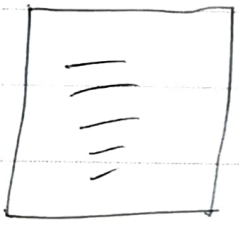
Reader - Writer Problem

For Writer

```
wait(wrt)
    write operation
signal(wrt)
```

initialize wrt = 1



Semaphore - wrt, mutex,
variable: readcount

For Reader

mutex = 1
readcount = 0

```
wait(mutex)
    readcount++
    if(readcount==1)
        wait(wrt)
signal(mutex)
    read operation.
wait(mutex)
    readcount--
    if(readcount==0)
        signal(wrt)
signal(mutex.
```

---

a = 10 Case 1.

① R(a)	P1 1→3
② a++	a = 11
③ W(a)	P2 1→2 = 12

Case: 2

P1 1: a = 10
P2 1: a = 10
P2 2: a = 10
P1 3: a = 11
P2 2: a = 11
P2 3: a = 11

Process synchronization

Race Condition:- When order of execution can change the result it is called race condition

Critical Section: In process execution, the area where process access a shared resources is called critical section.

Q.

```
P1()                    P2()
{                       {
① C = B - 1;            D = 2 * B    ③
② B = 2 * C;            B = D - 1    ④
}                       }
```

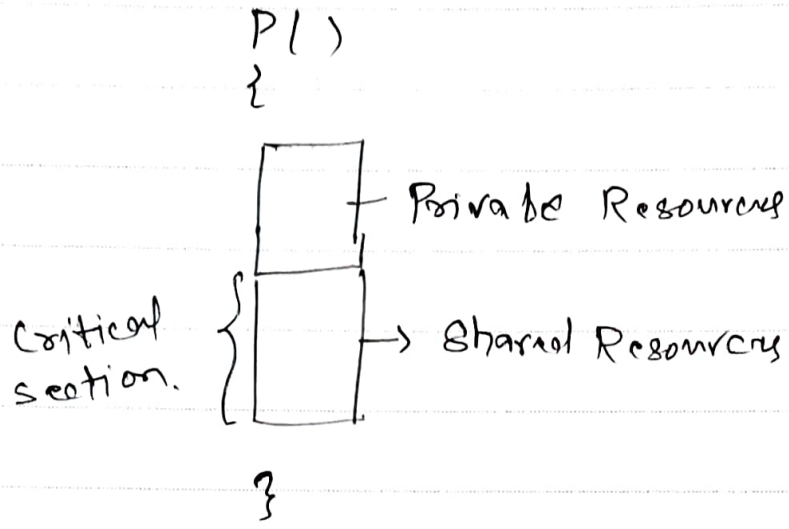
B is shared variable with initial value 2. How many different value possible after the execution of P1, P2

Ans

Case: 1	Case: 2	Case: 3	Case: 4	Case 5	Case: 6
1, 2, 3, 4.	3, 4, 1, 2	1, 3, 4, 2	3, 1, 2, 4	1, 3, 2, 4	3, 1, 4, 2
C = 2 - 1 = 1	D = 2 * 2 = 4	C = 2 - 1 = 1	D = 2 * 2 = 4	C = 2 - 1 = 1	D = 2 * 2 = 4
B = 2 * 1 = 2	B = 4 - 1 = 3	D = 2 * 2 = 4	C = 2 * 2 = 4	D = 2 * 2 = 4	C = 2 - 1 = 1
D = 2 * 2 = 4	C = 3 - 1 = 2	B = 4 - 1 = 3	C = 2 - 1 = 1	B = C * 1 = 2	B = 4 - 1 = 3
B = 4 - 1 = **3**	B = 2 * 2 = **4**	B = 2 * 1 = **2**	B = 2 * 1 = **2**	B = 4 - 1 = **3**	B = 2 * 1 = **2**
			B = 4 - 1 = **3**		

Critical section Problem :-

* The shared resources should be used in mutual exclusion fashion by processes otherwise it leads to inconsistency.



Criteria for critical section problem solution :-

(1) Mutual Exclusion (Mandatory) : If process P_i is executing in its critical section, then no other process can be executing in their critical sections

(2) Progress :- (Mandatory) If no process is executing in its critical in its critical section and some processes wish to enter their critical sections, then only those processes that are not executing in their remainder section can participate in the decision on, which will enter its critical section next.

and this selection cannot be postponed indefinitely.

