

These four are the essential condition for deadlock.

**Mutual exclusion** 🡪 Only 1 process can hold a resources at a particular time. A resource cannot be shared when a Process is holding that particular resource.  
(Similar to the critical section problem)

**Hold and wait**Me 🡪 holding (bat) waiting(ball)  
Brother 🡪 holding (ball) waiting(bat)

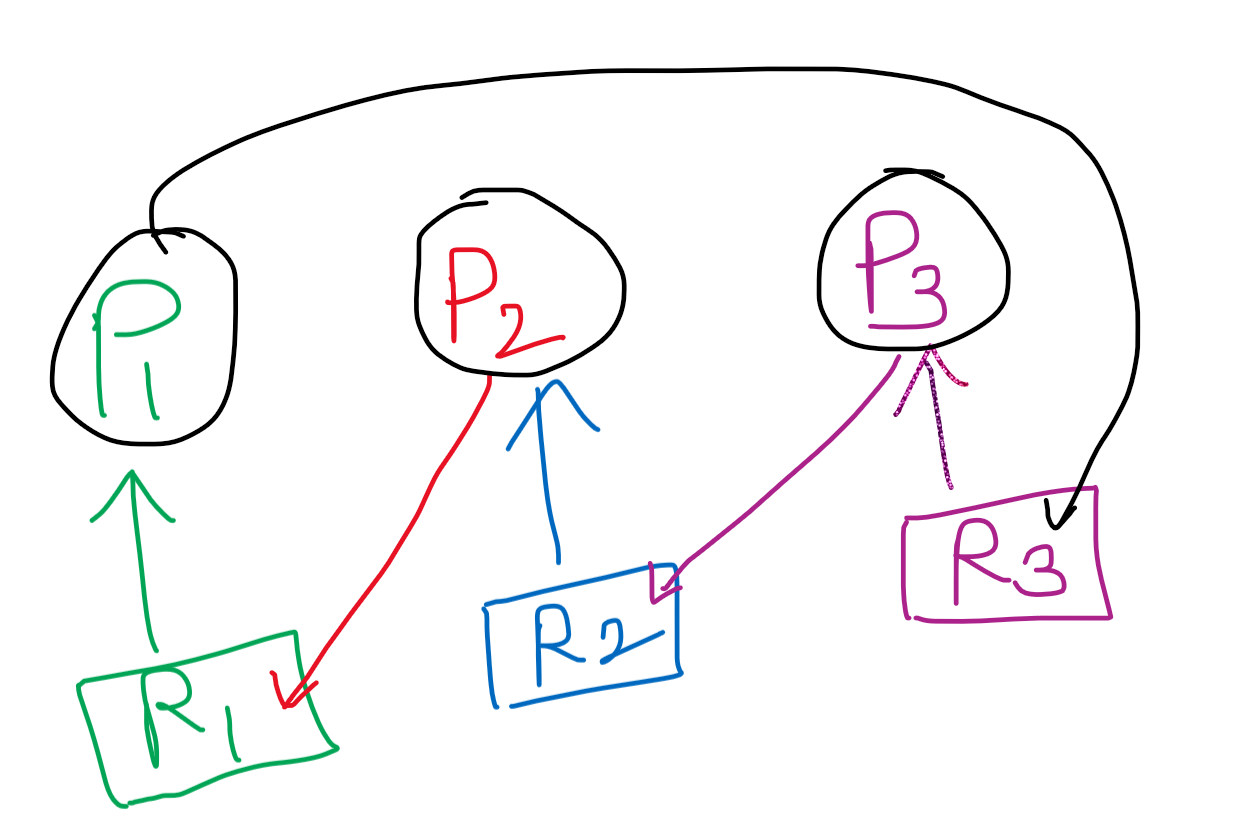
P0 🡪 holding R1 requesting R2  
P1 🡪 holding R2 requesting R1

**No pre-emption :**P1 🡪 Holding resource R1, This R1 cannot be taken forcibly and assign to P2. Once P1 completes the usage of R1 and releases, then only P2 can use R1

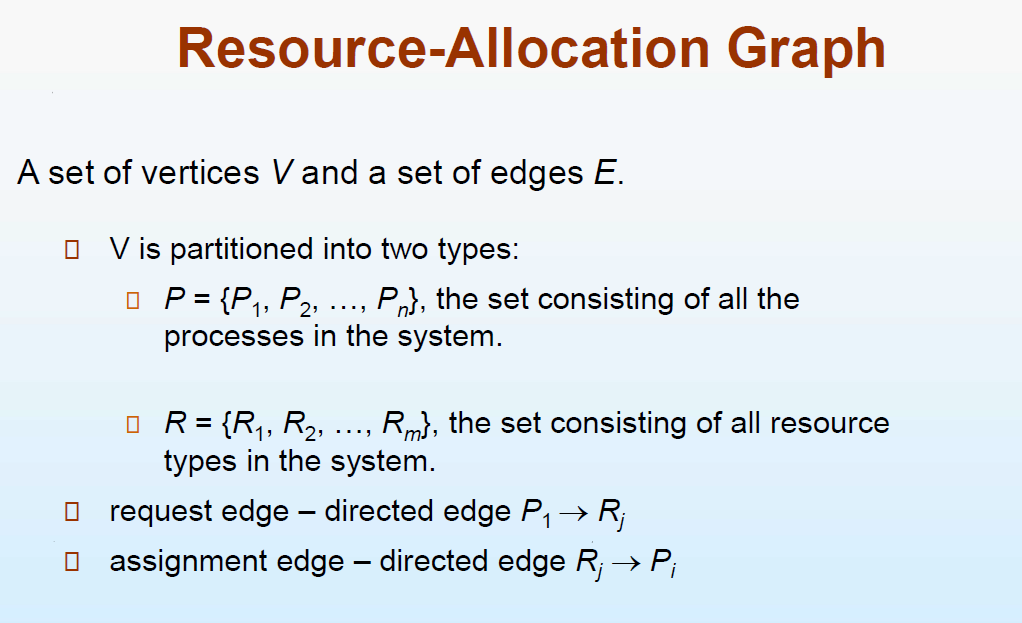
**Circular wait:**P1 🡪 holding R1 requesting R2  
P2 🡪 holding R2 requesting R3  
P3 🡪 holding R3 requesting R1



It moves in a circular fashion.

