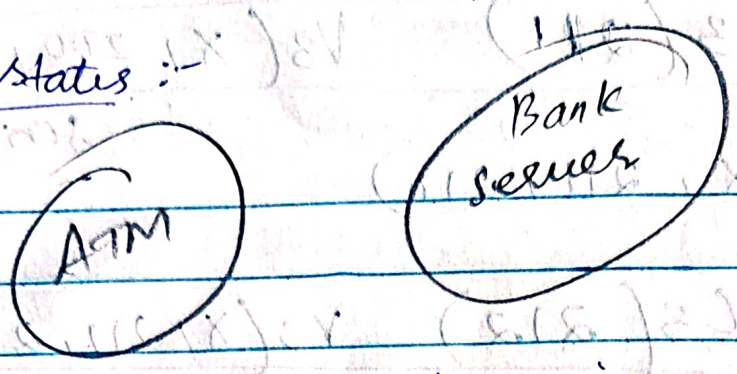
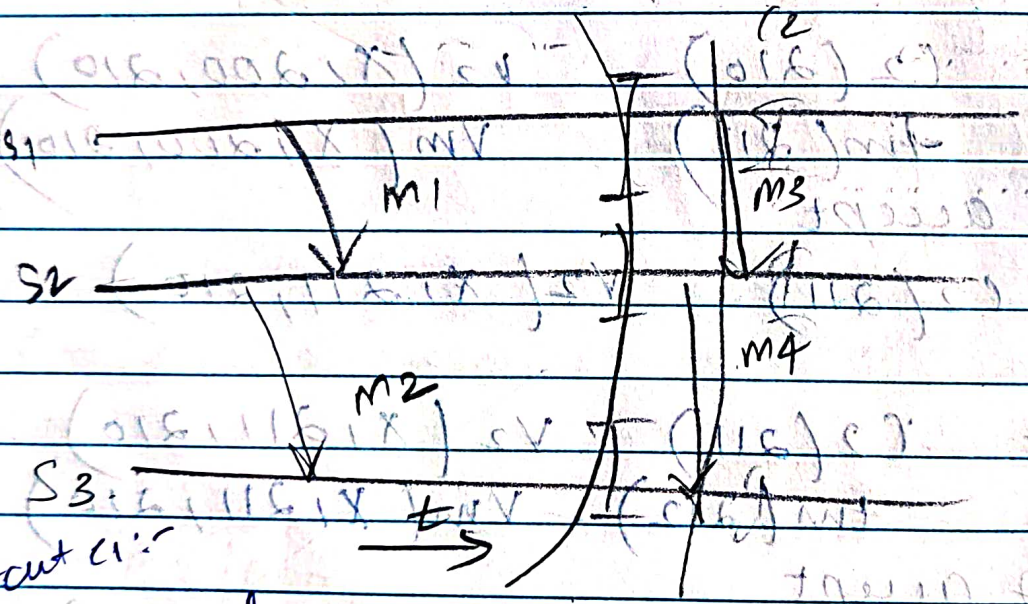
Global states :-



While withdrawing, bank is recording the state

Union of all local states = global state
has send & receive events.

$$GS = \bigcup_{i=1}^{k} LS_i$$



cut C1 :-

$LS_1 = \{ send(m_1) \}$

$LS_2 = \{ send(m_2), recv(m_3) \}$

$LS_3 = \{ recv(m_2) \}$

If every send has its receive → Strongly consistent GS.

~~Consistent GS~~ =

Cut : graphical line drawn in state.

For cut C2 :

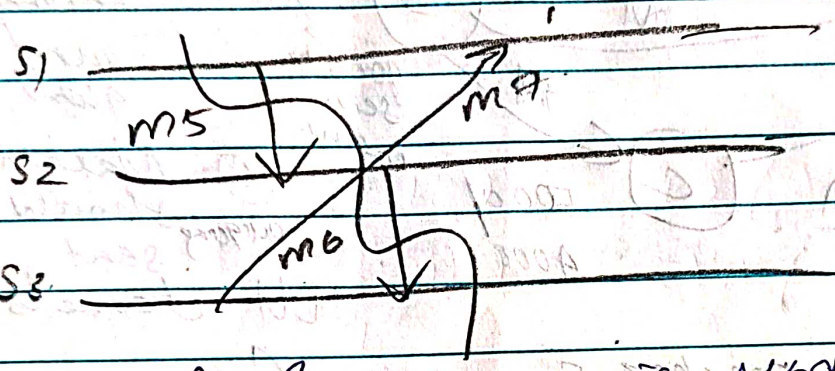
$LS_1 = \{ send(m_3) \}$.
$LS_2 = \{ send(m_4) \}$.
$LS_3 = \{ rec(m_4) \}$

] no strongly consistent

Every Receive should have corresponding send → CONSISTENT GS.

It can have transits.

↳ only the send() messages is recorded, but not their rec().



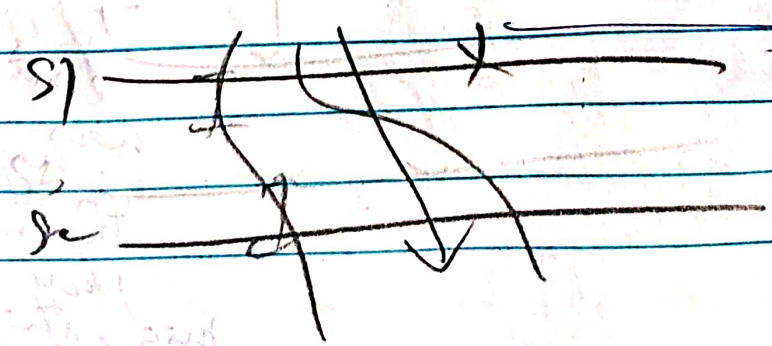
$LS_1 = \{ \}$ ⇒ S_1 is sending msg to everyone but none is receiving it

$LS_2 = \{ send(m_5) \}$
$LS_3 = \{ rec(m_6) \}$

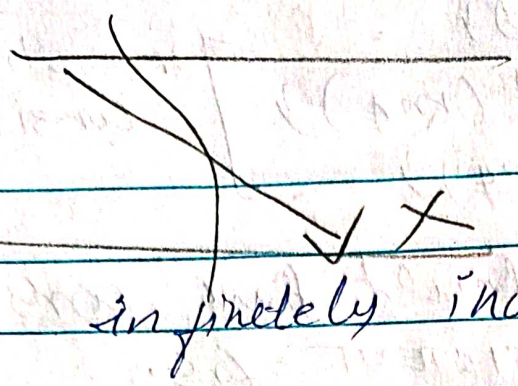
} INCONSISTENT GS

no send() for receive message()
[Receive has been recorded before send it recorded]

$LS_3 = \{ send(m_6), rec(m_7) \}$.

