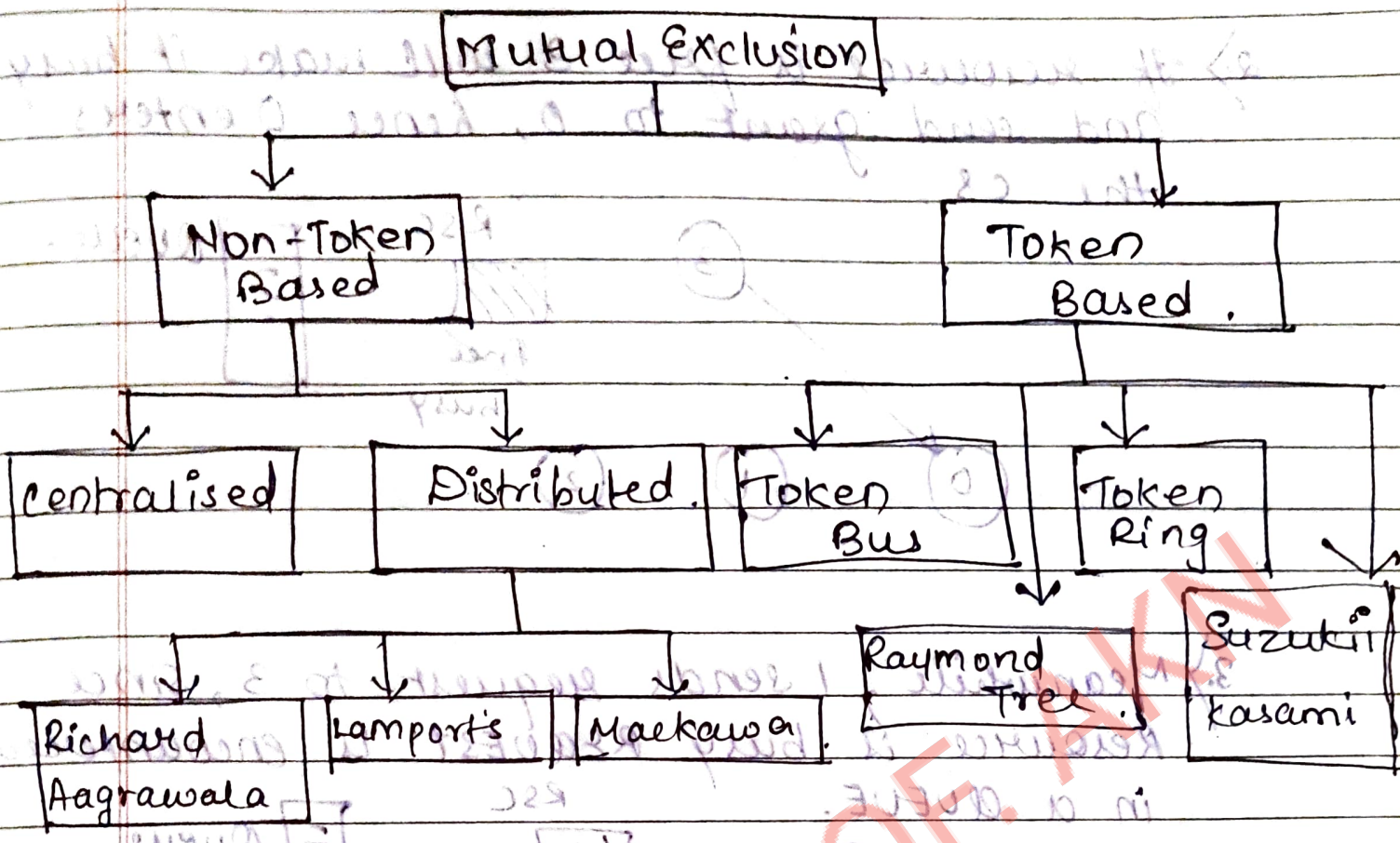


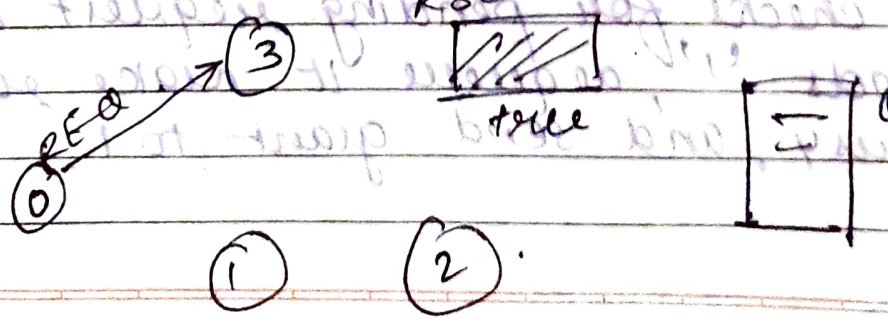
* Non-Token Based Algorithm:

1. Centralised

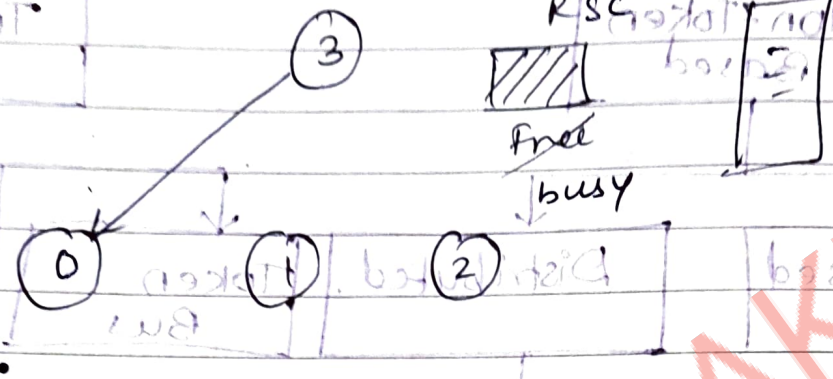
Used for distributed system but as centralised approach.