3. Bounded Waiting (optional) :- There exist a bound on the number of times that other processes are allowed to enter their critical sections after a process has made a request to enter its critical section and before that request is granted.

```
do {
        entry section
            Critical Section
        exit section
            remainder Section
} while(true);
```

← General structure of a typical process P_i.

Two Process Solution : using boolean variable turn = 0/[1]

P_0

```
While(1){
    while( turn != 0 );
     critical section
     turn = 1 ;
     remainder section
     }
```

P_1

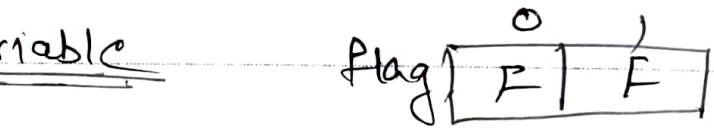
```
while(1) {
  while( turn != 1 ) ;
    critical section
    turn = 0
   remainder section
  }.
```

→ This soln satisfy mutual exclusion criteria

→ But it is not satisfy the progress (Because, after exit from critical section by P_0, P_0 cannot go in critical section again. For that P_1 has to execute its critical section then only P_0 can go in its critical section, so there is a loop formation $P_0 \rightarrow P_1 \rightarrow P_0 \rightarrow P_1$ ------)

So this is not satisfying the progress criterial, it forcing strict alternations.

Using Flag variable



P_0	P_1
while(1){	while(1){
flag[0] = T	flag[1] = T
while(flag[1]);	while(flag[0]);
critical section	critical section
flag[0] = F	flag[1] = F
}	}

→ Its satisfy mutual exclusion criteria

→ Its not satisfy progress criteria always but in some scenario it can't which is explain below:—

→ Problem with this soln is occur when suppose P_0 done the context switch after the execution of flag[0] = T and P_1 start to execute then both flag value become true and neither process can enter into critical section. (~~Not progress~~) So progress criteria not satisfied.

Peterson Solution :-

P_0

```
while(1)
{
   flag[0] = T
   turn = 1
   while(turn==1 && flag[1]==T);
     critical section
   flag[0] = F
}
```

P_1

```
while(1)
{
   flag[1] = T
   turn = 0
   while(turn==0 && flag[0]==T);
     critical section
   flag[1] = F
}
```

→ It satisfy mutual exclusion.

→ It satisfy progress.

→ Bounded wait

→ It good for two process solution.

→ Bit difficult to write for n process.

SEMAPHORES

A semaphore is an integer variable that apart from initialization, is accessed only through two standard atomic operations. wait [P(), down()], signal [V(), UP()].

~~Usages of Semaphores~~:

int S = 1 [S initialized with 1 for C.S prob. soln]

P_i:

```
do {
    entry section
    // Critical section
    exit section
    // remainder Section
} while (True);
```

(i) wait()

```
wait (S)
{
} while (S<=0);
S = S-1
}
```

(ii) signal()

```
signal (S)
{
S = S+1
}
```

Usages:-

(i) Used for providing the soln of n-process critical section problem.

(ii) Can be used to decide the order of execution among the processes.

(iii) Can be used for resource management.

(i) Critical section problem soln using semaphore.

P_i:

```
do {
    wait(s)
    // critical section
    signal(s)
    // remainder section.
} while(True);
```

* Here semaphore satisfy mutual exclusion, progress for critical section problem soln criteria but it not satisfying Bounded wait creteria.

(ii) Deciding the Order of execution using Semaphore:

$S1 = 0, S2 = 0$

wait(S1)		wait(S2)
P1	P2	P3
signal(S2)	signal(S1)	

Suppose we want the execution in following order

P2 → P1 → P3

(Reading Maping) (Area calculation) (Cost of the area)

→ Take two semaphore variable $S1 = 0, S2 = 0$

(222) Resource Managment using Semaphore:

If you have more than one replicas of some resource, you need to do only one change, initialize the semaphore variable to no. of resources and keep the other thing same. For eg: if there is 5 printer available then five process can use that and so we can initialize S=5.

Q: Suppose we ~~have~~ want to synchronize two processes P and Q, using binary semaphore S,T

Process P	Process Q
While(1)	While(1)
{	{
W	Y
print '0'	print '1'
print '0'	print '1'
X	Z
}	}

We want the o/p string should be like '001100110011' then what should be value of W, X, Y, Z?