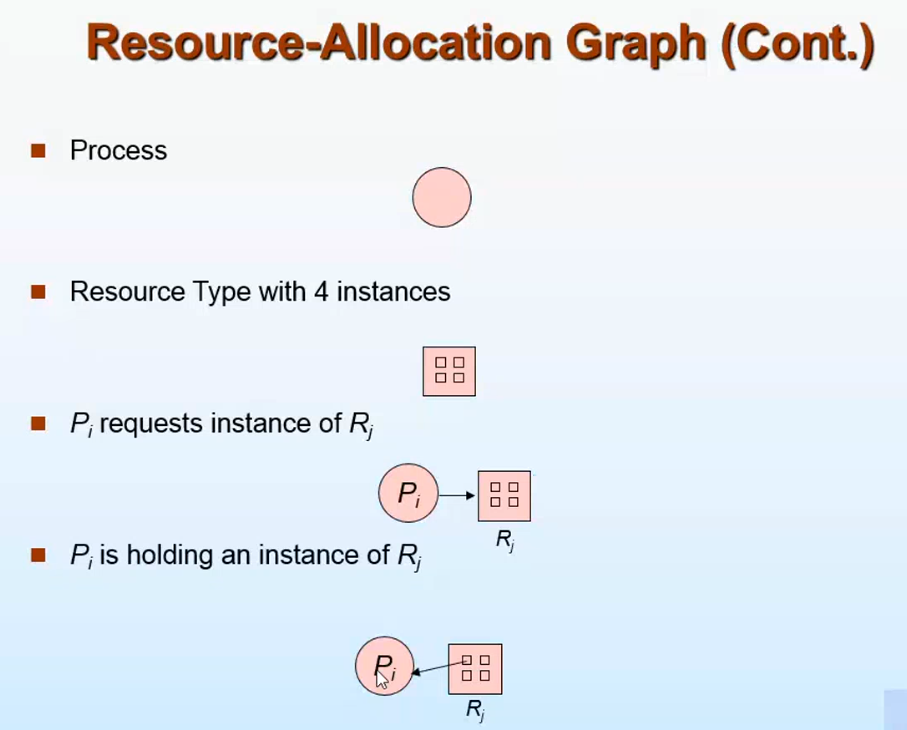


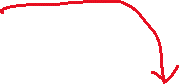
One way to find if a system has undergone deadlock 🡪 by means of resource allocation graph



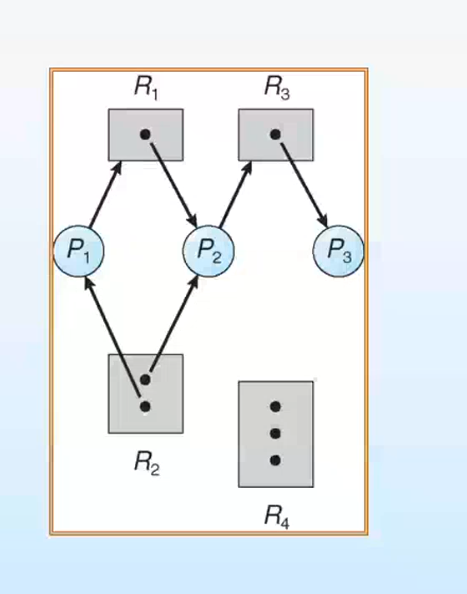
Single instance (circular wait) in RAG 🡪 Deadlock  
Single instance (no circular wait) in RAG 🡪 No Deadlock







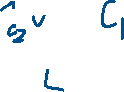
**Request edge** (from process to the resource)  
**Assignment edge** (assigns the whole resources to the process/assigns only a particular instance of the resource to the process)



**The dead lock condition is not there** . P3 is not requesting any resource and holding R3. After using R3, P3 releases R3.  
So P2 which is requesting R3 will be granted and P2 uses one instance of R2,and R3, R1 and then releases R1,R2 and R3.  
Now P1 can easily uses one instance of R2 and R1, and releases both R1 and R2

How to convert a graph into deadlock condition

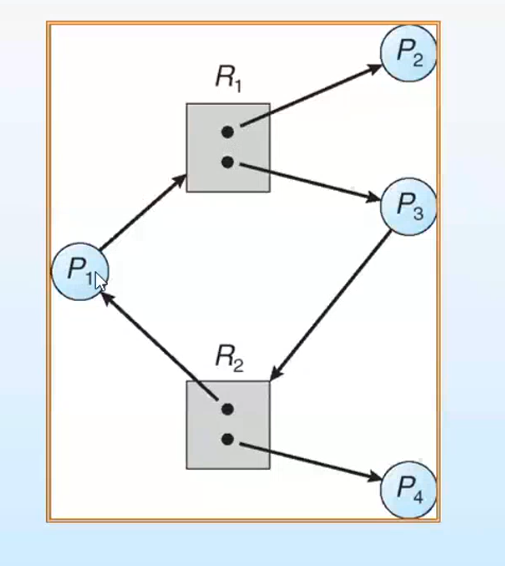




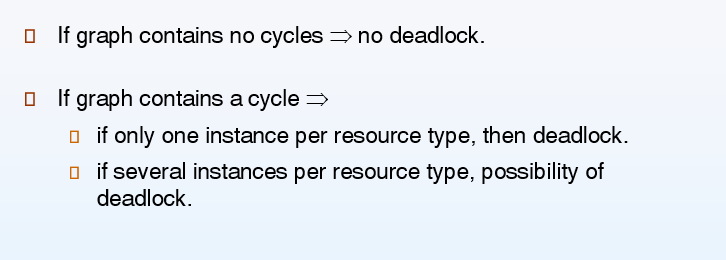
There is a cycle (and it is in deadlock). It is not necessary that if a cycle appears there is dead lock condition.