(1) There is a co-ordinator [3] which manages critical section [resource]

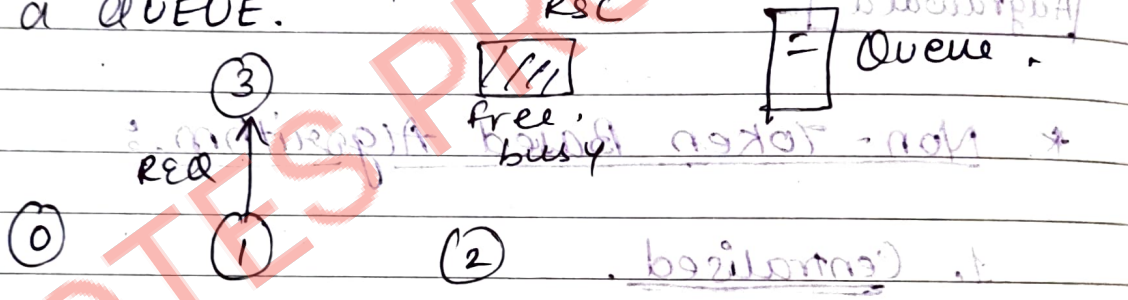
If any process [0] wants to enter critical section [CS] it sends request to 3



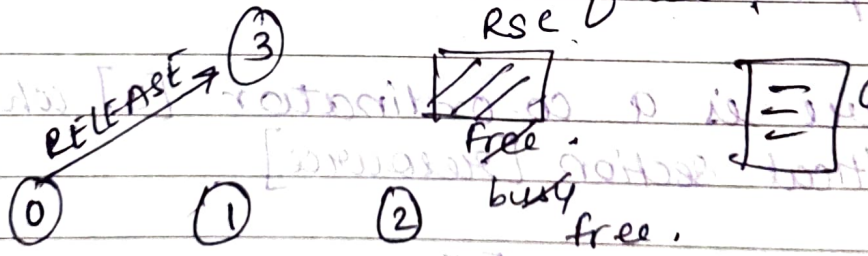
2.) If resource is free 3 will make it busy and send grant to 0, hence 0 enters the CS.



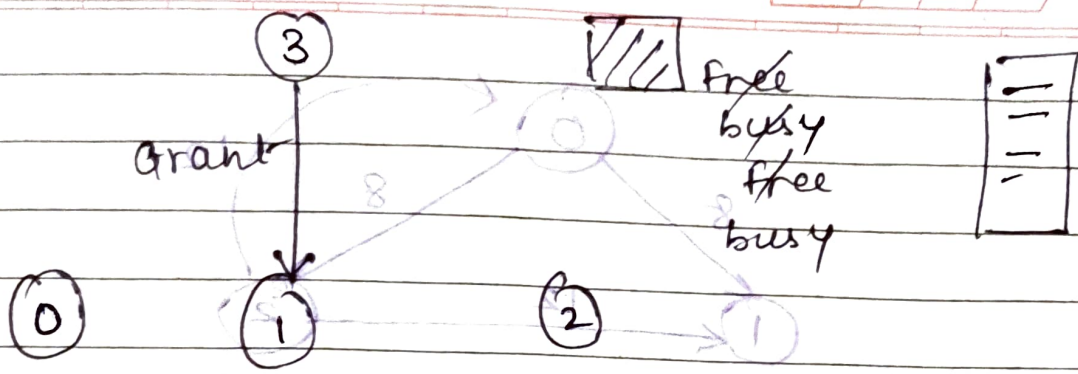
3.) Meanwhile, 1 sends request to 3, since Resource is busy REQUEST is encountered in a QUEUE.



4.) '0' after completion sends RELEASE to 3 which makes resource free.



5.) 3 checks for pending request in queue finds '1', dequeue it make resource busy, and send grant to 1.



Adv : 1. Simple three level protocol
[REQ → GRANT - RELEASE]
2. Mutual Exclusion successfully achieved.

Disadv : 1. Single point of failure.
Sol? : Keep backup.

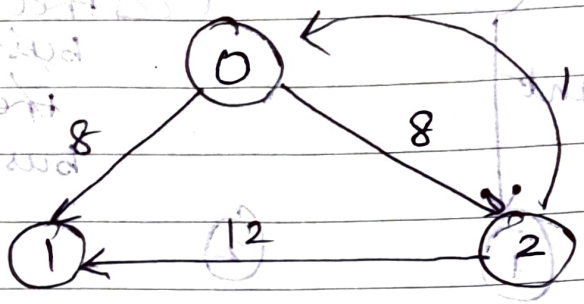
2. Nodes getting confused why co-ordinator is not replying is it because resource is busy [eg. step 3] or co-ordinator has failed.
Sol? : Co-ordinator should send reply for all messages.

D2D * Distributed

1. Richard Aagrawala.

Note : Since it is distributed, there is no co-ordinator.

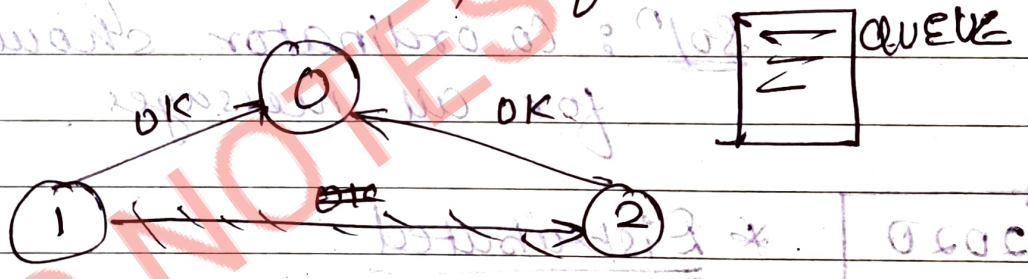
1. Any process that wants to enter critical section i.e. 0 and 2 will broadcast request only with timestamp.



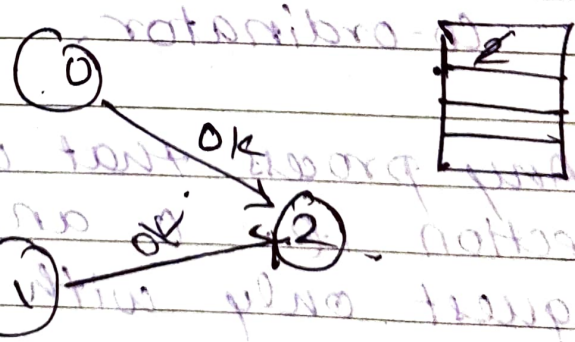
2.] One is neither in CS and is not willing to enter CS, so it send the consonants to both by sending "OK".

Comparing the timestamp 0 is the winner and hence 2 send the consents by sending "OK" to 0.

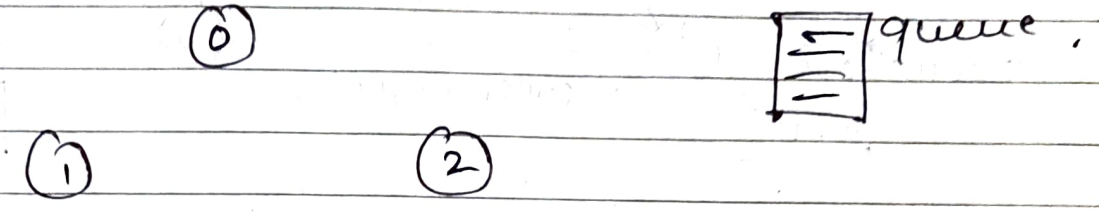
Since zero got N-1 consents (OK), 0 enters the CS and maintains a queue which contains entry of 2.



3.] When 0 comes out of the CS, it checks the queue finds 2, dequeues 2 and send "OK" to 2.