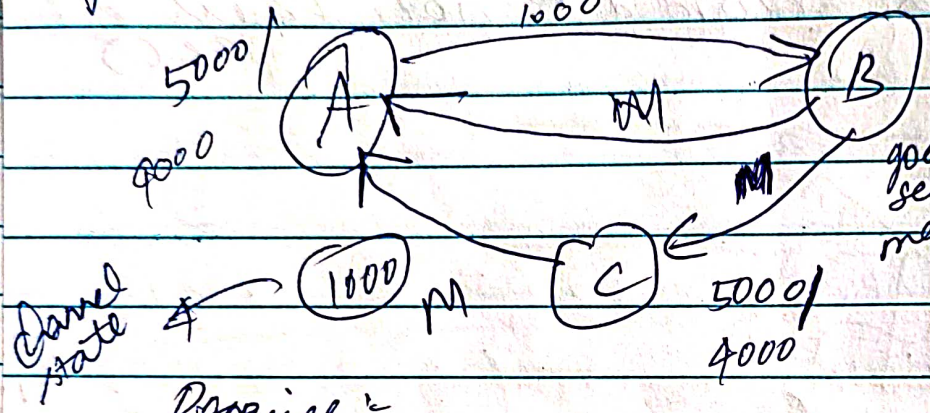


If there is a crash, no evidence of send is there.



send is sent, but there is a crash at receiver, So the msg is indefinitely in transit.

Global snapshot recording Algorithm :-



Senders:

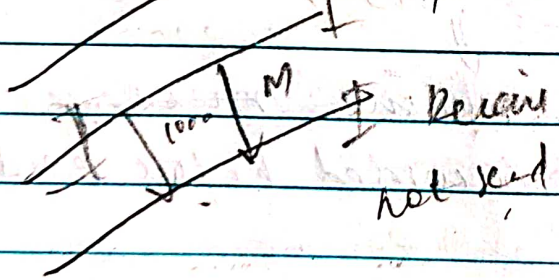
① Record its own state & global

② Marker msgs should be sent through all outgoing channels

Receiver:

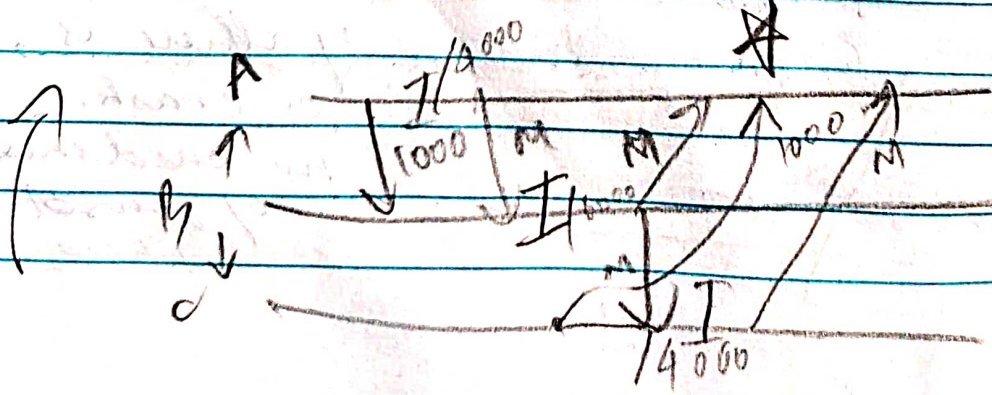
Snapshot marker & pass the marker to other outgoing channels (whenever as sender)

C stops when initiator gets the marker msg from all incoming channels



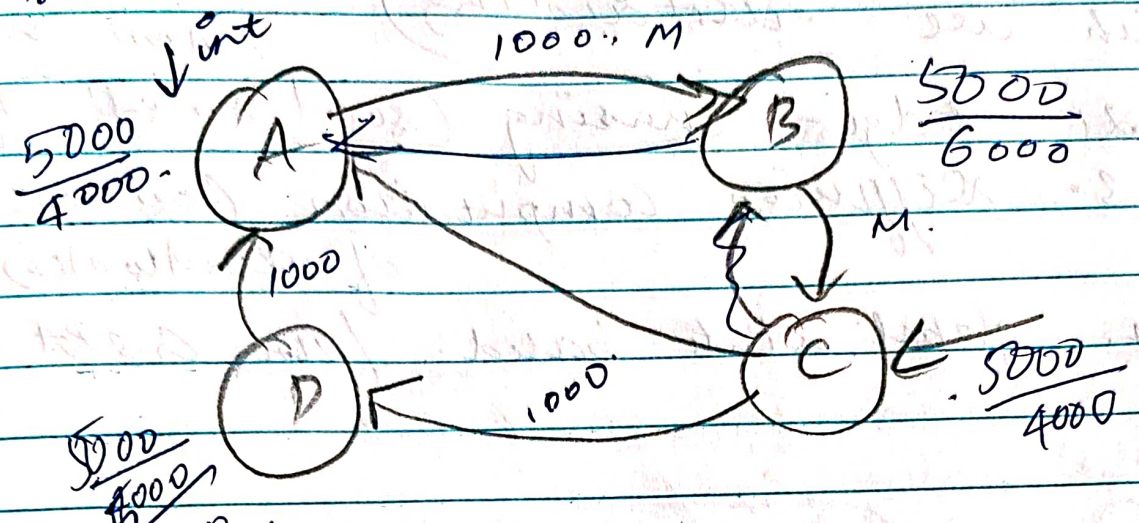
Record of B to C is snapshotted but not sent (∵ Inconsistent) Thus schedule the msg

Channel state :-



If any msgs are in any incoming channel then the msg (other) Snapshot

If snapshot is already taken ... the msg in channel state is considered as 'in transit.'