	W	X	Y	Z	
a	P(S)	V(S)	P(T)	V(T)	S,T = 1
b ✓	P(S)	V(T)	P(T)	V(S)	S=1, T=0
c	P(S)	V(T)	P(T)	V(S)	S,T = 1
d	P(S)	V(S)	P(T)	V(T)	S=1, T=0

(22) if o/p string should not be like

$0 1^n 0$ or $1 0^n 1$, n = odd

	W	X	Y	Z	
a	P(S)	V(S)	P(T)	V(T)	S,T = 1
b	P(S)	V(T)	P(T)	V(S)	S,T = 1
c ✓	P(S)	V(S)	P(S)	V(S)	S = 1
d	P(S)	V(T)	P(S)	V(T)	S,T = 1

Bounded Buffer Producer - Consumer Problem

Problem Statement:-

→ Producer must check empty space first, before producing (Overflow).

→ Consumer must check ~~availability~~ existency of data first, before consuming (Underflow).

→ While producer producing, consumer should not consume.

→ While consumer consuming, producer should not produce.

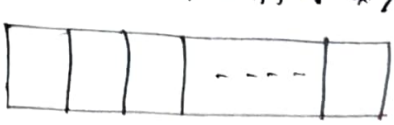
Soln:-

Semaphor S = 1

" E = n

" F = 0

/* n is size of butter */



```
Void producer()
{
 While(True)
  {
   produce()
   wait(E)
   wait(s)
   append()
   signal(S)
   signal(F)
  }
}
```

```
void consumer()
{
 while(true)
  {
   wait(F)
   wait(S)
   take()
   signal(S)
   signal(E)
   use()
  }
}
```

Reader-Writer Problem

Problem statement:

There is some piece of text. On this we can perform two operations read and write.

→ On text, more than two reader can enter.

→ While writer writing, another writer or reader cannot enter.

i.e Read-write or write-write operation can create a clash.

Soln :- → Take two semaphore wrt, mutex and one variable readcount.

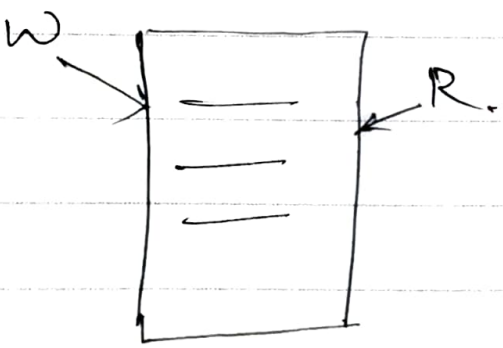
→ Initialize wrt = 1, mutex = 1 and readcount = 0

For writer

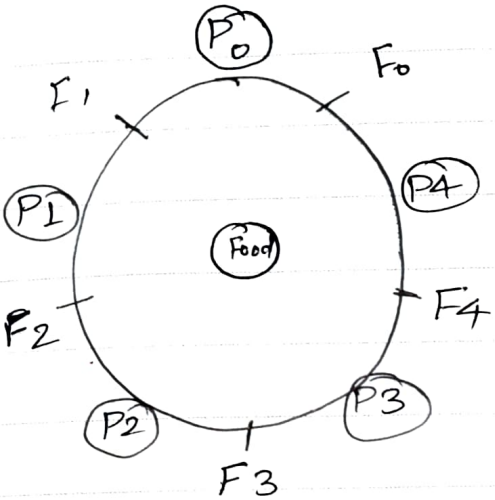
```
wait(wrt)
        write operation
signal(wrt)
```

For Reader

```
wait(mutex)
    readcount ++
    if(readcount == 1)        ] Performed by first reader
        wait(wrt)             ]
signal(mutex)
    read operation.
wait(mutex)
    ~~read operation~~
    readcount --
    if(readcount == 0)
        signal(wrt)
signal(mutex)
```



DINING PHILOSOPHER PROBLEM



→ We are having rounded dining table.

→ On the dining table some philosopher sitting for dinner.

→ There are some fork/chopstick for, by using philosopher can take food.

→ Every philosopher have two state: thinking and eating.

→ Philosopher can take left fork first and then right fork. after that can start to eat.

```
void philosophe( )
{
  while (true)
    {
    Thinking( )
    Wait( chopstick (i))
    Wait ( chopstick ((i+1)%.N)
     EAT( )
    signal( chopstick (i))
    signal( (chopstick.((i+1)%.N))
    }
}
```

Semaphore chopstick[5] (No. of philosophers)

S_0 S_1 S_2 S_3 S_4
1 1 1 1 1

P_0	S_0	S_1
P_1	S_1	S_2
P_2	S_2	S_3
P_3	S_3	S_4
P_4	~~S_4~~ S_0	~~S_0~~ S_4

(For avoiding deadlock)

Hardware Synchronization

→ This is used in multiprocessor environment.