4] Now 2 got N-1 consent and hence enter the CS.



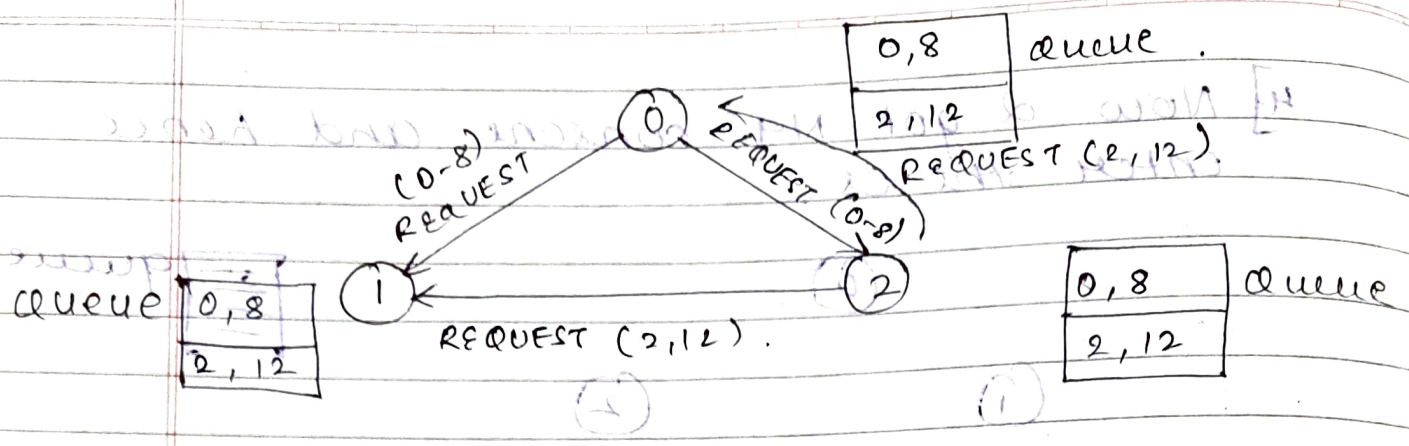
Adv : 1) No single point of failure.
2) Mutual Exclusion successfully achieved.

Disadv : 1) One to N-1 consents lot of congestion created in network.
2) Node getting confused when there is no reply, so always send an 'ack'.

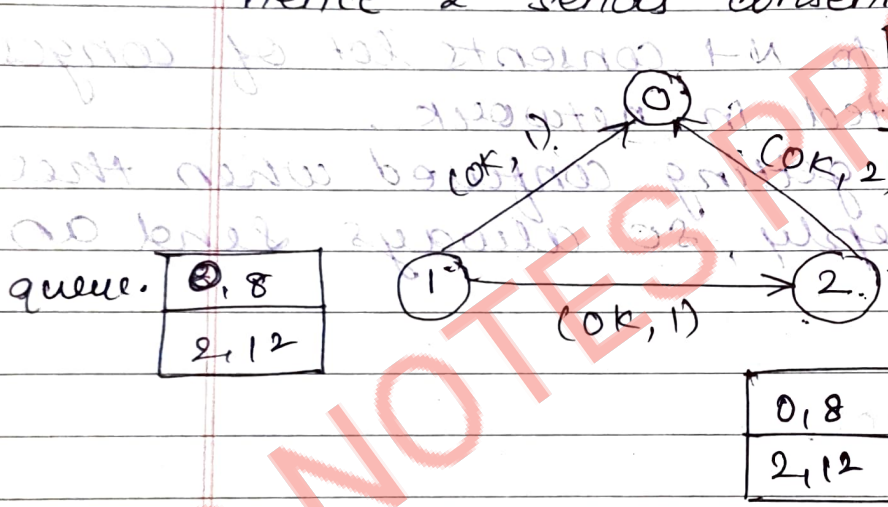
2] Lamport's Algorithm

Note : Everyone will maintain the queue.

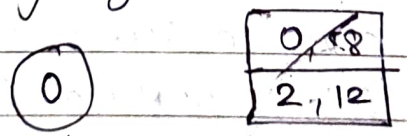
1. Any node that wants to enter the CS sends request to all in the form of "REQ (pid, Timestamp)" and every node maintains the queue which would be updated with respect to ascending order of timestamp (sorted).

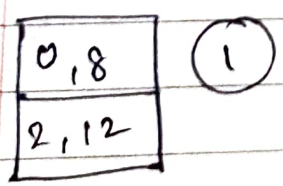


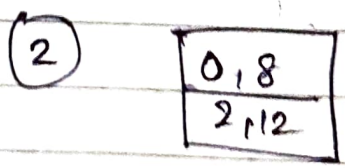
2. Since 1 is neither in CS ~~nor~~ nor wants to enter the CS, ~~therefore~~ it sends the consent in the form of "(OK, senders Pid)". Looking at the timestamp 0 is the winner and hence 2 sends consent to 0.



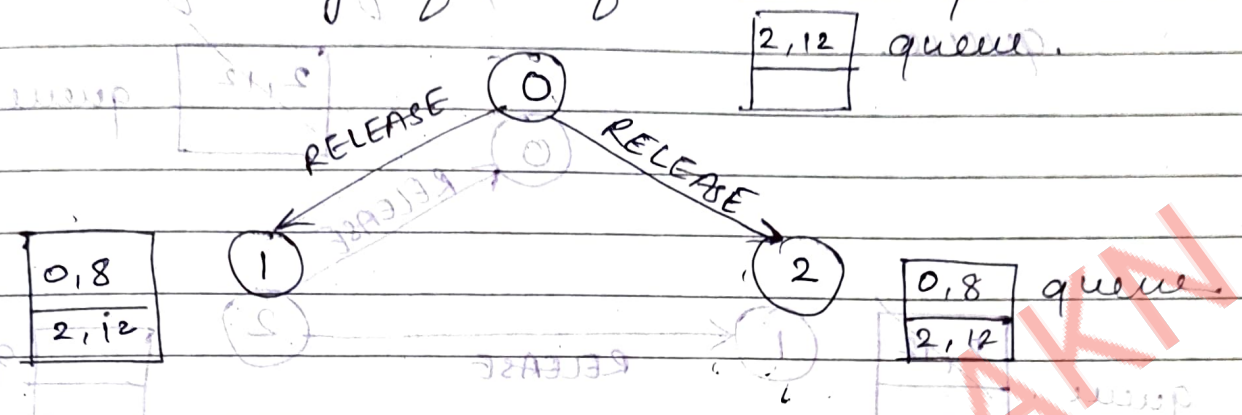
3. 0 got N-1 consents and hence enters the CS, but before entering the CS it will dequeue its own entry from its own queue.



