7.8.23

DISTRIBUTED SYSTEMS

- work is split among various systems ; load must be balanced
- does not need to be homogenous; heterogenous systems inevitable

problems ?

1) Absence of global, shared memory.
2) Local time difference; Ordering of computation might be meaningless.

ms also matters ; µs precision-based code

Absence of global clock.

All other problems caused by these problems.

CLOCK

piezo-electric quartz

physical clocks - difficult to be synchronised.

logical clock ✓

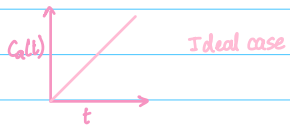
hard to do, bc of phy & chem prop

8.8.23

Physical clock sync

$C_a(t)$: clock for processor a at time t

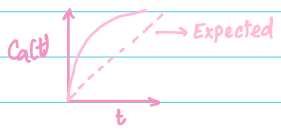
Ideal case $C_a(t) == t$



$\frac{dC}{dt} = 1$ → rate of change is constant

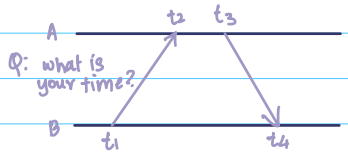
Clock skew → first derivative

$\frac{dC}{dt} > 1$ clock faster than Ideal time / any other clock



$\frac{d^2C}{dt^2}$ → clock drift

B is trying to sync with A's clock



1. B sends a query at t_1
2. A receives query at t_2
3. A processes & responds at t_3
4. B receives time at t_4

$t_1 - t_2$: time delay

time spent in channel ; traffic/routing issue/ buffering

B has to know delay in receiving message

2-way transmission delay : $(t_4 - t_1) - (t_3 - t_2)$

avg. delay : $[(t_4 - t_1) - (t_3 - t_2)] / 2$

represented as integers; only updates during events; "happened-before" relationships

9.8.23 **Lamport's Logical Clock** : distributed algorithm; implemented in multiple places

STEPS

S1: concurrent

S2: message passing

S3: receives 8:30, although self-time shows 8:29.

∴ won't update.

ordering is very important. Otherwise incorrect

" → " : happened before

STRICT PARTIAL ORDER — SIMPLE PO only reflexive

1. Irreflexive : $a \not\to a$
2. Antisymmetric : if $a \to b$, then $b \to a$ is false
3. Transitive : If $a \to b$ & $b \to c$, then $\Rightarrow a \to c$

Rule 1 : local event

If $a \to b$, $C_i(a) < C_i(b)$

$i \to$ same system/processor.

$C_i(e) = C_i(e) + d$ // $d \to$ some increment; ①

Rule 2 : message passing

$a \to b$, $C(a) < C(b)$

$C_j(e) = \max(C_j(e), t_m + 1)$

Rule 3:

if $a \not\to b$, $b \not\to a$

then $a \parallel b$ // a & b are concurrent

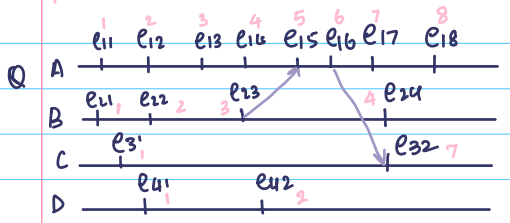
LIMITATION

- when causality is pre-known, order can be determined (clock values)

but, vice versa, when clock values are only given, causality cannot be determined

- system cannot distinguish local clock value & time received. you don't know why system not smart enough

performed time update (machine is not intelligent)



start w B.

e_{17} & e_{24} are concurrent

can only say smthn happ b4 smthn else if there is msg passage / is on same system.

e_{14} causally affects e_{24}? $e_{14} \rightarrow e_{24}$?