***Is the below graph a dead lock ???***



**No,** since P3 and P4 is having R3 and R2, P3 and P4 are not requesting anything. So P3 and P4 uses R3 and R2 and releases R3 and R2.  
***This is because there are multiple instance of R2 and R3, For eg: If there is only one instance of R3 and R2 🡪 Deadlock***

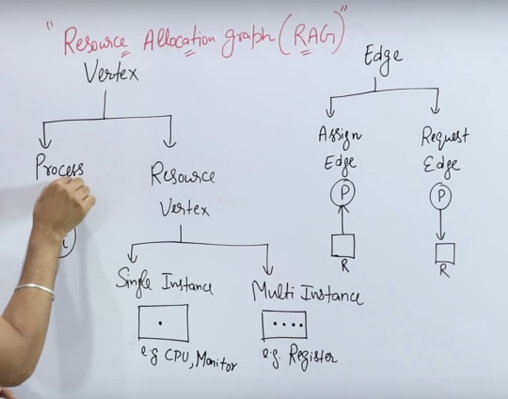


In the graph there is cycle, but not ends in dead lock.

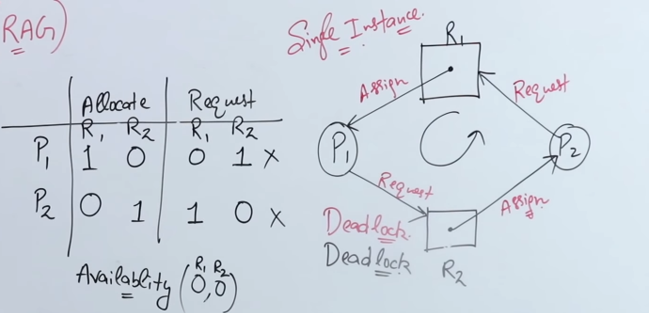


SO it is clear that only cyclic condition is not enough to say that it ends in dead-lock. But if a dead lock is there, definitely there will be a cycle.