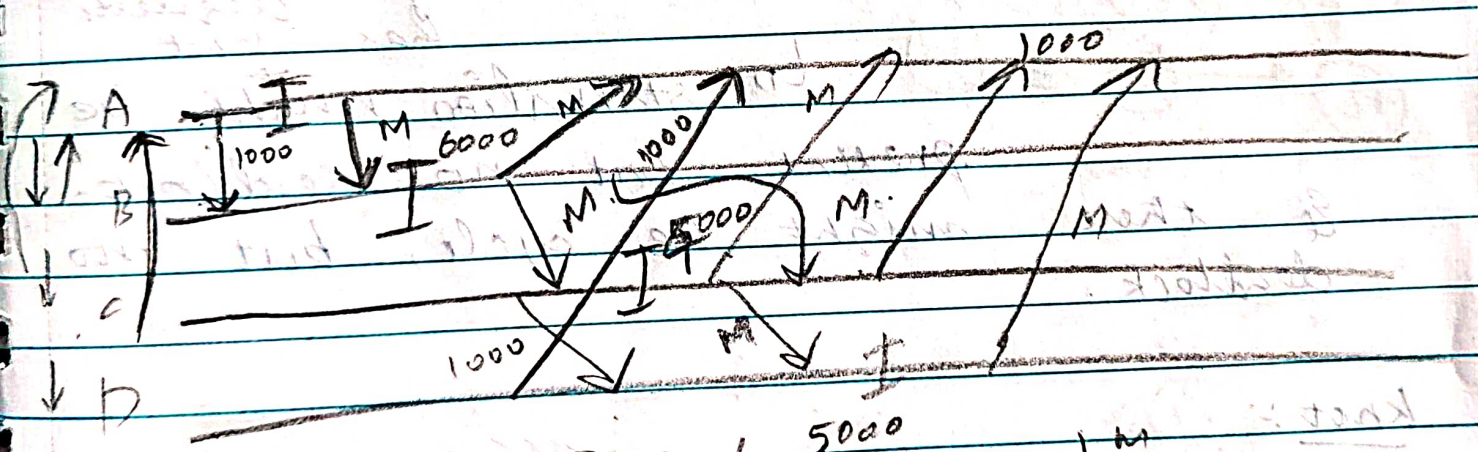
t1 :- A → B.

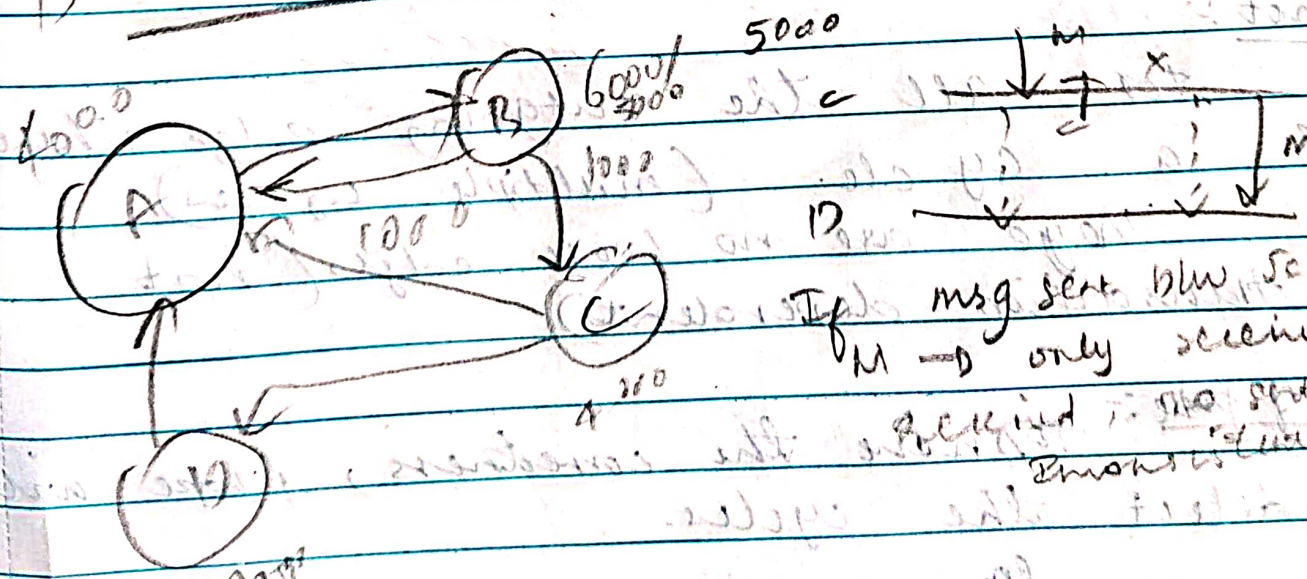
t2 :- init.

t3 :- D → A

t4 :- C → A, B → C / C → D, M from C

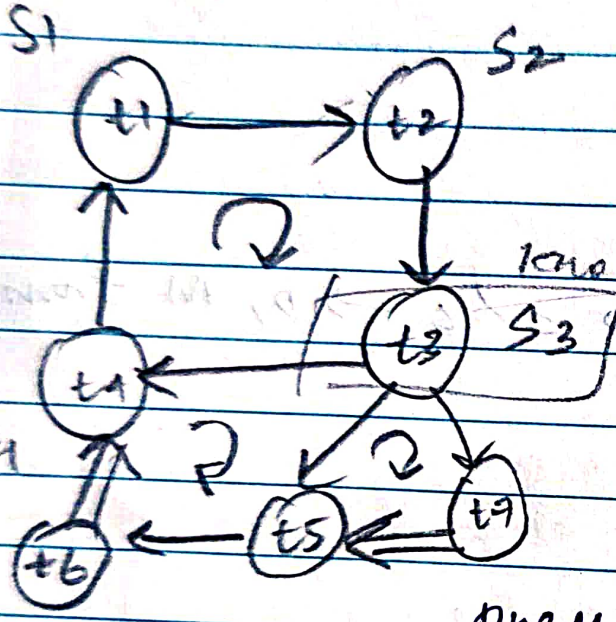
t5 :- C → A.