$e_{14} \rightarrow e_{15}$

$\rightarrow e_{16}$

$\rightarrow e_{23}$

→ vector values, instead of single int values

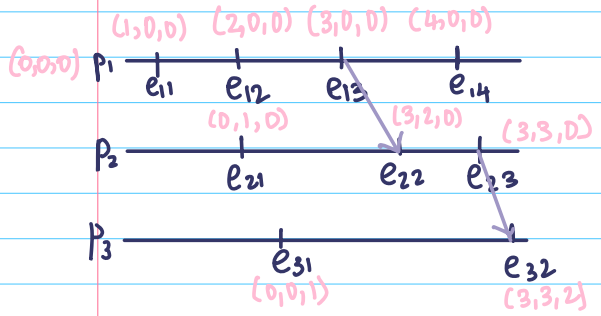
10.8.23

VECTOR CLOCKS

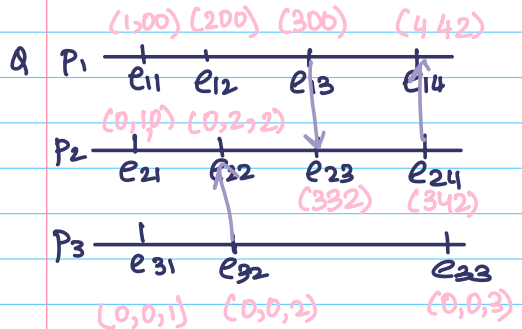
$C_i[i,j,k]$

I: $C_i[i] = C_i[i] + 1$

$\forall_{k \neq i} C_i[k]$, $C_i[k] = \max(C_i[i], tm)$



Information sent only through msg passing



$e_{22} \not\rightarrow e_{13}$

(3,2,0) (3,0,0)

if atleast one of the events is lesser the event was the preceding one.

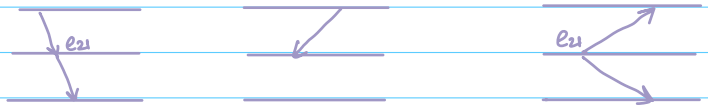
$\exists_i\ C_i[k] < C_j[k]$

21/8/23

$\forall k\ t_a[k] \le t_b[k] \Rightarrow t_a \le t_b$
$\exists k\ t_a[k] \not< t_b[k] \Rightarrow t_a \not< t_b$
} Rules of vector clocks

for concurrency if $t_a \not< t_b$ && $t_b \not< t_a \Rightarrow t_a \,||\, t_b$

NOTE:

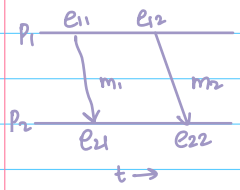


FATAL ERROR

logical error; not fatal

OK!

CAUSAL ORDERING OF MESSAGES

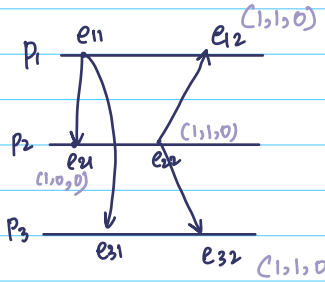


if send $(m_1) \rightarrow$ send (m_2), then
rec $(m_1) \rightarrow$ rec (m_2)

sender's rule · Ci)
1) increment i^{th} value, update tm
2) broadcast to everyone

receiver's rule Cj)
1) $C_j[i] = tm[i] - 1$
2) $\forall k,\ k \neq i\ \ C_j[k] \ge tm[k]$

received everything; consequent message.



e_{21} $i=1$ $j=2$
$C_2(0,0,0)$, $tm = (1,0,0)$

e_{31} $i=1$ $j=3$
$C_3(0,0,0)$, $tm = (1,0,0)$

22/8/23

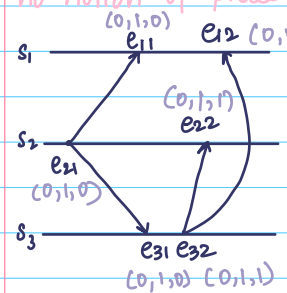
e_{12} $e_1(1,0,0)$ $tm(1,1,0)$: accept

e_{32} $C_3(1,0,0)$ $tm(1,1,0)$: accept

value of 2 → missed one message

received → @ the buffer.

no notion of processing events

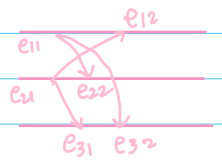


e_{11}: $C_1(0,0,0)$ $tm(0,1,0)$
$i=2$ $j=1$

e_{22} : $C_3(0,1,0)$ $tm(0,1,1)$
$i=3$ $j=2$.

streamlining receiving

Why ≥ in receiver rule 2?



e_{22} : C_2 (0,1,0) tm (1,0,0)

concurrent system: cannot stop other systems from sending.