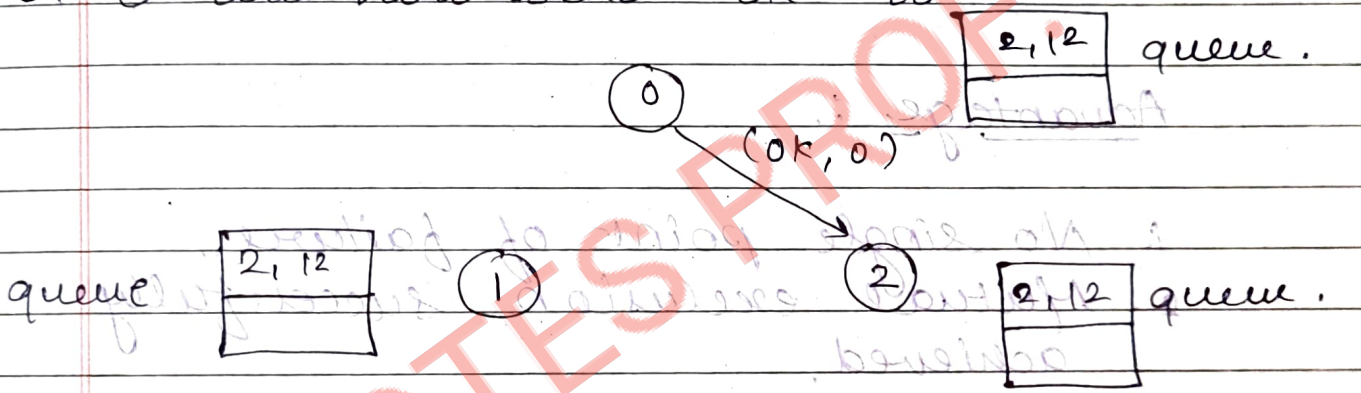




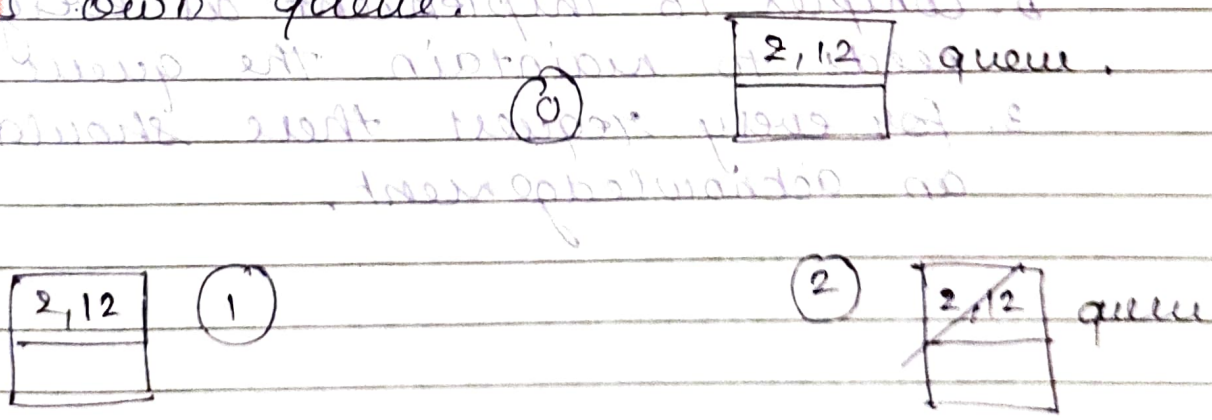
4. When 0 will come out of the cs, it will broadcast "RELEASE" and 1 and 2 will remove entry of 0 from their queue.



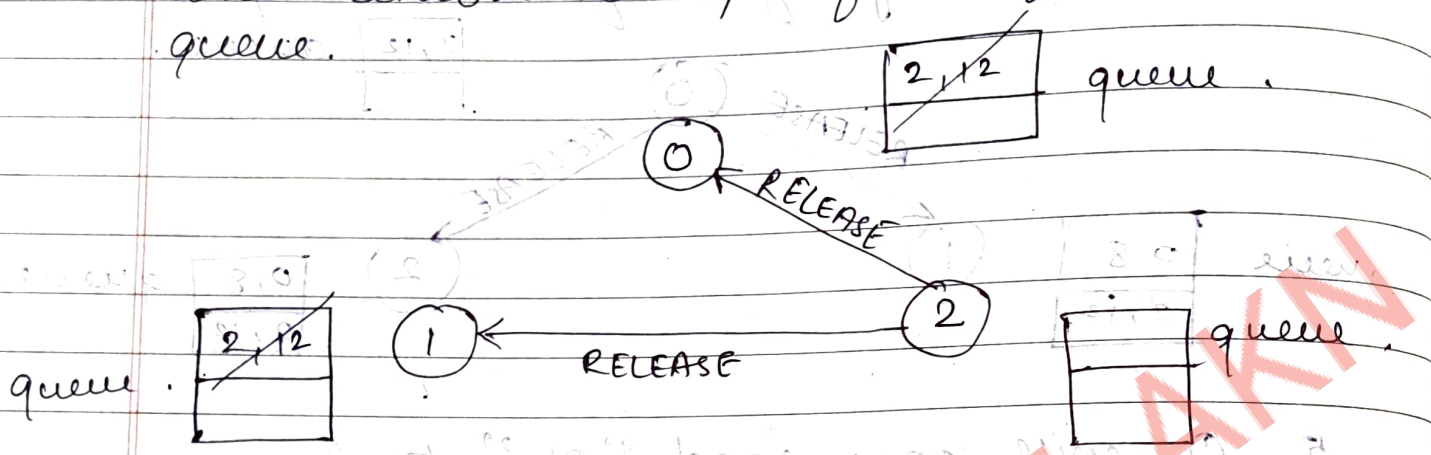
5. 0 will now send "OK" to 2.



6. Since 2 got N-1 consent. Therefore it enters the cs but before entering the cs it removes its own entry from its own queue.



7. When 2 comes out of the CS it will broadcast "RELEASE" and everyone will remove entry of 2 from their queue.



Advantage :.

1. No single point of failure.
2. Mutual exclusion successfully achieved.

Disadvantage:

1. Lot of congestion due to broadcasting.
2. Complex to implement as everyone needs to maintain the queue.
3. For every request there should be an acknowledgement.

3. Maekawa's Maekawa:

It works on the principle of Quoram (Groups) and maintains following data structure.

i) Q (Quoram)
ii) V (voting) : the value is true when a permission is given for cs.
iii) RD (Request Differed Queue).

Step 1 : Initial snapshot.

(1) Q = {1, 2, 3}
V = F
RD = { }

(2) Q = {2, 1, 4}
V = F
RD = { }

(3) Q = {3, 1, 4}
V = F
RD = { }

(4) Q = {4, 2, 3}
V = F
RD = { }

Step 2 : Assume 1 needs to enter the cs, so it sends to its Quoram members.