Classification of deadlock (Chandy's) :-

1. path pushing (push all the edges which are outstanding)
2. Edge-chasing (send small probe msgs) CAND models
3. Diffusion computation (OR kind of deadlocks)
4. global state based. (use Chandy algo)



OR Request Models

A system can continue if it gets only one resource no matter how many replies it requests has sent.

↳ starvation might be present, but no deadlock.

& there might be cycle, but no deadlock.

Knot :-

For all the outgoing edge there is a cycle. (multiple cycles)

There are no loose edges (not transaction dependent)

Use of probe :- To prove the correctness, probe will detect the cycles.

Co.

Initiator sends a query ~~probe~~ (i, x, k)
Query is sent only when there is an outgoing edge.

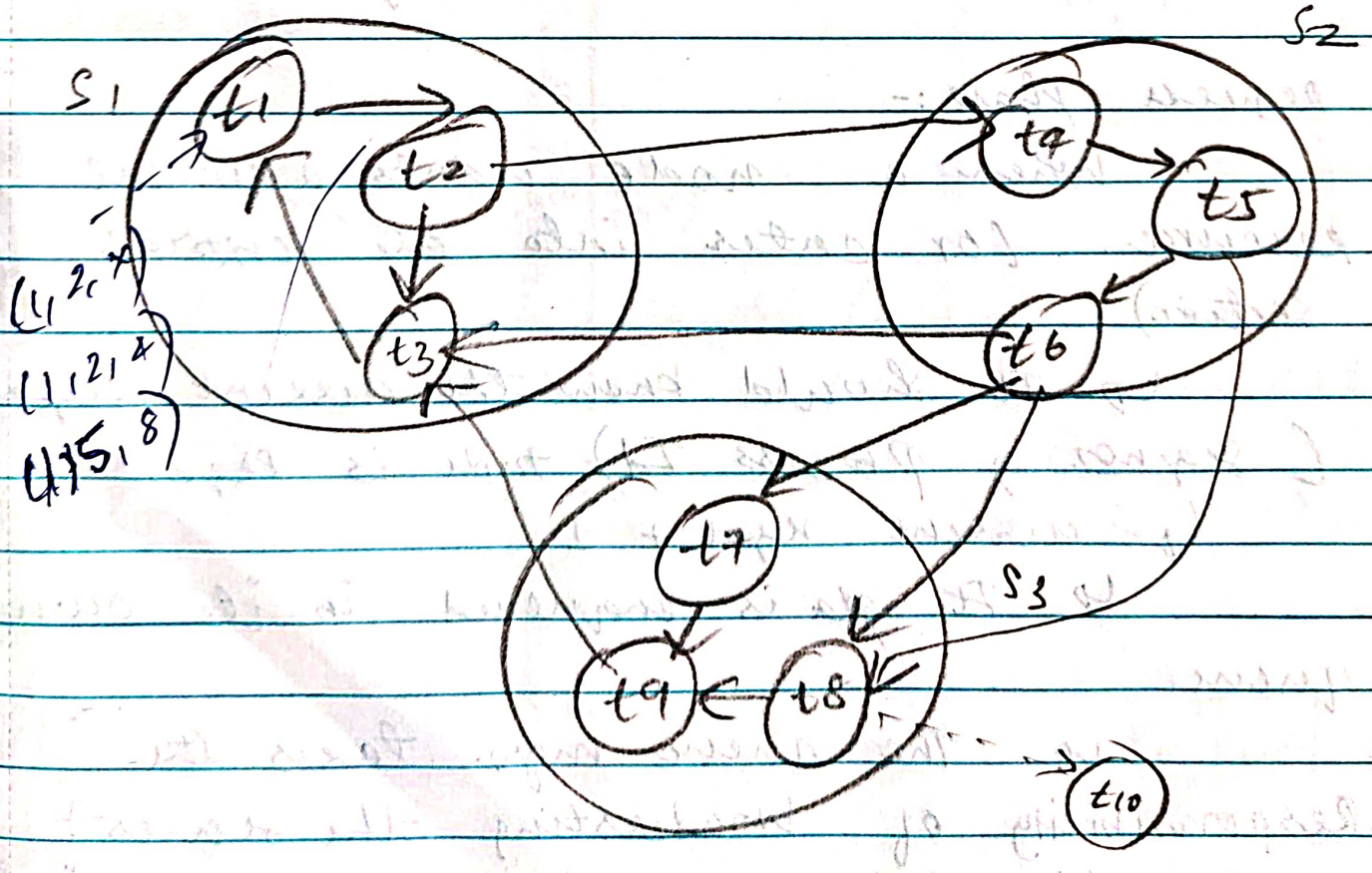
~~If the Re~~

After Receiving the receives updates & broadcasts the probe. to all its outgoing edge.

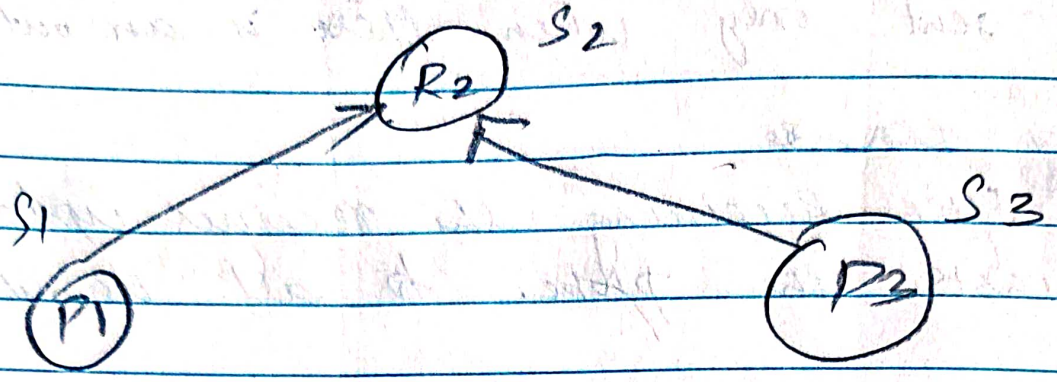
All the nodes, after Receiving has to send back a Reply to the sender after it Receives a reply when it sends to another Node.