→ Many machines therefore provide special hardware instructions that allow us either to test and modify the content of a word, or to swap the contents of two words, automically- that is, as one uninterruptible unit.

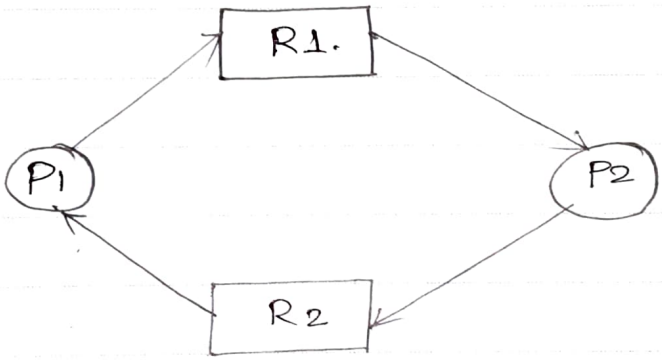
```
boolean testAndSet(boolean * target)
  {
      boolean rv = *target
      *target = True ;
      return rv;
      }

do {
    while (testAndSet(lock));
        // critical section
    lock = false ;
        // remainder section
} while(true);
```

DEADLOCK

→ In a multiprogramming system, a number of process compete for limited number of resources and if a resource is not available at that instance then process enters into waiting state.

→ If a process unable to change its waiting state indefinitely because the resources requested by it are held by another waiting process then system is said to be in deadlock.



System Model:-

→ Every process will request for the resources.

→ If entertained then process will use the resources.

→ Process must release the resource after use.

Necessary Condition of Deadlock :-

(i) **Mutual Exclusion** :- At least one resource type in the system which can be used in non-sharable mode i.e mutual exclusion (one at a time / one by one). eg. Printer.

(ii) **Hold and Wait** → A process is currently holding at least one resource and requesting additional resource which are being held by othe process.

(iii) **No-preemption** → A resource can not be preempted from a process by any other process, resource can be released only voluntarily by the process holding it.

(iv) **Circular Wait** :→ Each process must be waiting for a resource which is being held by another process, which in turn is waiting for the first process to release the resource.

S = Q = 1.

```
PROCESS(P0)          PROCESS(P1)
wait(S);             wait(Q);
wait(Q);             wait(S);
  :                    :
signal(S)            signal(Q);
signal(Q);           signal(S);
```

Deadlock Handling Methods:

(i) **Prevention** :- It means design such a system which voilate at least one of four necessary conditions of deadlock and ensure independence from dead lock

(ii a) **Avoidance** :- System maintains a set of data using which it takes a decision weather to entertain a new request or not, to be in safe state.

(ii b) **Detection and Recovery** :- Here we wait until deadlock occurs and once we detect it, we recover from it.

(iv) **Ignorance / Ostrich algo** :→ We ignore the problem as if it does not exist.

Violation of Mutual exclusion condition under deadlock Prevention Approach

We cannot violate mutual exclusion because a resource can be utilized by how many process at a time, depend on the resource hardware property.

Violation of Hold and Wait Under deadlock Prevention

* Conservative approach :- Process is allowed to start execution if only if it has acquired all the resources (less efficient, not implementable).

* Do not hold :- Process will acquire only desired resources but before making any fresh request it must release all the resources that it currently hold (efficient, implementable)

* Wait timeout :- We place a maximum time upto which a process can wait, after which process must release all the holding resources and exit.

Violation of No-Preemption under Deadlock Prevention Approach

* Force-full Preemption :- We allow a process to forcefully preempt the resource holding by othe process.
  - → This method may be used by high priority process or system process.
  - → The process which are in waiting state must be selected as a victime instead of process in the running state.

Violation of Circular Wait Under Deadlock Prevention Approach

- → Circular wait can be eliminated by first giving a natural number of every resource.
  $f: N \rightarrow R$
- → allow every process to either only in the increasing or decreasing order of the resource number
- → If a process require a lesser number (in case of increasing order) then it must first release the resources larger than required number.

Deadlock Avoidance.

Safe State:- A state is safe if the system can allocate resources to each process (upto its maximum) in some order and still avoid a deadlock. A system is in a safe state only if there exist a safe sequence.

→ A safe state is not a deadlock state.

→ A deadlock state is an Unsafe state.