(1) Q = {1, 2, 3}
V = F
RD = { }

REQ

(2) Q = {2, 1, 4}
V = F
RD = { }

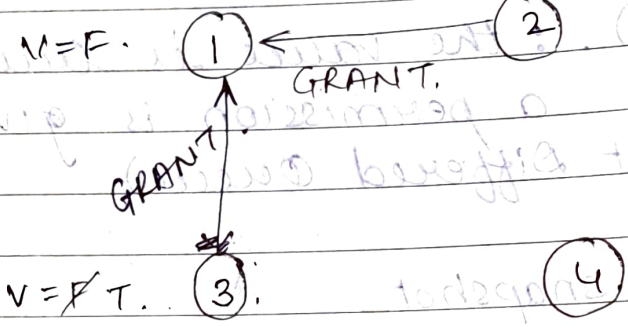
REQ

(3) Q = {3, 1, 4}
V = F
RD = { }

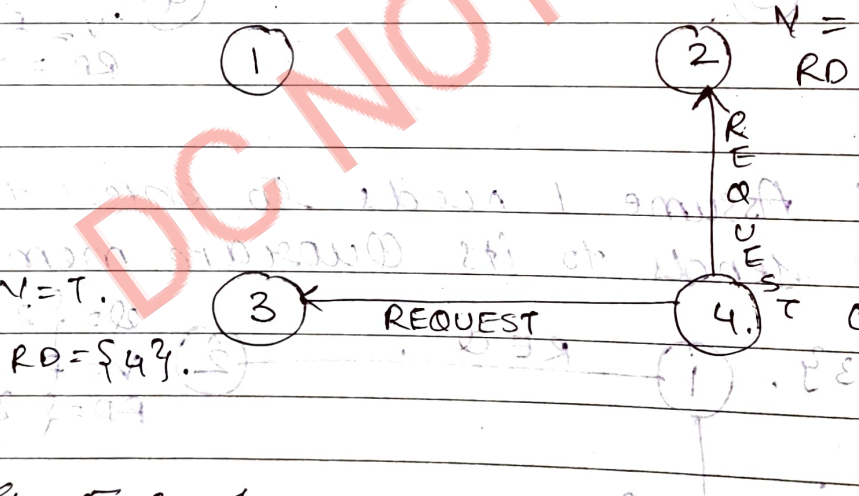
(4) Q = {4, 3, 2}
V = F
RD = { }

is false.

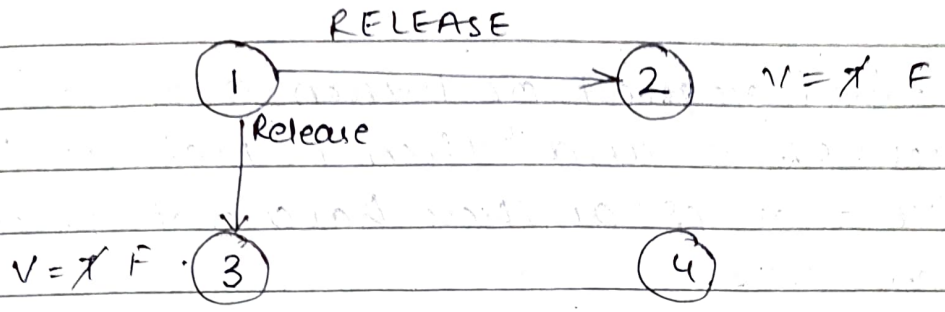
Step 3 : Since voting of 2 and 3, if FALSE which means that they have not permitted anyone to enter the CS, so both will set V = TRUE and send "GRANT" to 1. 1 got N-1 consent (w.r.t Quorum) 1 enters CS.



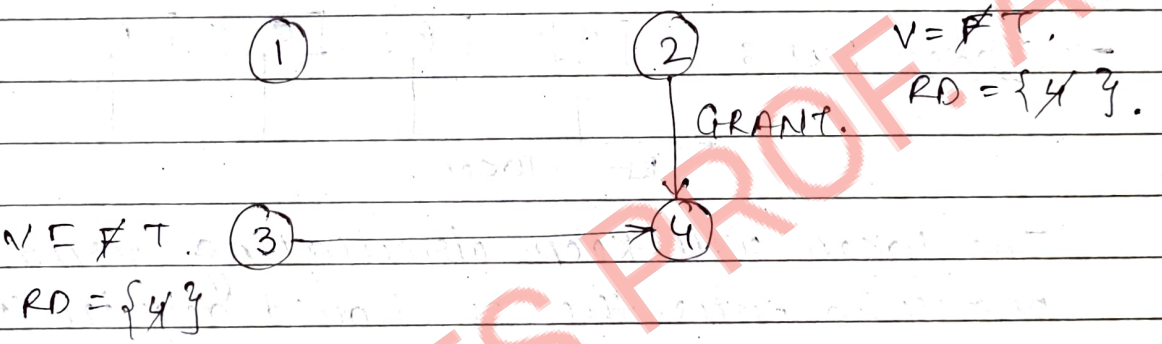
Step 4 : In a meanwhile, 4 request to enter CS to its Quorum members i.e. 2 and 3, since voting of 2 and 3 is TRUE, Enqueue request of 4 in RQ.



Step 5 : 1 comes out of CS and sends "RELEASE" to its Quorum which will set voting to FALSE.

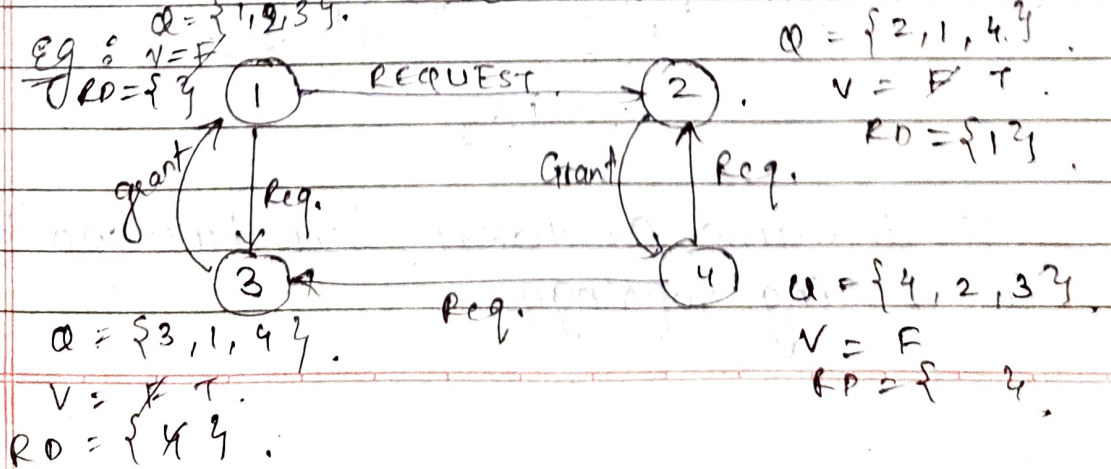


Step 6 : Now, 2 and 3 check RD and find 4, dequeue it set voting to TRUE and send "GRANT" to 4 which will make 4 to enter CS.