↳ only when a transaction is waiting, it Receives a Reply

↳ only when it receives Reply for all queries, it Replies.



DISTRIBUTED MUTUAL EXCLUSION:



① Non token Based (Queue Distributed)
② token Based: (forcefully maintain consistency)

Lamport Ricart

Lamport's Algo:

(Resource can be local/ distributed, but it should follow the order in which they are requested)

(not token)

Has 3 phases
↳ Request, Reply, Release

Request phase:-

When a node wants a critical resource, (or enter into the critical section)

↳ It should know the current Req no. (Seq no., Process Id) triple is Reqd.
↳ current reqst + 1
↳ It pla is enqueued in its own queue
↳ The queue mgr. takes the Responsibility of broadcasting the request to all. (consistency is forcefully maintained by this)

Reply :- Everyone Responds to the Request in order in which they get Request

Condition :

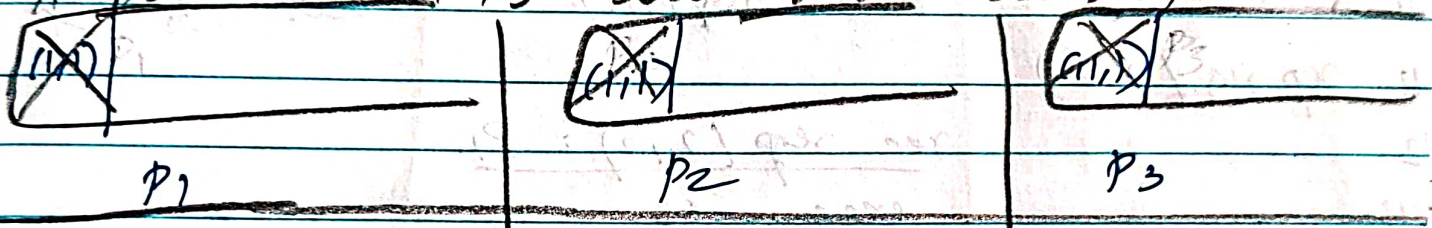
If 2 process are replying to same resource

Release :-

After use, release the resource, so that others can use.

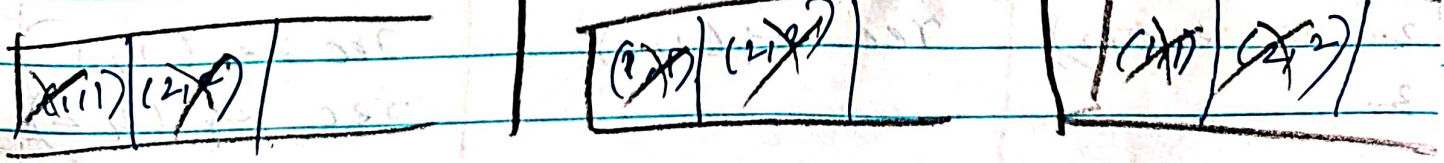
Reliable channel is Req.

// both P2 & P3 are not accessing resource



P_1	P_2	P_3
1. Req (1,1)		
2.	Rec req (1,1)	Rec req (1,1)
3.	Rep req (1,1)	Rep req (1,1)
4. rec reply (1,1) : P_2		
5. rec reply (1,1) : P_3		
6. Exec C.S		
7. rel req (1,1) (delete its own Req)		
8.	rec rel (1,1)	rec rel (1,1)
	del req (1,1)	del req (1,1)

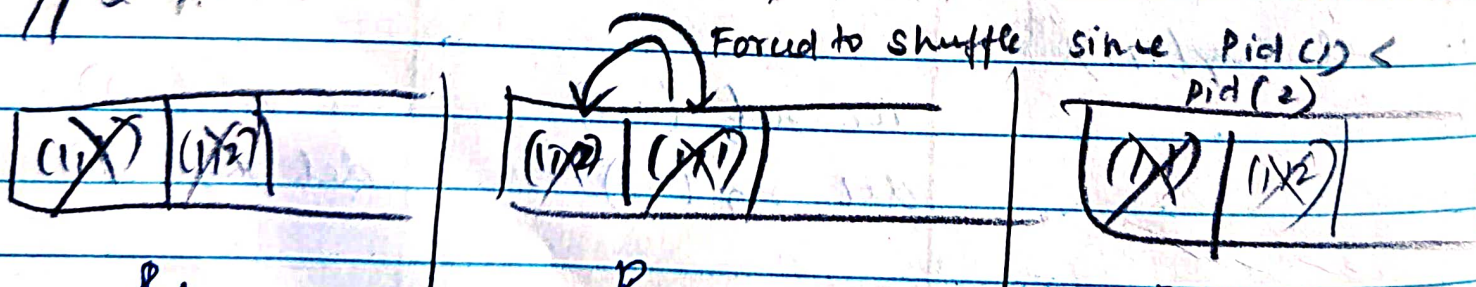
// P_2 is alone interested in accessing the Resource



1. req (1,1)		
2.	rec req (1,1)	rec req (1,1)
3.	req (2,2)	rep req (1,1)
4. ~~rec rep (1,1)~~ rec rep (1,1) : P3	rep req (1,1)	rec req (2,2)
5. rec rep (1,1) : P2		rep req (2,2)
6. rec req (2,2)	rec rep (2,2) : P3	
7. exec CS		
8. Rel req (1,1)		
9.	rec rel (1,1)	rec rel (1,1)
10.	del req (1,1)	del req (1,1)
11. rep req (2,2)		
12.	rec rep (2,2) : P1	
13.	exec CS	
14.	Rel req (2,2)	
15: rec rel (2,2)		rec rel (2,2)
16. del req (2,2)		del req (2,2)
}		

// 2 process requests for CR at same logical time

Forced to shuffle since Pid (1) < Pid (2)