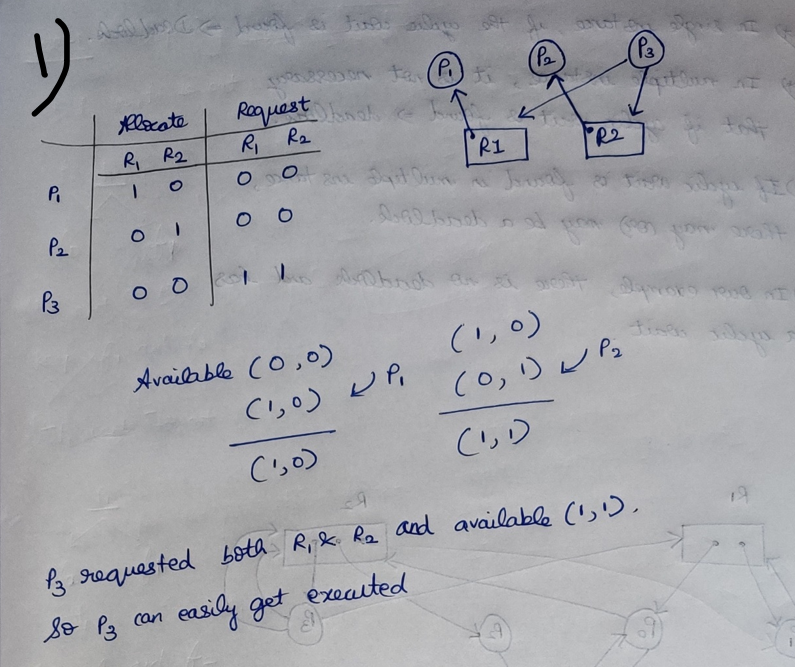


**1)**



**P1 is holding R1 and requesting for R2 (hold and wait)  
P2 is holding R2 and requesting for R1 (hold and wait)**

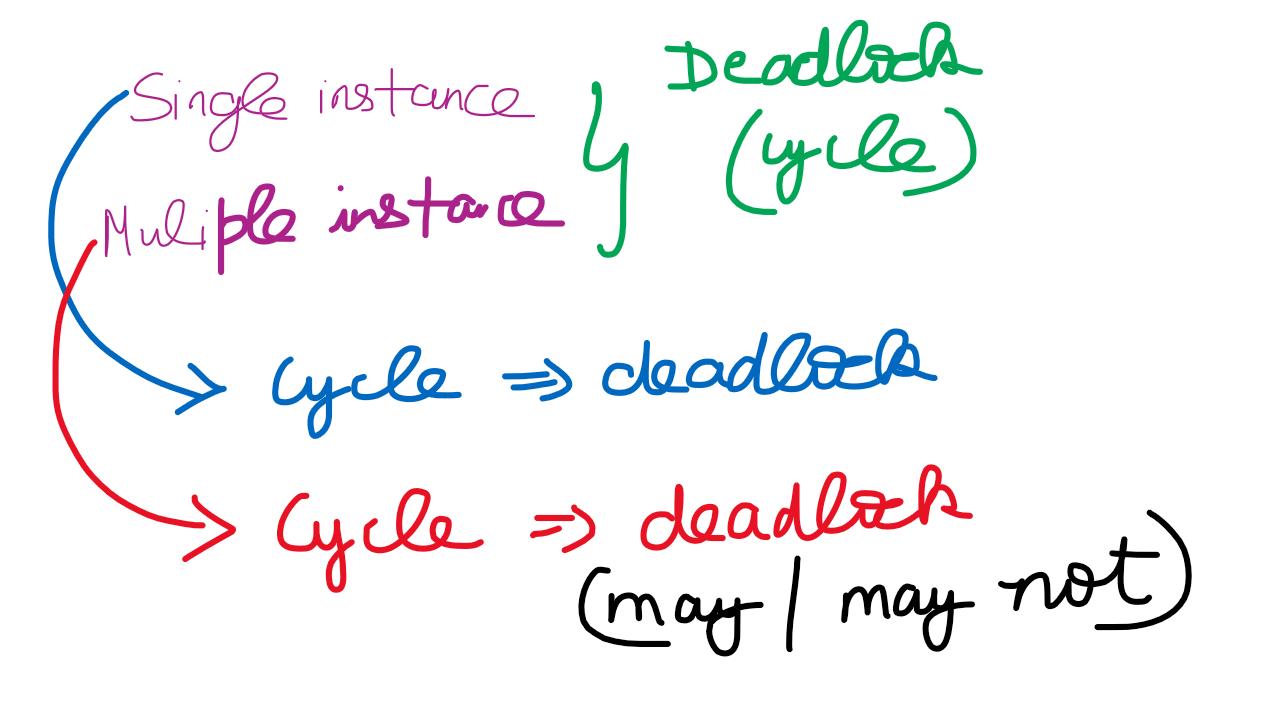
**Deadlock (No starvation)**

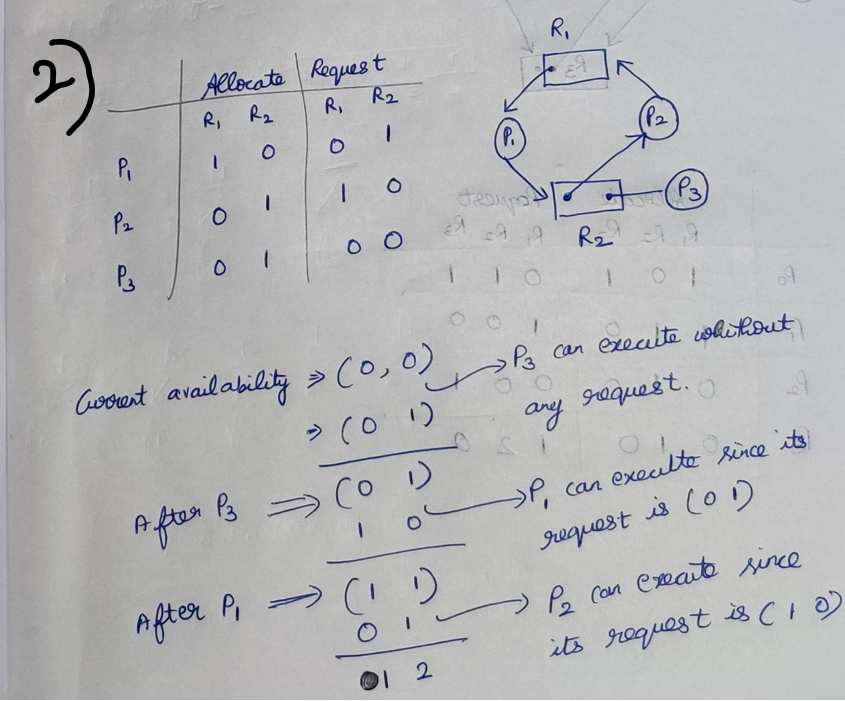


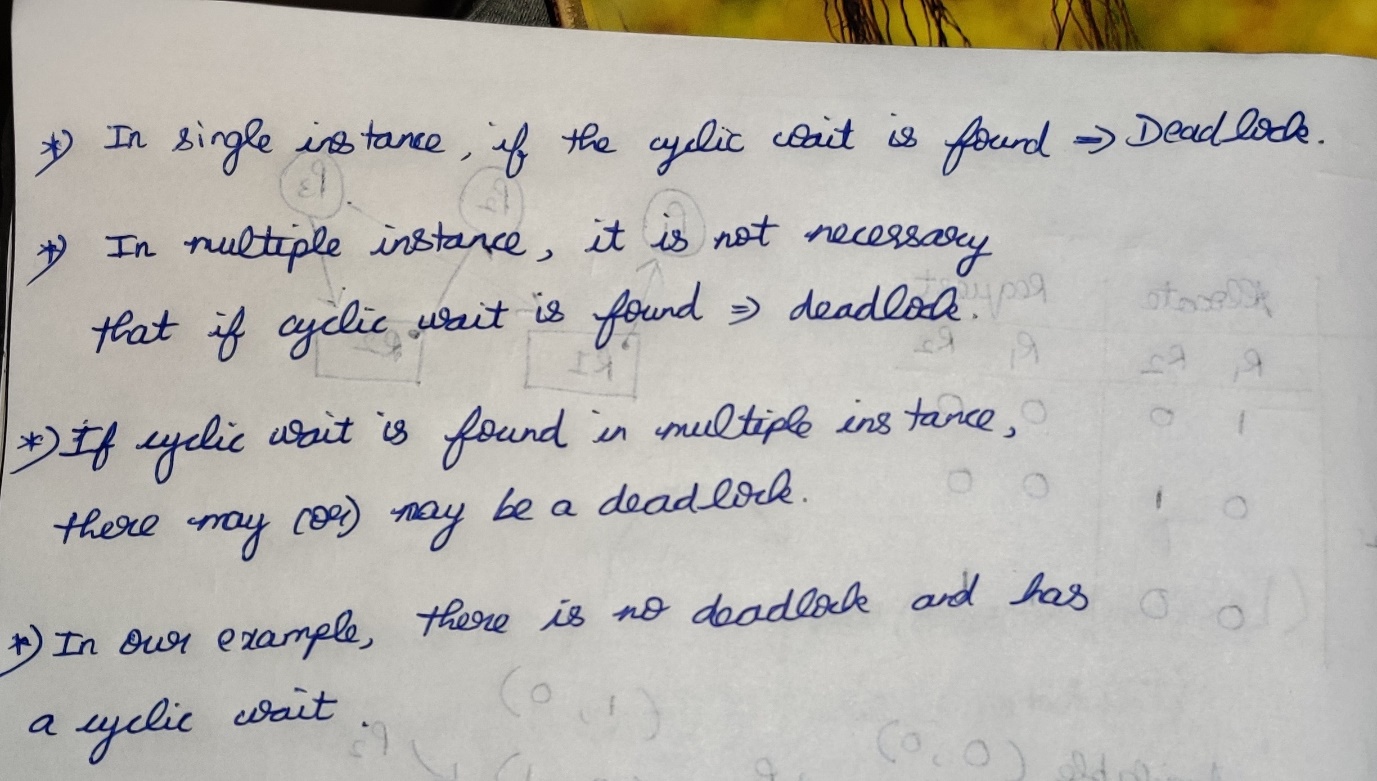