Advantage : 1. No single point of failure.
2. Mutual exclusion successfully achieved.
3. Since no broadcasting therefore proper utilization of bandwidth.

Disadvantage : 1. Complex to implement.
2. Leads to problem of deadlock.

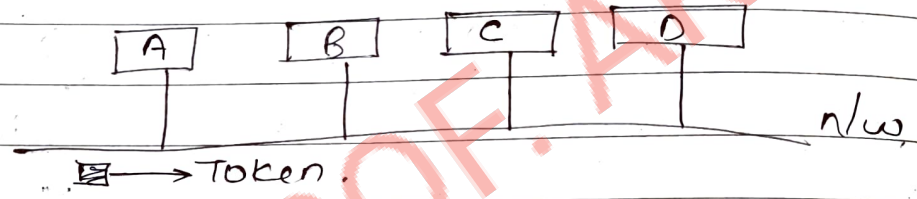


3 received request of 1 then 4, 2 received request of 4 and then 1. Neither 1 nor 4 will enter CS as they have not received N-1 consents.

Sol^n: Use time stamp.

* Token based Mutual Exclusion algo:-
The one who has the token will enter CS. {critical section}.

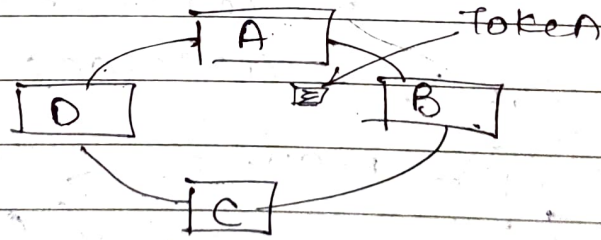
→ Token Bus :



The token will keep circulating serially the one who enter the CS should acquire it, the only issue is of starvation.

Sol^n: use Round Robin Technique.
↳ If static method is used token will be present at every node for 'x' amt of time.
↳ If dynamic, the token would be based on dynamic timing {jitne samay ke liye node ko token chahiye} with threshold policy.

→ Token Ring.



It is same as token bus but working in a ring topology.

28/01/2020

Imp

* Raymond Tree Algorithm:

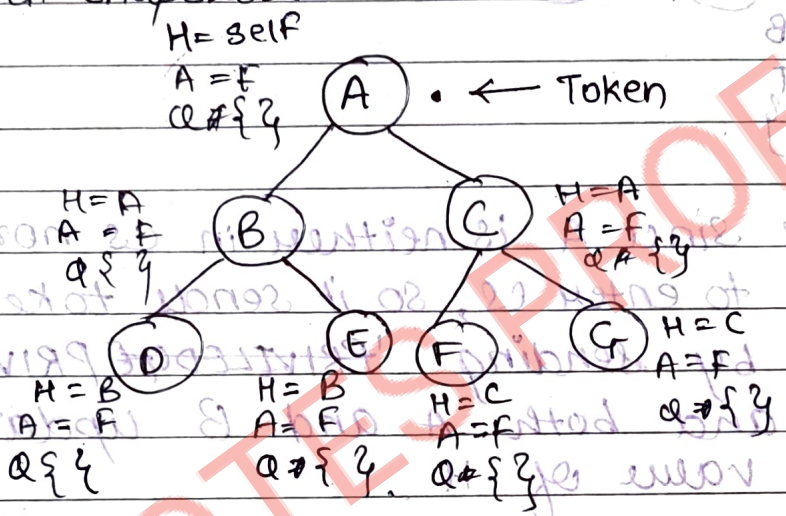
This work will work on Binary search tree structure and use following set of variable.

H = hold (Info about token)

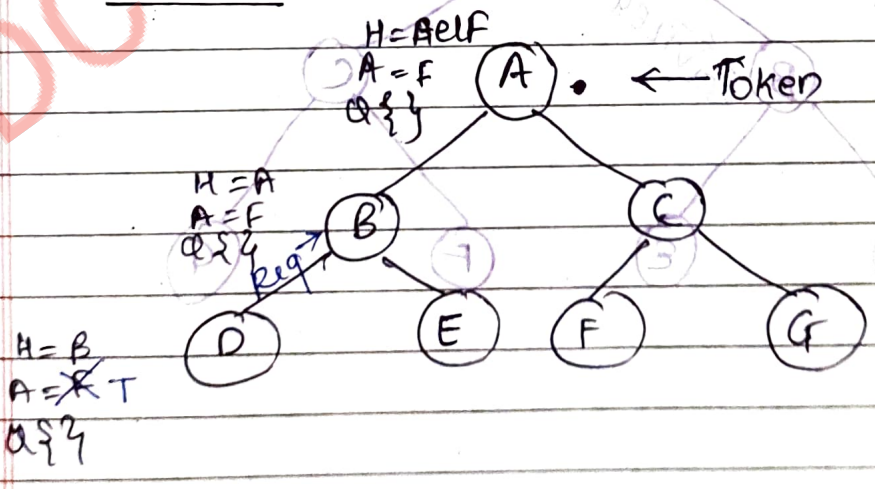
A = Ask (true or false).

Q{}. = Queue (Data structure for pending req).

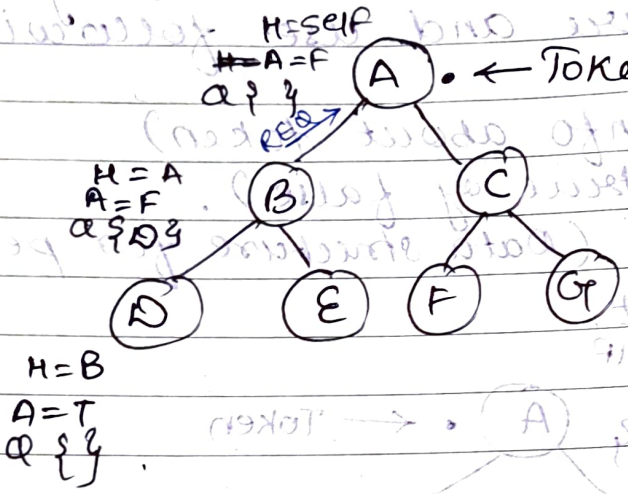
Step1 : Initial snapshot.