1. req (1,1)	req (1,2)	
2. rec req (1,2)	rec req (1,1)	rec req (1,1)
3.		rec req (1,2)

4.		rep req (1,1)
5.		rep req (1,2)
6. rec rep(1,1) : P_3	rec rep (1,2) : P_3	
// STARVATION	HAPPENS HERE	
7.	rep req (1,1)	
8. rec rep(1,1) : P_2		
9. exec CS		
10. rel req (1,1)		
11.	rec rel (1,1)	rec rel (1,1)
	del req (1,1)	del req (1,1)
12. rep req (1,2)		
13.	rec rep(1,2) : P_1	
14.	exec CS	
15.	rel req (1,2)	
16. rec rel (1,2)		rec rel (1,2)
17. del req (1,2)		del req (1,2)

. Constraint for bringing Fairness :-

① For a process, if one of the Req is outstanding, cannot bring another req for same process in the queue

② If the current Req nos. is greater than the received req no, then the received request is old, so delete it & continue.

Total nos. of msgs. reqd = $\boxed{3 * (n-1)}$

(n-1) req, (n-1) rep, (n-1) rel for 'n' processes.

Ricart - Agarwala's Algo (D-Mutex) (non-token)

Message complexity is reduced to $2(N-1)$

P1 X	P2 X	P3
(1,1) | (2,2)	(1,1) | (2,2)	(1,1) | (2,2)
		X X
1. Request (1,1)		
	Rec Req (1,1)	Rec Req (1,1)
	Request (2,2)	Rep Req (1,1)
Recv Req (2,2)	Rep Req (1,1)	Rec Req (2,2)
Recv Rep (1,1) : P2	Recv Rep (2,2) : P3	Rep Req (2,2)
Recv Rep (1,1) : P3		
exec C.S		
Rep Req (2,2)	~~Recv~~	
	Recv Reply (2,2) : P1	
	exec C.S	

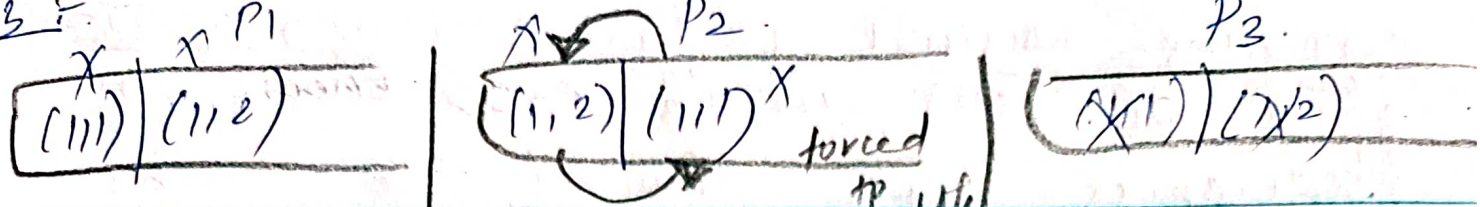
The queue entry is deleted from its own when it enter C.S

Once Reply is given, immediately request is deleted, instead of waiting for Release msg.

No need of any explicit Release Msg

Reduced to 2 phase algo (Req, Reply)

ase 3 :-



P_1	P_2	P_3
Request (1,1)	Request (1,2)	Rec Req (1,1)
Recv Req (1,2)	Rec Req (1,1)	Recv Req (1,2)
Recv Rep (1,1) : P_2	Rep Req (1,1)	Rep Req (1,1)
Recv Rep (1,1) : P_2		Rep Req (1,2)
Exec C.S	Rec Rep (1,2) : P_3	
Rep Req (1,2) :		
	Recv Reply (1,2) : P_1	
	Exec C.S	

There is a problem with fairness because an entry is deleted immediately a Reply is sent.

↳ But this is solved bcoz, the entry is deleted in Receive, not in sender. It is deleted only when it enters into C.S.

↳ ?

For outstanding requests, send new request after a time-out. By checking the sequence no. if it is older / higher only it can process the live request, else it is an old Request.

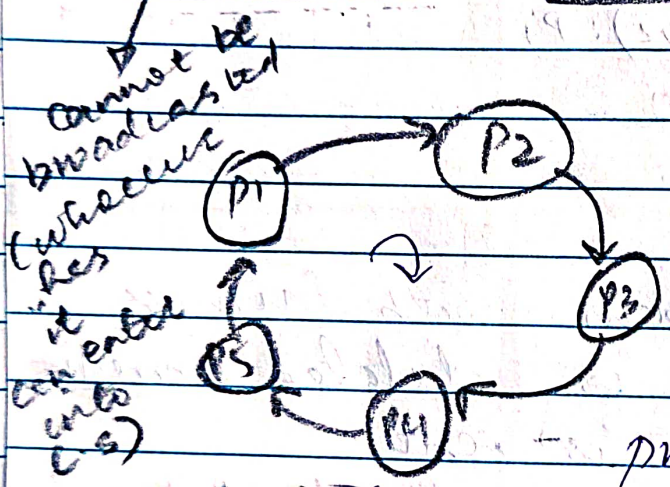
↳ This solves liveliness problem.

Supporting Mutual Exclusion using Contradiction

If 2 processes are in C-S, their process sequence no. should be same (without freshness condition), which is not possible.

So it is that only one process can exec in C-S

TOKEN BASED D-MUTEX (SUZUKI-KASAMI'S Algorithm):-



Each Process passes the token in the circular ring of processes which are predefined. Then if the process needs it (requirement), it takes the token, else passes it to neighbour.

Limitation:-

If P5 needs the token, P1 cannot send it to P2., not to P5.

The process does not know who needs the token. [does not know the demand]

This leads to starvation. and has performance issue.