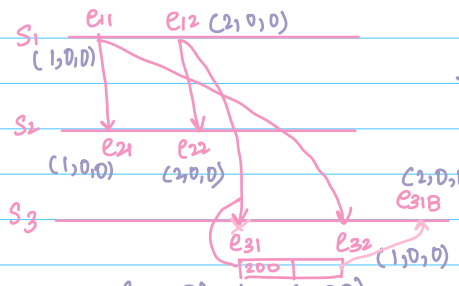
e_{22} (1,1,0)

e_{31} : (0,1,0)

e_{32} : (1,1,0)

e_{12} : (1,1,0)

1. receiver air initiated send
2. can receive concurrent messages.



→ causality is not maintained in S3.
first sent not first received

e_{31B} ✗ - clock still remains (0,0,0)

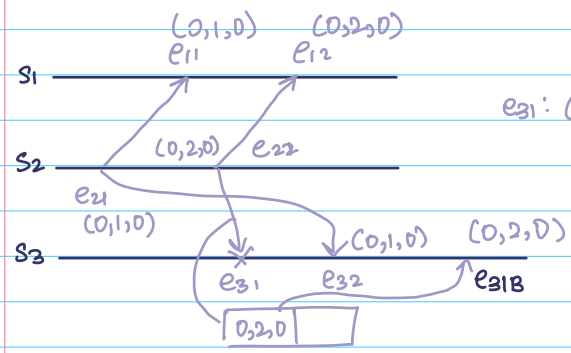
e_{21} C_2 (0,0,0) tm (1,0,0)

e_{31} C_3 (0,0,0) tm (2,0,0)
will not accept; buffer instead of delivering

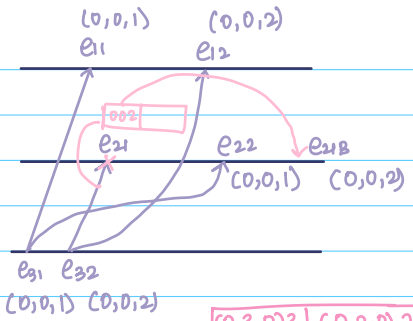
e_{32} C_3 (0,0,0) tm (1,0,0)
accept

e_{31B} C_3 (1,0,0) tm (2,0,0) accept.

Q.



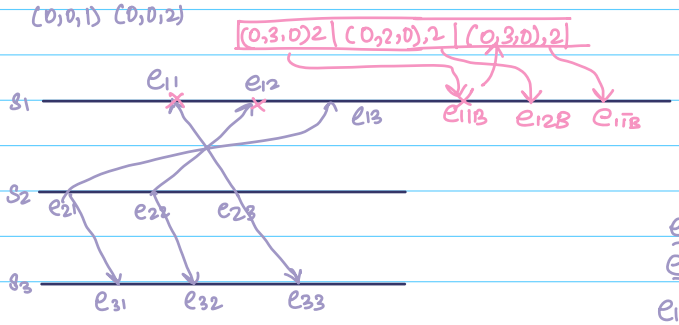
Q.



when do you check buffer?
→ after every receive, you can check.

2/18/23

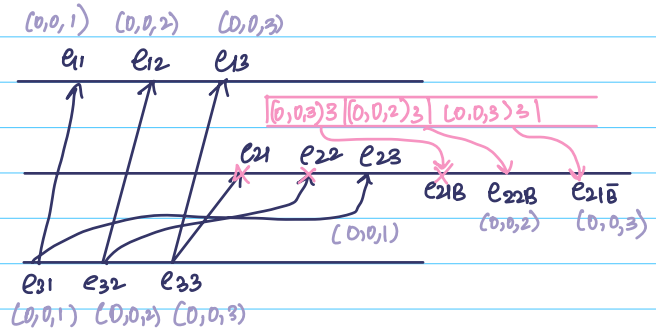
Case :



e_{11} : C(0,0,0) tm(0,3,0)
2 msg missing!

e_{12} : C(0,0,0) tm(0,2,0) buffer!
e_{13} : C(0,0,0) tm(0,1,0) Accept
e_{11B} : C(0,1,0) tm(0,3,0) buffer!