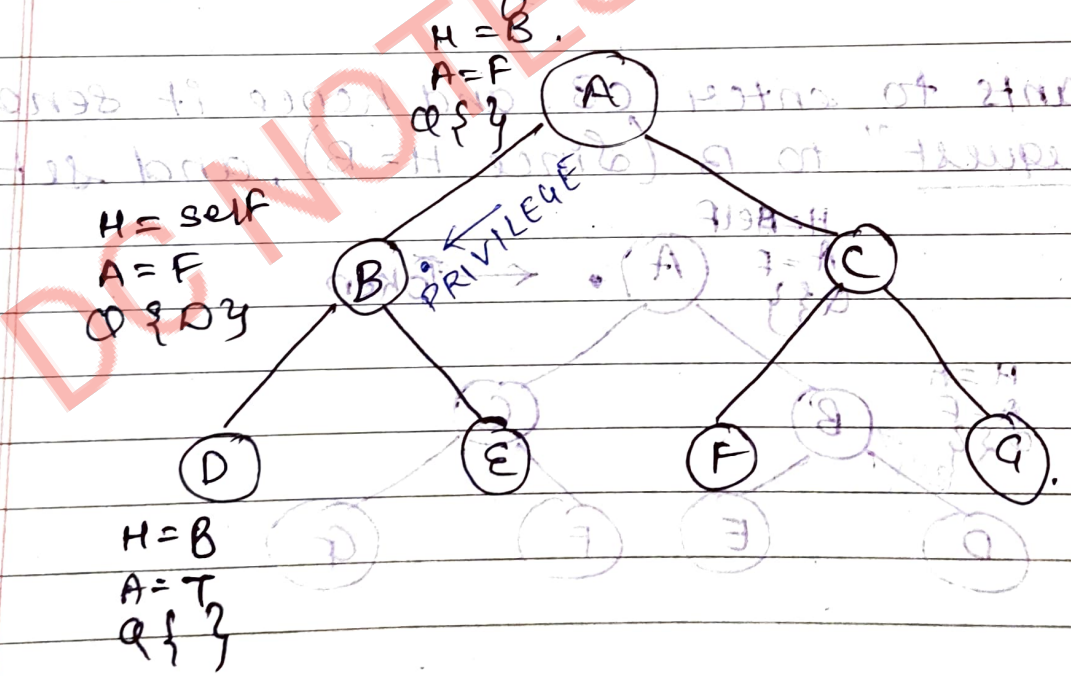
Step2 : D wants to enter CS and hence it sends "request" to B (since H=B), and set A = True



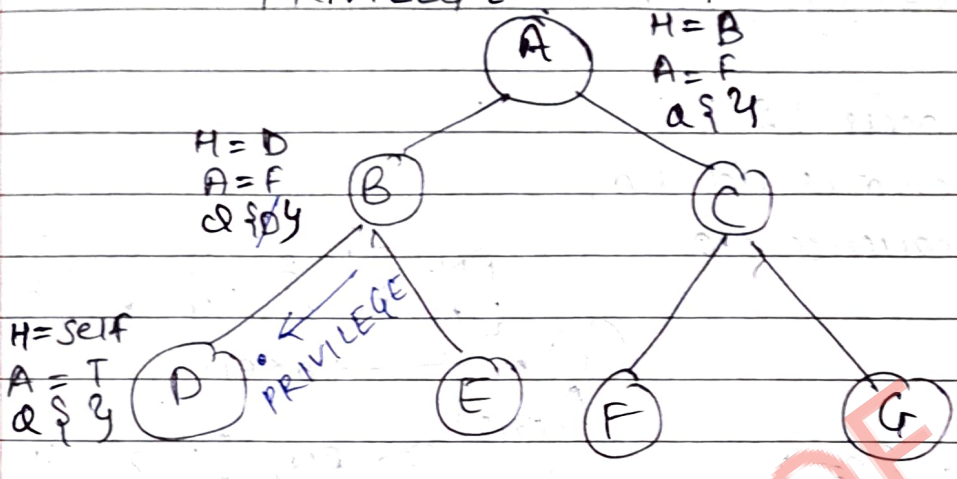
Step 3 : B enqueues request of D in a queue and sends "REQUEST" to A (since H=A).



Step 4 : Since A is neither in CS nor wants to enter CS, so it sends token to B by sending "~~PRIVILEDGE~~ PRIVILEGE" and both A and B updates their value of H.



Step 5 : Since A [Ask] of B is false, B checks Q finds D, dequeues it and sends token to D. By sending "PRIVILEGE"



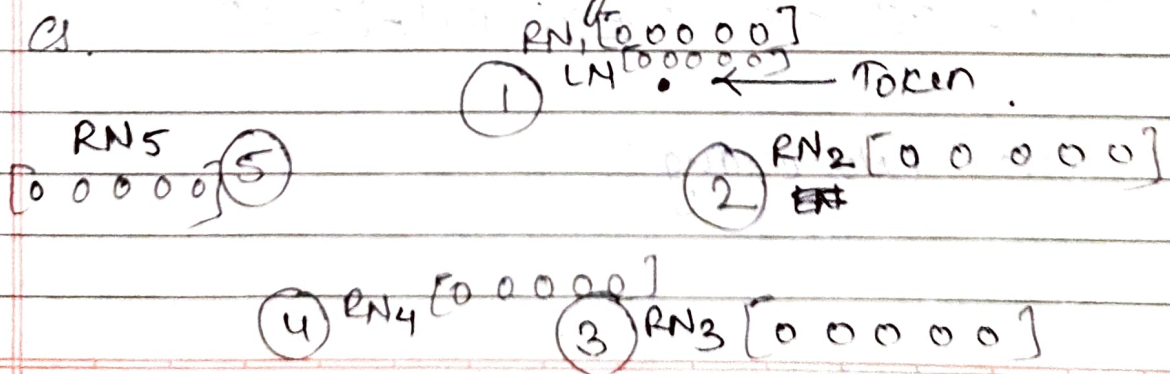
Now, D enters CS, after coming out of CS it sets A=F.

29/01/2020

(*) Suzuki Kasami

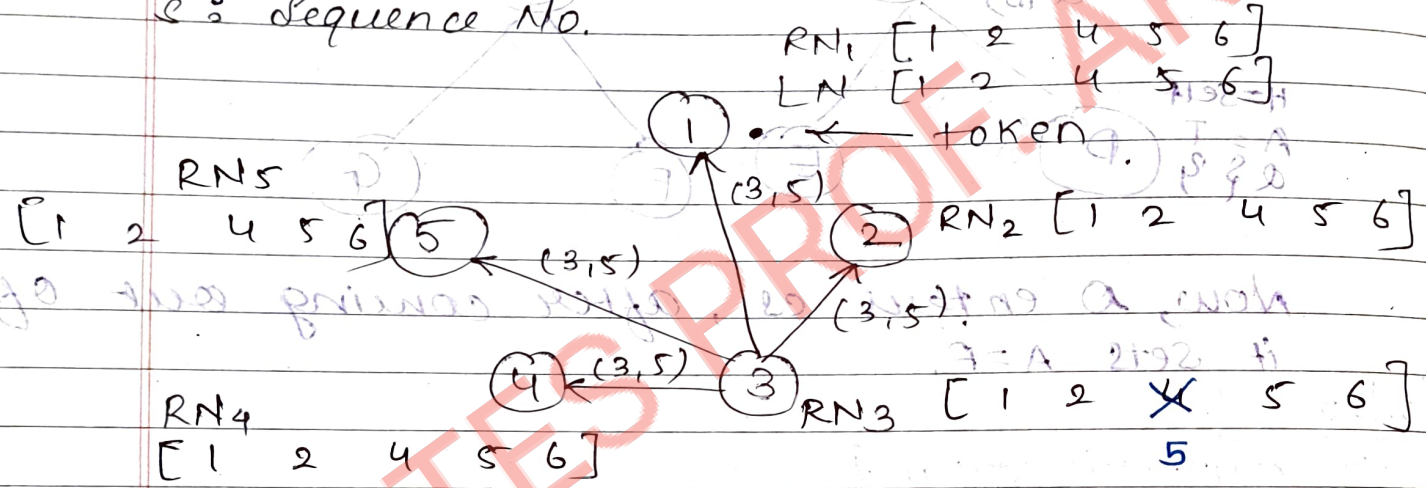
Step1: Initial Snapshot.