To solve this :-

2 Data structures are used :-

R - Request vector (available at every node)

T - Token Vector (available as only 1 instance)

size ⇒ n [no. of processes] (total)

Algorithm :-

Request :-

Inc Ri[i] [Request for its own entry in R]

and broadcast it

Response :- [* to identify outdated Request, it should not process old Request]

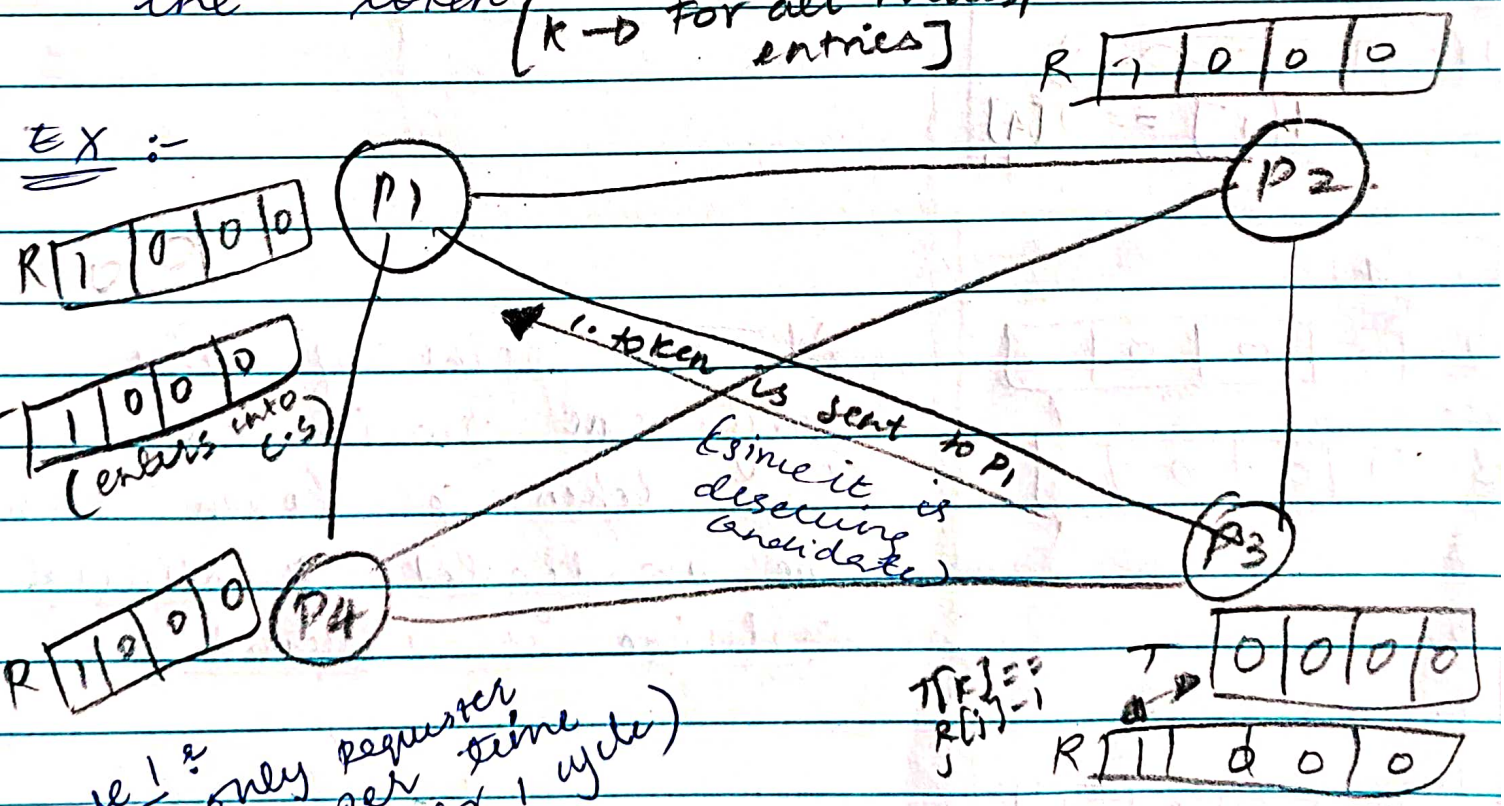
On Receiving the broadcast message,

$R_j[i] = \max(R_j[i], R_i[i])$ (every node)

If a node e has T then, if

T[K] == $R_j[i]$ - 1, pass the token [K → for all values/entries]

Ex :-



case 1 :- only requester per time (for 1 cycle)

(Im) If token is not being used but it is possessed by a node → IDLE TOKEN

In detail ex :-

P1

1. R [0|0|0|0]
2. R [1|0|0|0]
3.
4.
5. T [0|0|0|0]
6. Enter C.S
7. T [1|0|0|0]
8.
9. [1|0|0|1]
10. R1[4] == T[4] +1
11. T [1|0|0|1]
12. exec CS
13. T [1|0|0|1]

P2

R [0|0|0|0]

R [1|0|0|0]

T

R3[1] == T[1] + 1

If a request has to be broadcasted, the receiver's be lesser

[1|0|0|1]

P3

R [0|0|0|0]
T [0|0|0|0]

R [1|0|0|0]

should receive only (seq nos., process id) than ex

[1|0|0|1]

P4.

R [0|0|0]

[1|0|0|0]

[1|0|0|1]

At one point of time, only one process can have token & token cannot be broadcasted → Mutual exclusion is achieved