→ Not all unsafe states are deadlock, but an unsafe state may lead to a deadlock.

(1) BANKER'S Algorithm :-

Let n be the number of processes in the system and m be the number of resource type

* **Available:-** A vector of length m indicates the number of available resources of each type. If available[j] = k, there are k instance of resources type R_j available.

* Max: An nxm matrix defines the maximum demand of each process. If Max[i,j] = K, then process Pi may request at most K instance of resource type Rj

* Allocation: An nxm matrix defines the number of resources of each type currently allocated to each process. If allocation[i,j] = K, then process Pi is currently allocated K instances of resource type Rj

* Need: An nxm matrix indicates the remaining resources need of each process. If need[i,j] = K, then process Pi may need K more instance of resource type Rj to complete its task.

$$Need[i,j] = Max[i,j] - Allocation[i,j]$$

Safety algorithm :-

The algorithm for finding out whether or not a system is in a safe state can be described as follows:-

(a) Let Work and Finish be vectors of length m and n, respectively.

Initialize Work := available and Finish[i] = false for i = 1, 2, ---, n.

(b) Find an i such that both
(i) Finish[i] = false
(ii) Needi ≤ Work
If no such i exist, go to step (d)

(c) Work = Work + Allocationi
Finish[i] = true
goto step (b)

(d) If Finish[i] = true for all i then system is in a safe state.

Complexity :- $m \times n^2$

Resource-Request Algorithm

Let Requesti be the request vector for process Pi. If Requesti[j] = k then process Pi wants k instances of resource type Rj. When a request for resources is made by process Pi, the

following actions are taken:

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

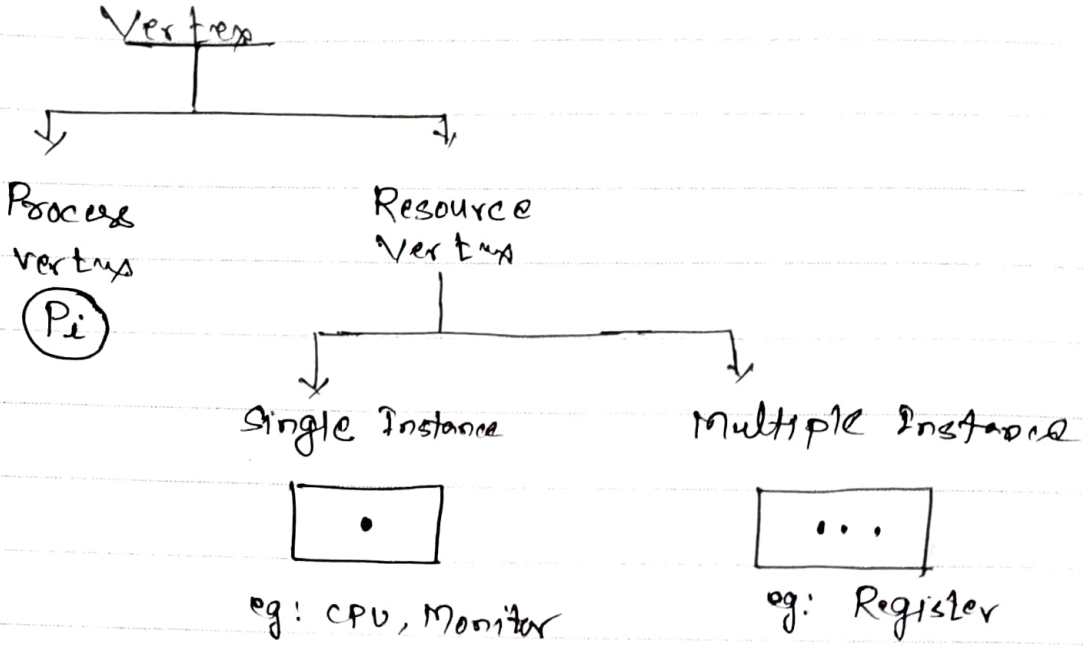
[2]. If $Request_i \leq Available$ go to step 3. Otherwise P_i must wait, since the resources are not available.