If a process is not requesting anything then that process can execute. After executing that process will be terminated.

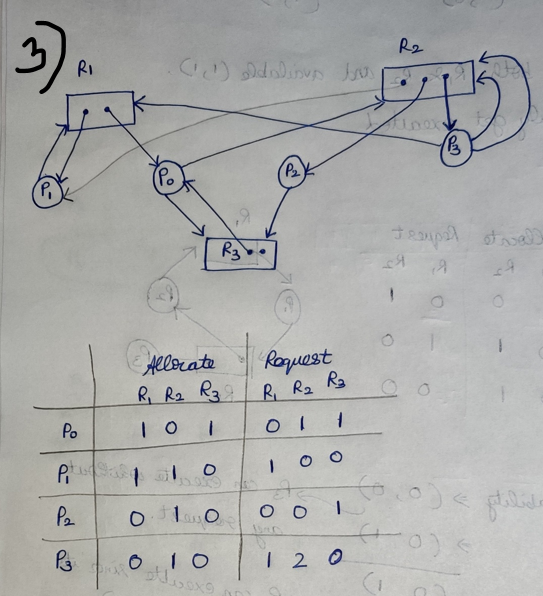
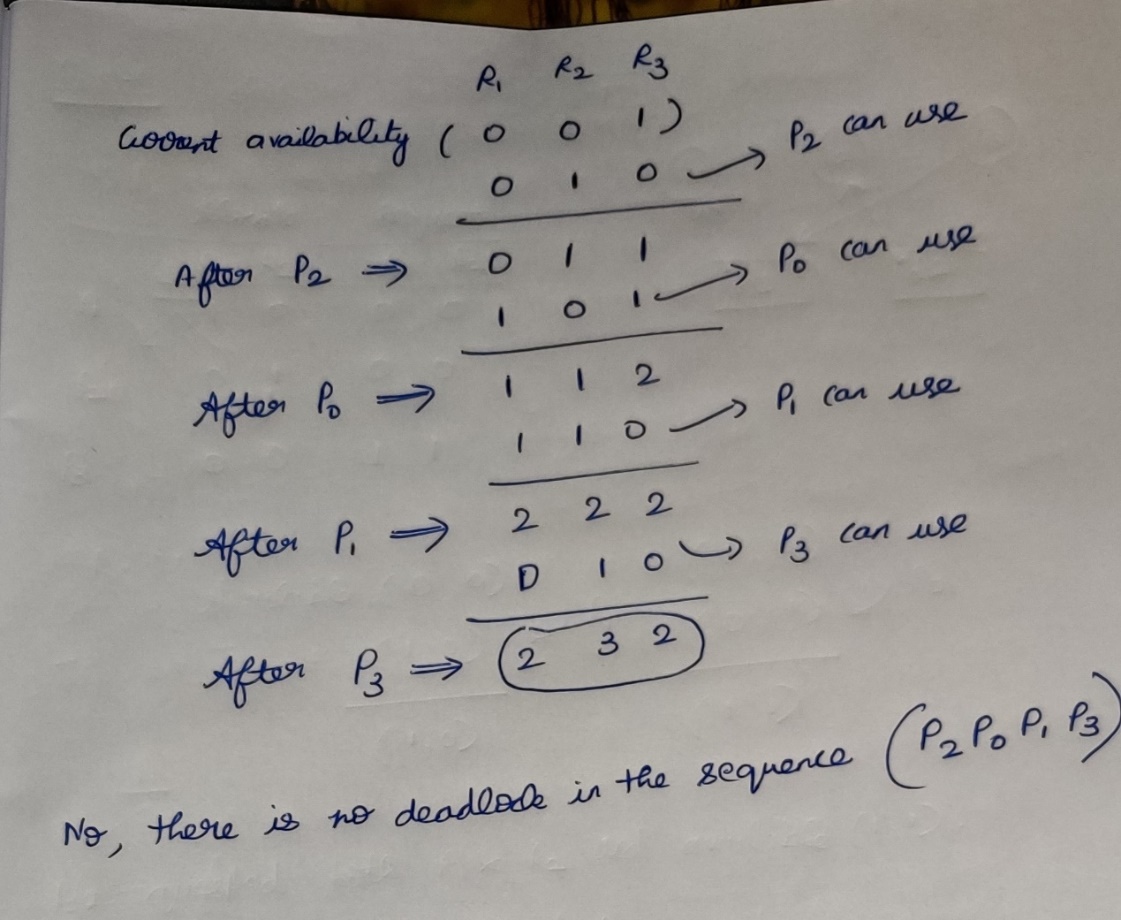
After P1 finishes its execution Available 🡪 (1,0)  
After P2 finishes its execution Available 🡪 (0,1)  
Then the total availability 🡪 (1,1) [ So P3 can execute easily]