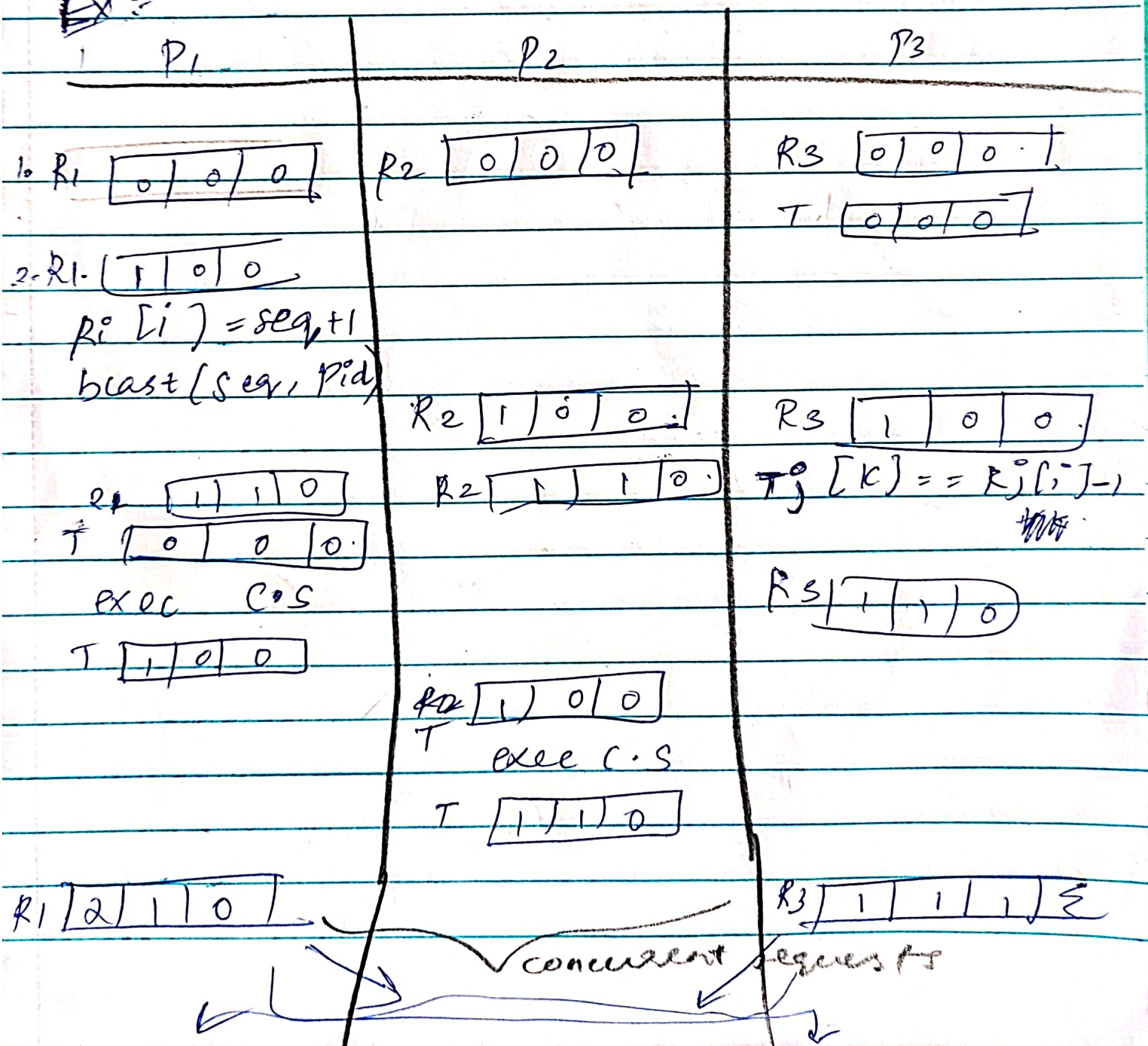
To achieve fairness :-

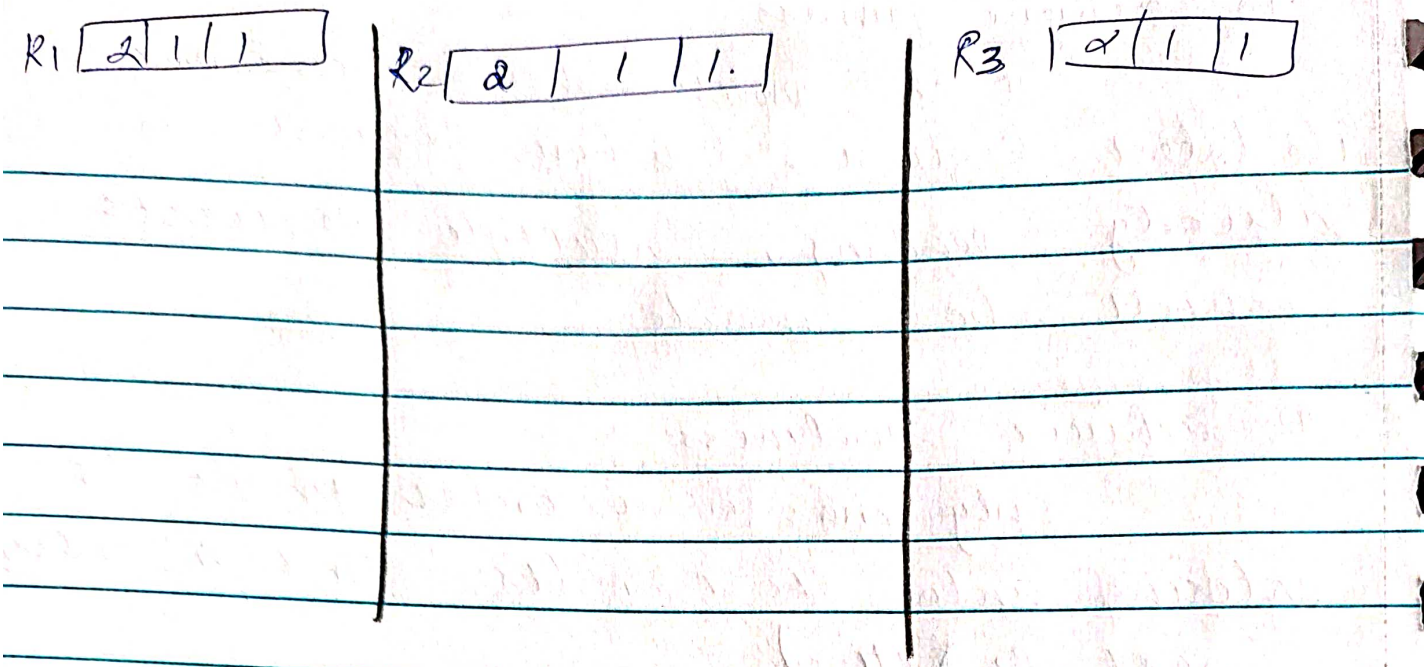
(Each process should remember its latest Request. When Request is already pending, multiple Requests cannot be made

To achieve Orderliness :-

only one in a order process is entering into the C.S, (Ex : First P_1 entry, next P_2 }, etc...]

Ex :-




R1
2
1
1
R2
2
1
1.
R3
2
1
1