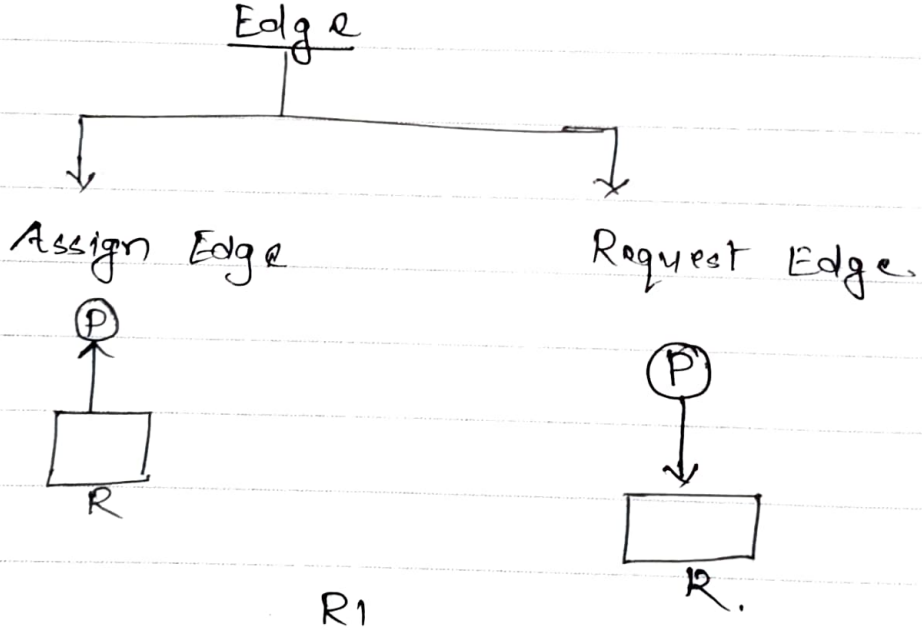
[3]. Have the system pretended to have allocated the request resources to process P_i by modifying the state as follow:

$$Available = Available - Request$$
$$Allocation_i = Allocation_i + Request_i$$
$$Need_i = Need_i - Request_i$$

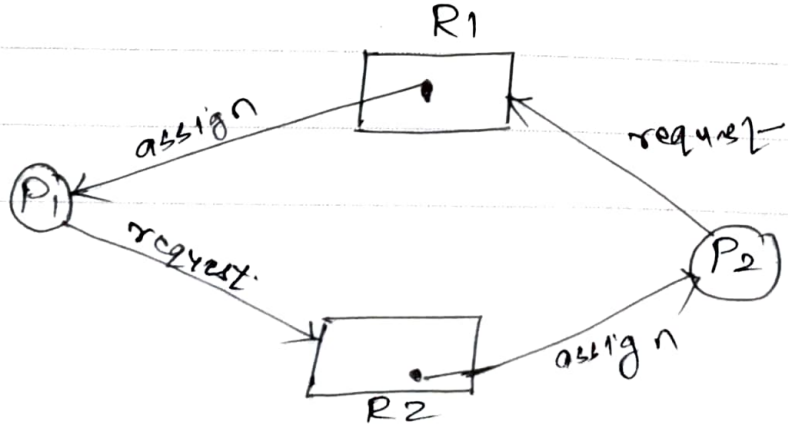
If the resulting resource allocation state is safe, the transaction is completed and process P_i is allocated its resources. However, if the new state is unsafe, then P_i must wait for $Request_i$ and the old resources-allocation state is stored.

(2) Resource Allocation Graph (RAG)





eg.

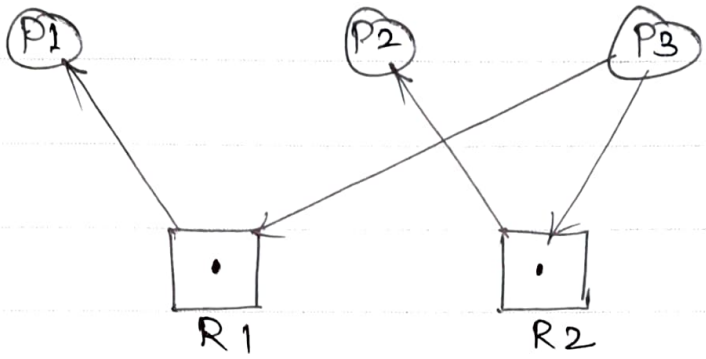


	Allocate R1	Allocate R2	Request R1	Request R2
P1	1	0	0	1
P2	0	1	1	0

Availability : (R1, R2) = (0, 0)

So Deadlock.

eg: 2



	Allocate R1	Allocate R2	Request R1	Request R2
P1	1	0	0	0
P2	0	1	0	0
P3	0	0	1	1

Available (R1, R2) = (0, 0)

1 0

1 1

No deadlock.