Q.



e_{11} : C(0,0,0) tm(0,0,1) ✓
e_{12} : C(0,0,1) tm(0,0,2) ✓
e_{13} : C(0,0,2) tm(0,0,3) ✓

[only works for broadcast]

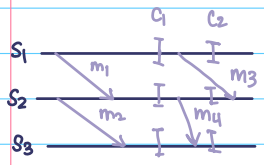
8/9/2023

GLOBAL STATE

snapshot / recording : consistency for

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

LS → send & receive events



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

$LS_2 = \{send(m_2), recv(m_1)\}$

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

If every send has a receive, strongly consistent global state
↳ no omitted messages.

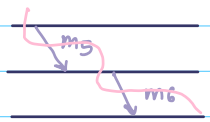
- not practically possible: cannot stop operation, affects performance (issues)

consistent global state : every receive has corresponding send

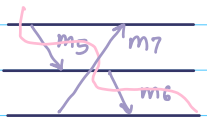
For C_2

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

} consistent
- has transit (here: m_3)
↓
only send, no receive recorded

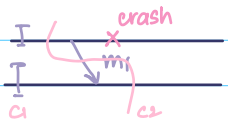
- 

$LS_1 = \{\}$
$LS_2 = \{recv(m_5)\}$ Inconsistent
$LS_3 = \{recv(m_6)\}$

- 

$LS_1 = \{\}$
$LS_2 = \{recv(m_5)\}$
$LS_3 = \{recv(m_6), send(m_7)\}$

Still inconsistent

- 

x - rolls back.

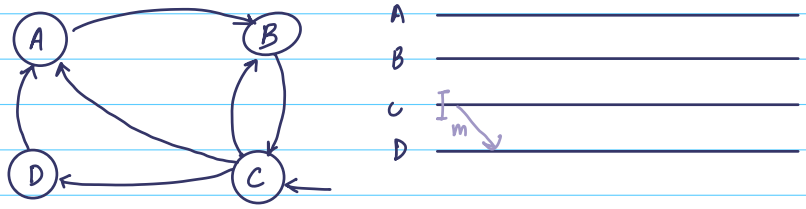
S1 will have no memory of m_1,

but S_2 will have receive.

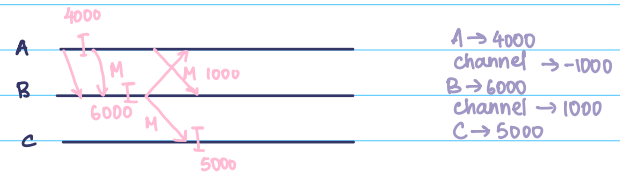
11/9/23

Global Snapshot Reloading Algorithm

If marker is seen, no transactions can happen before screenshot



19/9/23



record global state → to fix deadlocks; recovery

Distributed Deadlocks

resource trap cycles not easy to detect.

early solution 1:

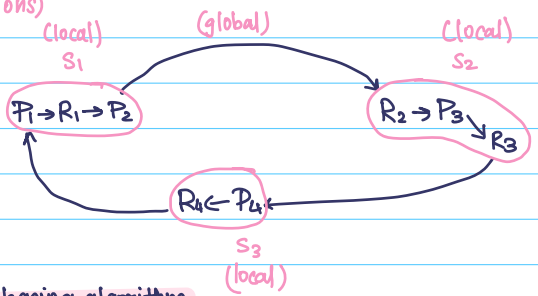
dedicated, centralised server to detect deadlocks.

↓

not easy

deadlock that might have existed in the past, but not right now → phantom deadlock

path pushing algorithm — initiator constructs a string (as transactions)



- phantom deadlock
- delay in communication channels
- limited BW earlier; computationally & data intensive transactions (to create a string)

t_1 → transaction (abstract concept; waiting)

no concatenation; 3 bit data only transfer.

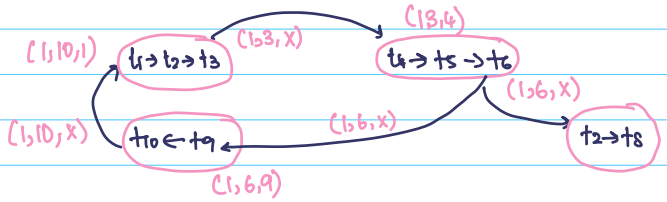
cannot be phantom

but, proof difficult

edge chasing algorithm

first check local deadlocks, then global deadlocks

send triplets along outgoing edge



→ all resources read to execute.

"AND" request model

t_3 holding resource, but also requests; deadlock.