Every node maintains RN array and the node having the token maintains LN array. RN array will have information about how many times node has requested for CS, whereas LN array will have information about how many times node has entered CS.



Step2: Now, if 3 wants to enter the CS it would first increment $RN_3[3]$. And now send broadcast request to everyone in the form ~~$RN_3[3]$~~ of (P_i, S_j) [P3, S5]

P: Process ID
i: denotes id no.
s: Sequence No.



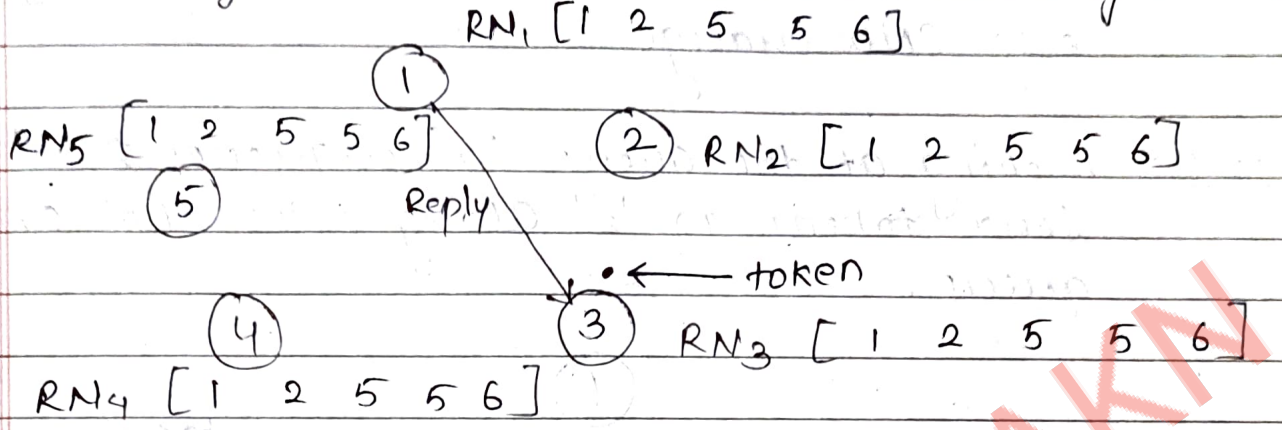
Step3: Checking for outdated request.

if $(RN_i[P_i] < S_j)$ then $(RN_i[P_i] = S_j)$ ~~else (NO change)~~.

$RN_1[3] < 5 = T \Rightarrow RN_1[3] = 5$
$RN_2[3] < 5 = T \Rightarrow RN_2[3] = 5$
$RN_4[3] < 5 = T \Rightarrow RN_4[3] = 5$
$RN_5[3] < 5 = T \Rightarrow RN_5[3] = 5$

else (NO change).

Step 4 : @ 1 is neither in CS nor wants to enter CS, so it gives token to 3 with LN array.



Step 5 : When 3 was in CS '2, 4, 5' requested for token and hence everywhere value of RN ~~value~~ array is.

RN [1 ~~2~~ 5 ~~6~~ ~~7~~]
3 6 7

Step 6 : Now, '3' comes out of CS, so it updates LN.

LN [1 2 4 5 6]
RN3 [1 3 5 6 7]

~~LN[3] = RN3[3]++~~
∴ LN [1 2 5 5 6]

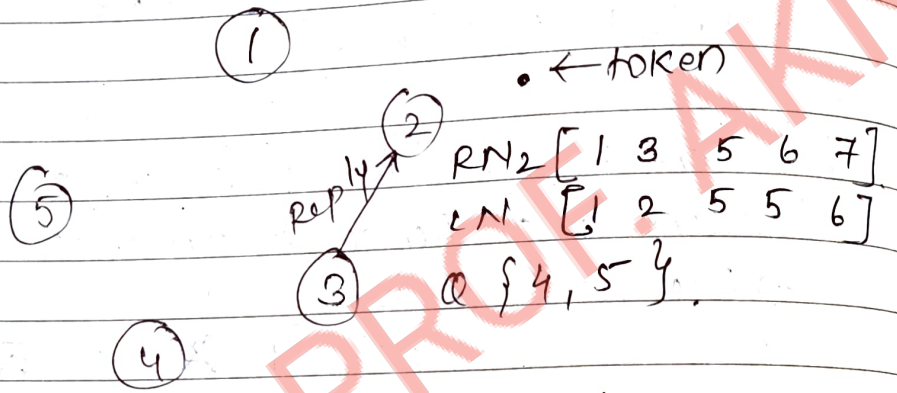
Logically, it should be incremented by 1.

Step 7 : Checking for pending request.

if (RNi[j] = LN[j] + 1) then (enqueue j in Q)
RN3[1] = LN[1] + 1 = F
RN3[2] = LN[2] + 1 = T Q {2}.

$RN_3[4] = LN[4]+1 = T$ @ {2, 4}
$RN_3[5] = LN[5]+1 = T$ @ {2, 4, 5}
else (NO change).

Step 8: Looking at the queue '3' dequeues ~~it~~ '2' and send tokens to '2' along with LN and queue.



The only problem with this approach is this may lead to the problem of starvation, i.e. even though '5' has requested first but due to sequential checking 2 gets the token, and hence use time stamp along with the request.

(*) Algorithm

Step 1 : Any node that wants to enter CS would first update its RN entry as:
$RN_i[i]++$
Eg: $RN_3[3]++$
and send broadcast request to all in the form of (P_i, S_j) $[P_3, S5]$.

Step 2 : Every node checks for outdated request.
if $(RN_i[P_i] < S_j)$ then $(RN_i[P_i] = S_j)$

Step 3 : When any node enters CS and comes out it updates LN array for its own entry.

Step 4 : The node that came out of the CS checks for pending request.
if $(RN_i[j] = LN[j] + 1)$ then enqueue (j in Q)

Step 5 : Handover the token to that node which is dequeued from the queue.