**No Deadlock (but starvation)**

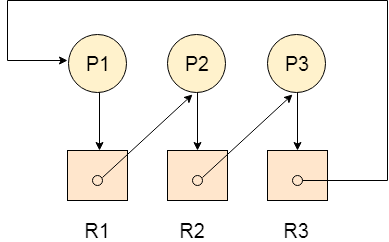


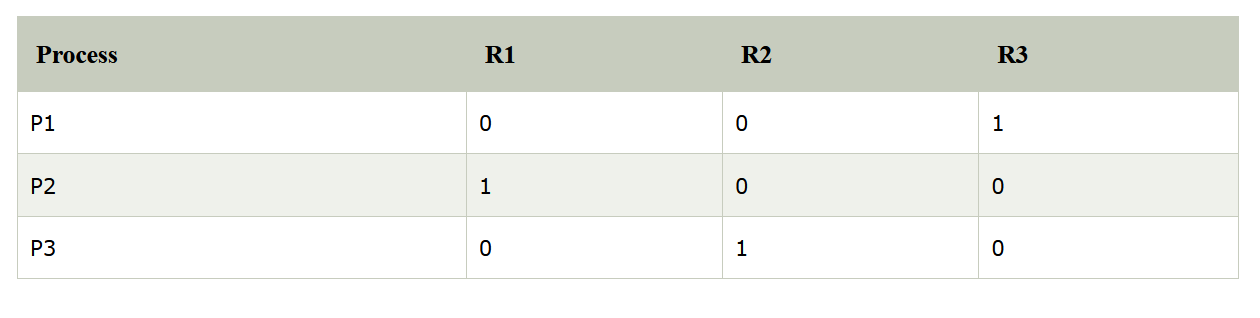


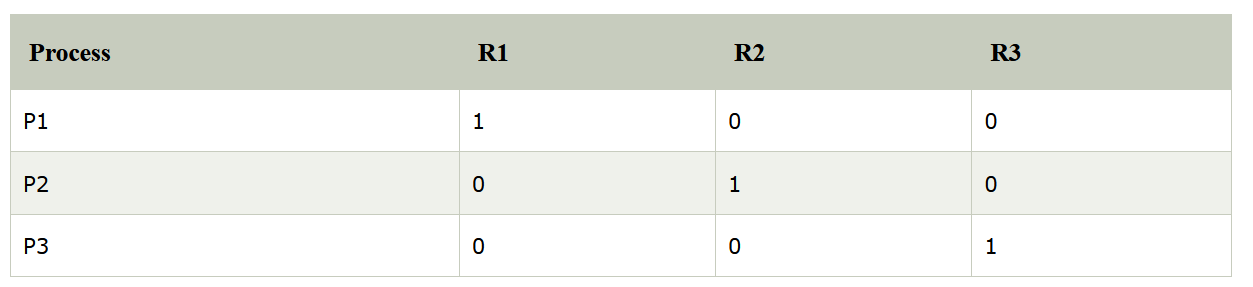


****

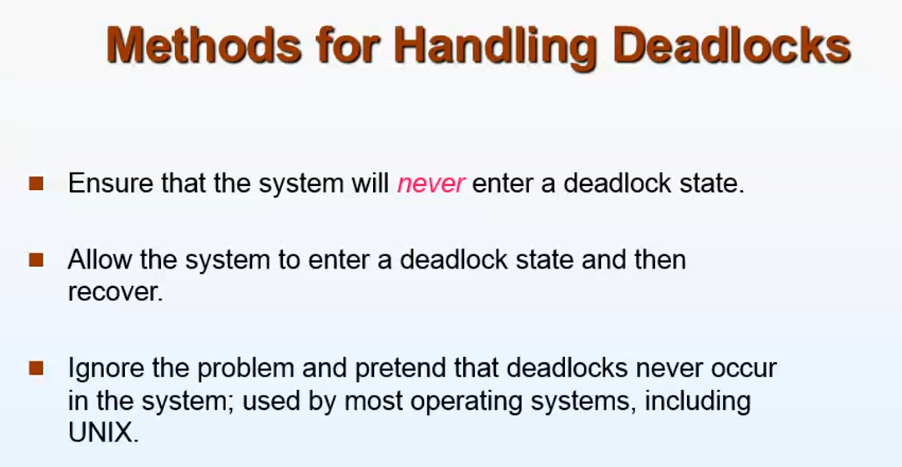
**4)**



**Allocation matrix**

**Request matrix**

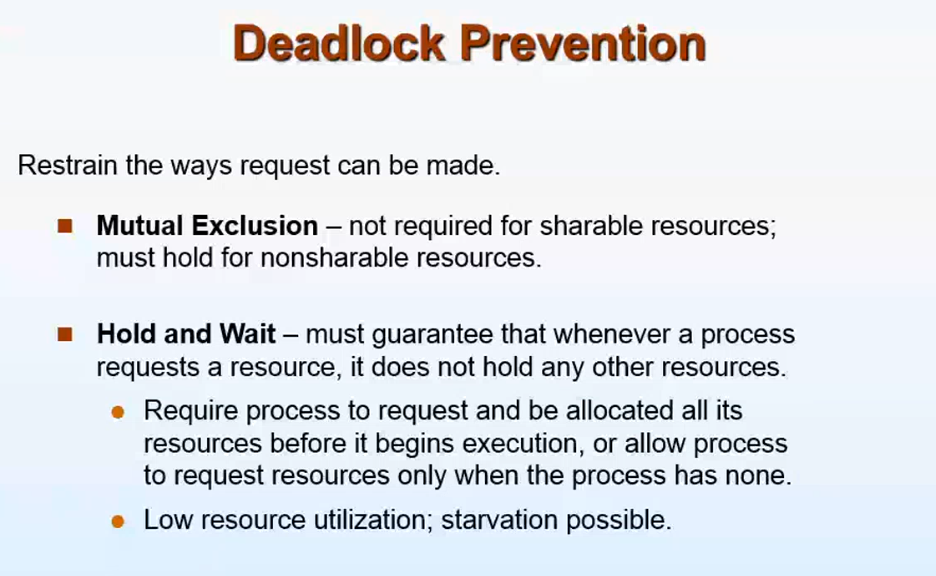
**Various method to handle deadlocks**



**Deadlock Avoidance**

1. Deadlock Ignorance (Ostrich method)
2. Deadlock Prevention
3. Deadlock Avoidance (Banker’s Algorithm)
4. Deadlock Detection and Recovery

**\* Allow the system to enter into deadlock and recover it  
\* Simply Switch OFF when the system freezes. So that all the process in RAM will be erased and start from the first.**

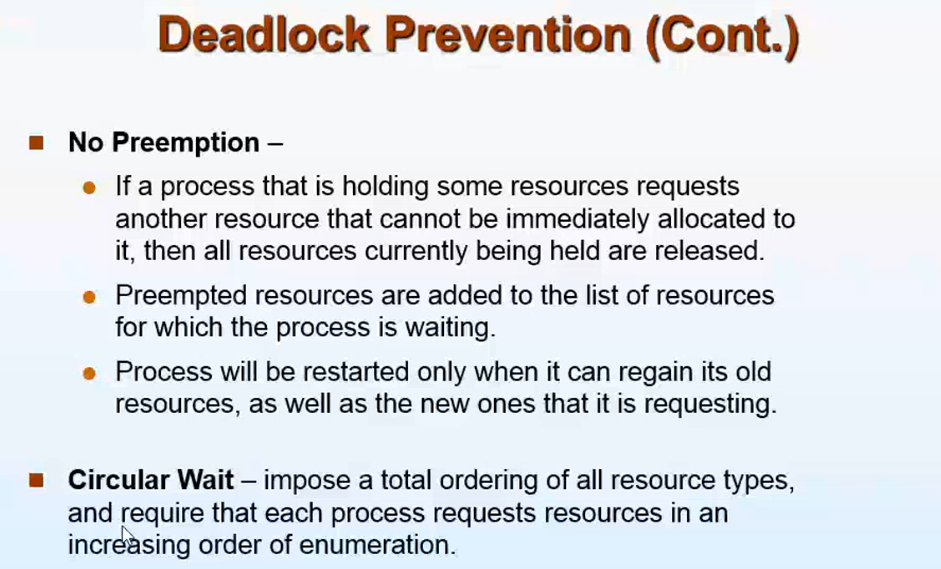


**Deadlock Prevention:**Reasons for deadlock   
\* Mutual Exclusion  
\* No pre-emption  
\* Hold and wait  
\* Circular wait   