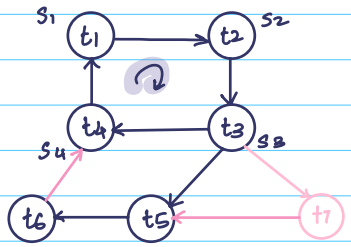
when i==k, i & k from same system → deadlock. "OR" request model
↓
any resource obtaining is enough

20/9/23

Knapp's classification

1. path pushing
2. Edge chasing (AND model)
3. diffusion computation (OR model)
4. Global state based

DIFFUSION COMPUTATION



with AND request model, the t3→t5→t6 execution will not continue.

with OR request model, it can continue when it gets some resources.

- t6 releases resource, t5 uses
- t3 won't get t4, but t5 is possible

(no deadlock, but starvation) — not always; leads to (always)

KNOT -

→ causes multiple loops/cycles

→ t3 waiting; all dependent resources are in cycles. t3 is a knot

- knot will have outgoing edge

∀ out edge leads to some cycle.

- no loose edges (OR request model)

we can detect deadlocks by detecting knots; all knots will have cycles.

- it is a subgraph, which will always contain a cycle.

probes are sent to detect cycles } probe algorithm but difficult in this context

when initiator receives probe, detects deadlock

knots - detects all the cycles

for probe algorithm, marginalize certain transactions

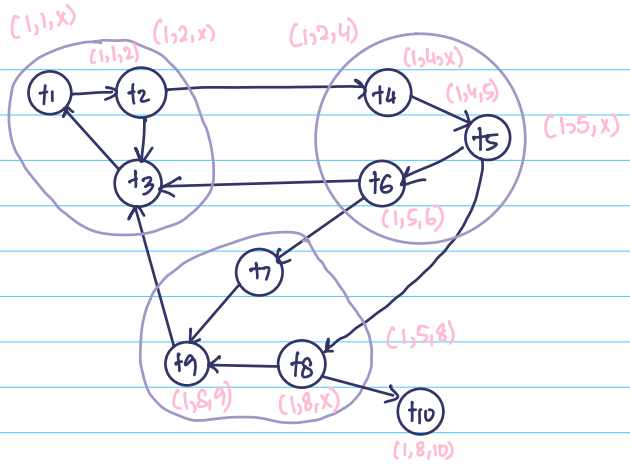
outgoing edge only present when waiting on resource.

probe is initiated

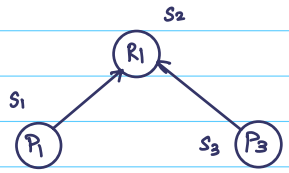
(if t is waiting, it generates reply → acts as response (denotes deadlock))

Receiver only replies if all outgoing edges reply; cannot reply if no outgoing edges

Chandy-Misra-Haas Algorithm



22/9/ **DISTRIBUTED MUTUAL EXCLUSION**



1) Non Token based (Queue)
2) Token based

Lamport's Algorithm

1) Request — know current req no (no req - 0); sequence no
   — know processID
2) Reply
3) Release

1) REQUEST

- places in own queue
- consistency forcefully maintained by broadcasting request

2) REPLY

- consent given
- tie breaker, in case of clash

3) RELEASE

- again broadcasted

resource < local / broadcasted

	P1	P2	P3
1	req (1,1)		
2		rec req (1,1)	rec req (1,1)
3		rep req (1,1)	rep req (1,1)
4	rec rep (1,1); P2		
5	rec rep (1,1); P3		
6	exec cs		
7	rel req (1,1)		
8		rec rel (1,1)	rec rel (1,1)
9		del req (1,1)	del req (1,1)
10			

	P1	P2	P3
1	req (1,1)		
2		rec req (1,1)	rec req (1,1)
3	doesn't reply bc of fairness principle	req (2,2)	rep req (1,1)
4	rec req (2,2)	rep req (1,1)	rec req (2,2)
5	rec rep (1,1) : P3		rep req (2,2)
6	rec rep (1,1) : P2	rec rep (2,2) : P3	
7	exec cs		
8	rel req (1,1)		
9	rep req (2,2)	rec rel req (1,1)	rec rel req (1,1)
10		del req (1,1)	del req (1,1)
		rec req (2,2) : P1	
		exec cs	

→ when both processes request @ the same time, starvation occurs; resource free, but nobody accesses.

message complexity: $3 * (n-1)$

since consent given, no explicit release msgs.

