P_1 \rightarrow P_3$.

⟶ If RAG has multiple instances and contains a cycle

↓

no gaurenty for deadlock

Deadlock Avoidance & Deadlock handling methods & prevention

→ 4 methods to handle deadlock

most widely used.

① Deadlock Ignorance (Ostrich method)

↓

just ignore deadlock.

(used widely) Used in Windows, Linux etc

→ When deadlock occurs in system, just ignore it.

(hang)

(Deadlock Occurs very rarely)

Windows

if we add more code regarding deadlock handling & removing, its performance gets affected. ↓ speed ↓

→ (So we just ignore deadlock) ↑ as only rare occurance.

→ Why we are ignoring?

We don't want to affect performance / speed.

*

② Dead lock prevention.

→ Before deadlock occur we try to find some solutions.

for deadlock.

4 conditions, mutual exclusion
no preemption
hold & wait
Circular wait

Either try to remove all 4 conditions from System / try to discard one of the 4 conditions.

Either 1 condition → F
or
all conditions → F

Mutual exclusion

↓

what ever resources we are using, they have to be non sharable.
(cannot be used at same time by multiple processes)

↓

If we make this condition F then deadlock can't occur

↓

i.e. if we make our system, such that all resources (eg CPU...) can be shared by more than 2-3 processes

↓

So deadlock can be removed

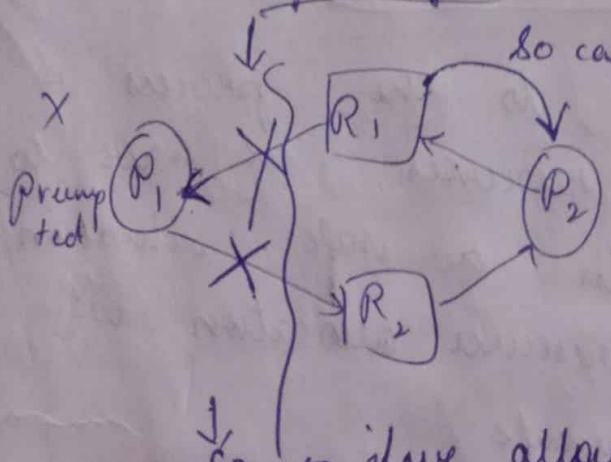
Problem with mutual exclusion:-

(Not all the resources can be made sharable Eg:- printer, tape drive,.... these cant be made sharable)

↓

they are made non sharable.

~~No~~ No preemption.



By using Time Quantum / time stamps.

→ if we allow preemption no Deadlock

Hold & Wait

(In order to prevent DL, no hold & wait)

→ Before a process gets started provide all the resources it wants

So further (no hold & wait)

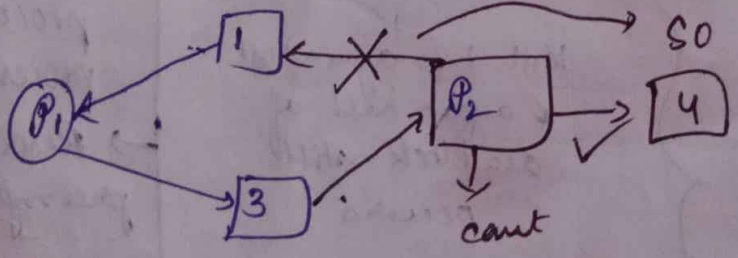
Circular Wait

First, Order all the resources by giving numbering

Eg:-

Printer	Scanner	CPU	Register...
1	2	3	4

Condition is, A process can request only in increasing order.



So no circular wait

↓

So no deadlock

Dead lock avoidance

→ giving resources to the process & we have to while giving resources check if it is a safe situation/not

→ So for every resource allocation we check for safe/ unsafe

(Done By Bankers algorithm)

↓

Dead lock Avoidance.

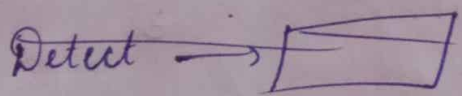
Deadlock Detection & Recovery

↓

Detecting for a deadlock & then trying to recover the system

→ Complex.

→ Will make system complex



Detect → no deadlock

Detect → Deadlock → recovery

Recovery Techniques.

- Kill 1 & again test & again kill if deadlock still occurs
- Kill the process in/ processes
- Resource preemption.

Bankers Algo

→ Deadlock avoidance algo

↓

→ As we have to provide info to OS well in hand about upcoming processes, their Request of resources & their delay & count. So that OS decides on which process sequence should be executed / waited.

→ Also used for Deadlock detection.

Total A = 10 (-7) B = 5 (-2) C = 7 (-5)

process	Allocation A B C	Max need A B C	Available A B C	Rem need A B C
P_1	0 1 0	7 5 3	3 3 2	7 4 3 P_1
P_2	2 0 0	3 2 2	5 3 2	1 2 2 P_2
P_3	3 0 2	9 0 2	7 4 3	6 0 0 P_3
P_4	2 1 1	4 2 2	7 4 5	2 1 1 P_4
P_5	0 0 2	5 3 3	7 5 5	5 3 1 P_5
	7 2 5		10 5 7	
			(Total - alloc)	(max need - Allocation)

safe & unsafe?

↓ ↓

No DL DL

↓

So find Safe sequence (sequential execution of processes so that DL never occurs → that order)

→ P_1 X P_2 ✓ P_3 X P_4 ✓ P_5 ✓ P_1 ✓ P_3 ✓

A = 5 3 2 A = 7 4 3 A = 7 4 5 A = 7 5 5 A = 10 5 7

safe sequence.

P_2 P_4 P_5 P_1 P_3.

current Availability $\geq$ current Rem. need.
to allocate resources.

→ Here we took static resources, processes, but in real life it will be dynamic.
Processes can come at runtime / will not come at same time.

→