If we try to make false of any of these 4 conditions, then deadlock can be avoided very easily.

**Hold and wait :**

P1) Me 🡪 holding (bat) waiting(ball)  
P2) Brother 🡪 holding (ball) waiting(bat)

If P1 leaves the bat P2 acquires the bat and go to play. After P1 leaves the bat, it has to request the bat once again to use it.   
But P1 don’t know when P2 will come back , so P2 starves for a long time.

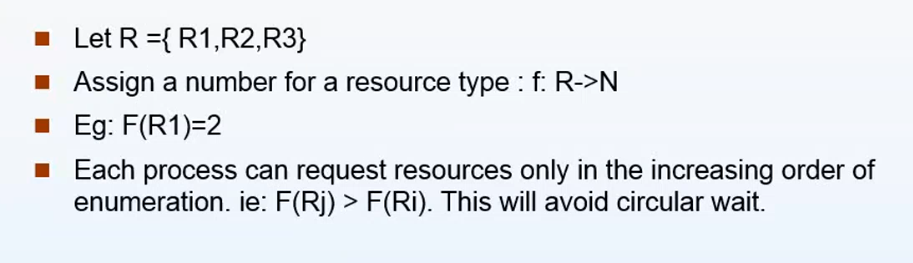


**No Pre-emption:**

P1 holding R1 . P2 requesting R1, so P2 forcibly remove R1. Once again P1 should request R1 to use it.  
This also leads to low utilization and starvation.   
(Higher priority process 🡪 More preference and   
lower priority process 🡪 Less preference)  
This approach is practical only when applied to resources whose state can be

easily saved and restored later, as is the case with a processor.