→ once replied, remove req from queue, w/o release request

25/9/23

RICART - AGARWALA'S ALGORITHM (D-MUTEX)

	P1	P2	P3
	~~(1,1)~~ | (2,2)	~~(1,1)~~ | (2,2)	~~(1,1)~~ | (2,2)
1	req (1,1)		
2		rec req (1,1)	rec req (1,1)
3		req (2,2)	rep req (1,1) → delete from queue
4	rec req (2,2)	rep req (1,1) → delete from queue	rec req (2,2)
5	rec rep (1,1) : P2		rep req (2,2)
6	rec rep (1,1) : P3	rec rep (2,2) : P3	
7	exec cs	↑ waiting for P1's consent	
8	rep req (2,2)	↓	
9		rec req (2,2) : P1	
10		exec cs	

CONFLICT

	P1	P2	P3
	(1,1) | (1,2)	X X (1,2) | (1,1)	(1,1) | (1,2)
1	req (1,1)	req (1,2)	
2	rec req (1,2)	rec req (1,1)	rep req (1,1)
3	rec rep (1,1) : P3	rep req (1,1)	rep req (1,2)
4	rec rep (1,1) : P2	↳ priority given to process ID	
5	exec cs		
6	rep req (1,2)		
7		rec rep (1,2) : P1	
8		exec cs	
9			
10			

requester can only delete after getting all replies.
Other systems can delete after replying.
} DOS Attacks prevented.

Expired requests: resend request after time-out.

Old msgs can be avoided by making sure only requests w a higher ack.

no. is executed.

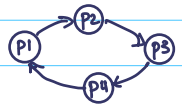
Message complexity reduced from

$3(N-1) \Rightarrow 2(N-1)$

26/8/2023

TOKEN BASED D-MUTEX (SUZUKI-KASAMI'S ALGORITHM)

LIMITATION: Mutex achieved; but system possessing token does not know who needs token - starvation



circular arrangement → performance issues.

∴ we need some way to know who needs the token, so it can be directly sent to them.

Data Structures

R - Request vector (at every node)

T - Token Vector (only one instance)

REQUEST

Inc Ri[i] & broadcast

RESPONSE

on receiving broadcast message,

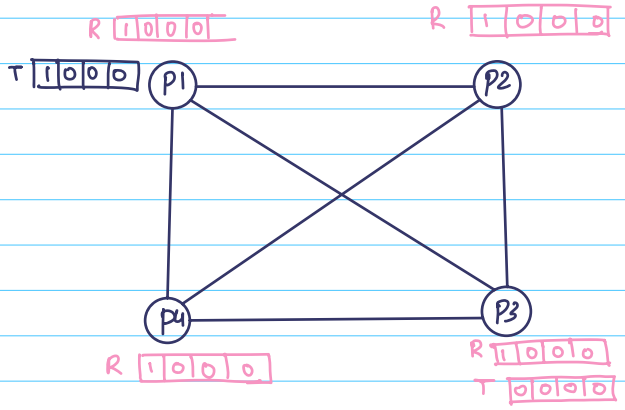
Rj[i] = max(Rj[i], Ri[i])

(every node)

if a node has T, then

if T[k] == Rj[i] - 1,

pass the token



NOTE:

To avoid outdated msgs, we need to take max value

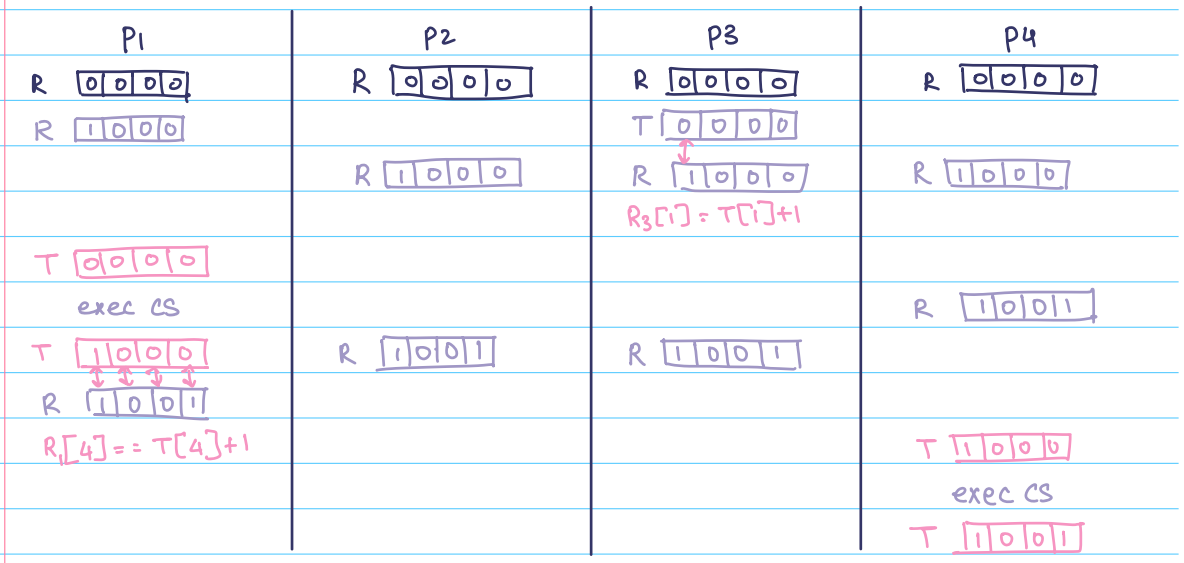
(k - all values; i - requester; j - receiver)

↓

updates sequence number & broadcast

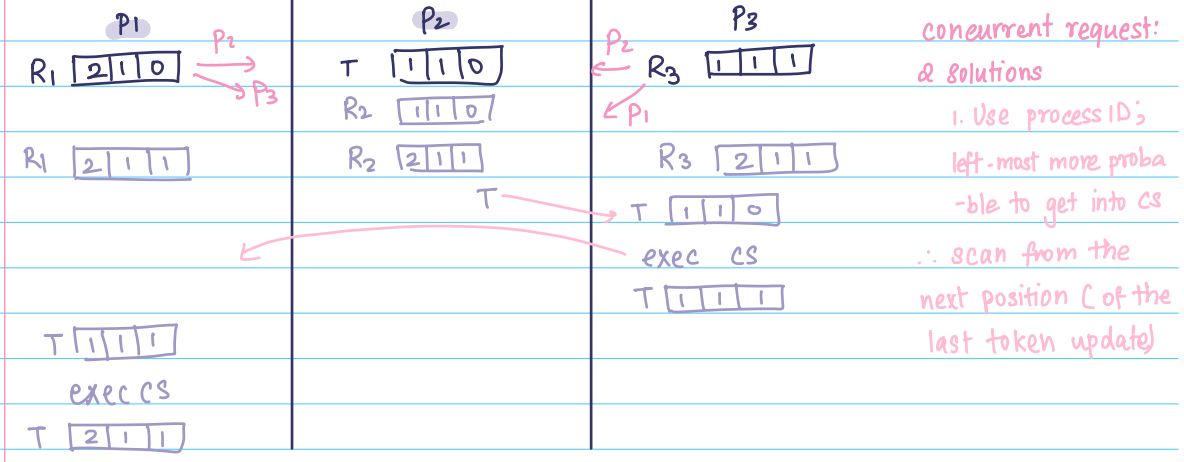
The deserving process is determined by checking which node has T[k] == Rj[i] - 1

In exam, expl ain your steps. Don't just write array.



for fairness: no concurrent requests by a processor; especially if there is an outstanding request

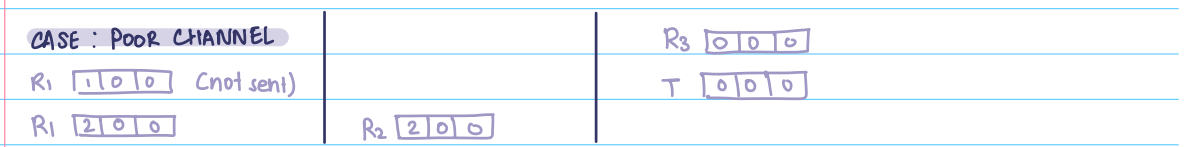
27/8/23



concurrent request: & solutions

1. Use process ID; left-most more proba -ble to get into CS

∴ scan from the next position C of the last token update)



process suspended.

if no candidate identified, increment difference & scan again

when outdated msg arrives, ignore / discard it.

DOS Attack prevention: cannot request concurrently before expiration of time period.