**Circular wait**



**Each process should request the resource in the increasing order of enumeration**

**A process which has acquired a larger number will never go ahead and acquire the process with the smallest number**

P1 R1 🡪 2  
 R2 🡪 3  
 R3 🡪 4



R1 🡪 2 (2>3)  
 R2 🡪 3 (3>4)  
P2 R3 🡪 4

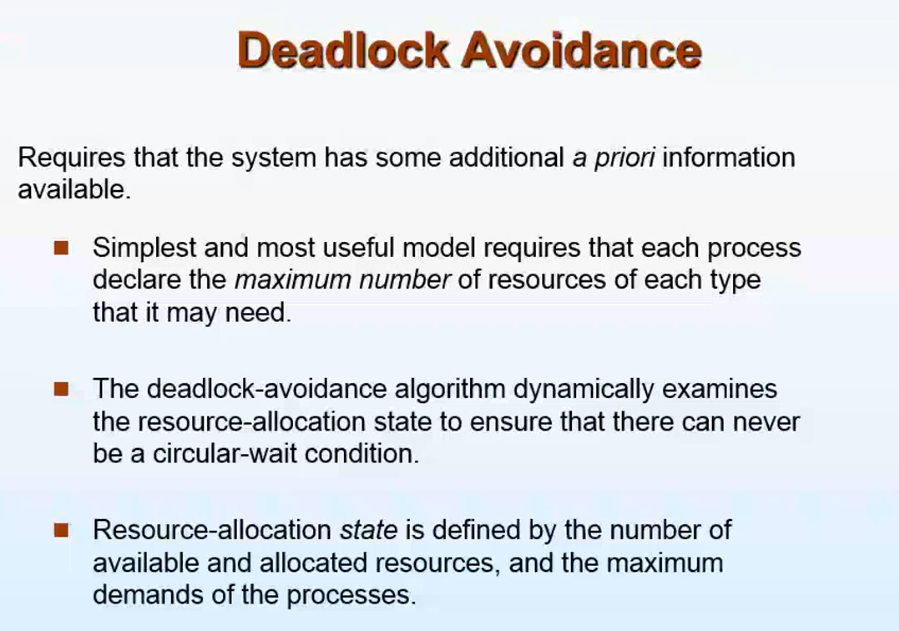


P2 should release R3 and then only it can use R1 and R2/ P2 can use R4.

P2 R3 🡪 4  
 R4 🡪 5



The Above conditions (Prevention of deadlock) are not at all possible.   
So another approach



A Small example

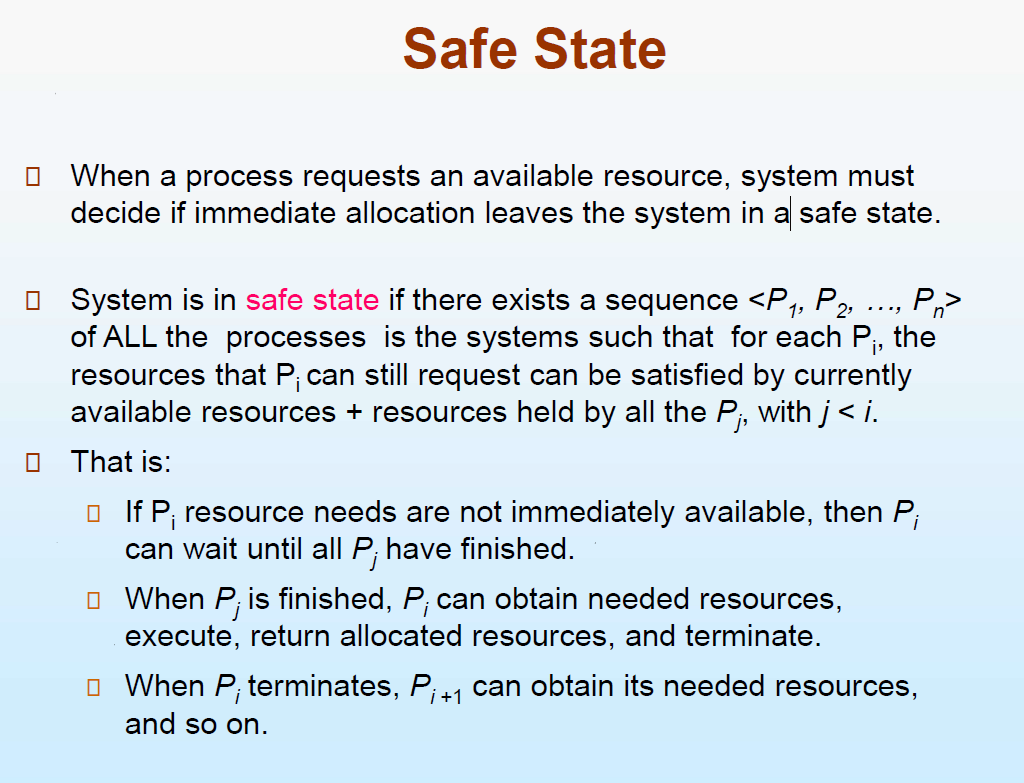
Father,mother,me,sister   
Available resources 🡪 A car and a bike   
**Someone wants to go out(No car, no bike) 🡪 Dead lock**

***Brother gone out with bike. If father gets a phone call from office he want to immediately go to office with car.***

* If I ask the car to go out, even if the car is available my father won’t give it to me unless my brother returns.
* Since if I take the car out and the phone call comes 🡪 Father can’t go out(dead lock)

Like this as soon as the process requests the resource OS immediately won’t give it, It checks after giving this resource whether it will end in deadlock/not. If not it will give.

In dead-lock avoidance, every process must declare the maximum number of resources it needs (from start to end)



Bankers Algorithm to be continued