CAT - 2

Portion

Global State of a DS

↳ collection of local states of its components, ie, the process & communication channels

State of process → content of processor registers, stack & local memory

State of channel → set of msgs in transit in the channel

send(m) → affects state of sending process & channel

rec(m) → affects state of receiving process & channel

LS_i^x → local state of process P_i after occurrence of event e_i^x & before event e_i^{x+1}

$SC_{ij}^{x,y}$ → state of channel bw processes P_i & P_j consists of all msgs sent by P_i upto e_i^x & which process P_j had not rcvd until event e_j^y

$$GS = \{\cup_i LS_i^{x_i}, \cup_{j,k} SC_{j,k}^{y_j, z_k}\}$$

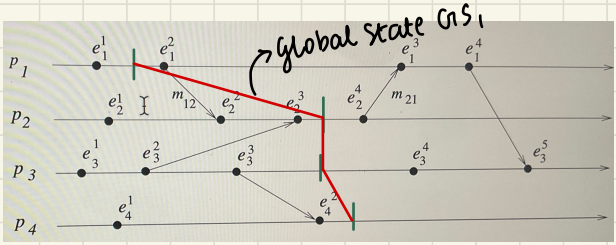
Consistent global states

↳ should follow causality

↳ state should not record a receive before/without its corresponding send

$GS = \{\cup_i LS_i^{x_i}, \cup_{j,k} SC_{j,k}^{y_j, z_k}\}$ is consistent iff

$$\forall m_{ij}: send(m_{ij}) \notin LS_i^{x_i} \Leftrightarrow (rcv(m_{ij}) \in LS_j^{y_j}) \wedge (m_{ij} \notin SC_{ij}^{x_i, y_j})$$



$GS_1 = \{LS_1^1, LS_2^3, LS_3^3, LS_4^2\}$

Inconsistent as: $send(m_{12}) \notin LS_1^2$ but $rec(m_{12}) \in LS_2^3$

ie, P1 has not recorded send of m12 but P2 has recorded rcv of m12 → causality violation

One of the 2 conditions for consistency

↳ $send(m_{ij}) \in LS_i \Rightarrow rcv(m_{ij}) \in LS_j \oplus \underbrace{m_{ij} \in SC_{ij}}_{\text{transit}}$

↳ $send(m_{ij}) \notin LS_i \Rightarrow m_{ij} \notin SC_{ij} \wedge rcv(m_{ij}) \notin LS_j$

Absence of shared memory leads to

↳ Inconsistency } eg: Bank withdrawal
wrt state

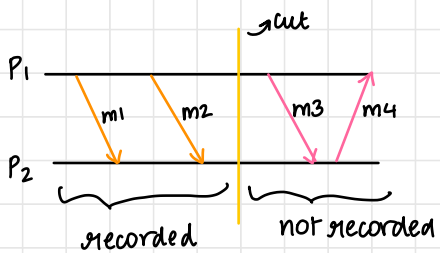
To solve this, we use global states → snapshot of local states

$$\text{Global state} = \bigcup_{i}^{n} (\text{local state})_i$$

Global state recorded using snapshot → only local states up till ss is recorded in Global state,
other info abt actions are stored in Local state until next ss

Snapshot is a cut : split time-space graph into 2

eg :



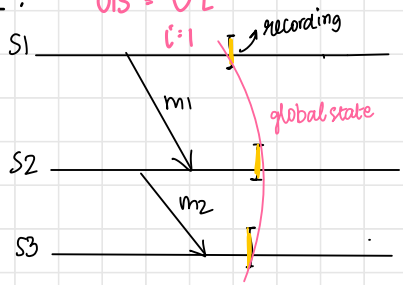
Local states wrt Cut : LS(P1) = {send(m1), send(m2)}
LS(P2) = {rcv(m1), rcv(m2)}

send(m3), rcv(m3) & send(m4), rcv(m4) are not recorded in global state

Global States

Store states to ensure consistency & for rollback

eg 2: $GS = \bigcup_{i=1}^{n} L$



LS_1 → local state = {send (m_1)}
LS_2 = { send(m_2), receive(m_1)}
LS_3 = {receive(m_2)}

} Strongly consistent

Consistency

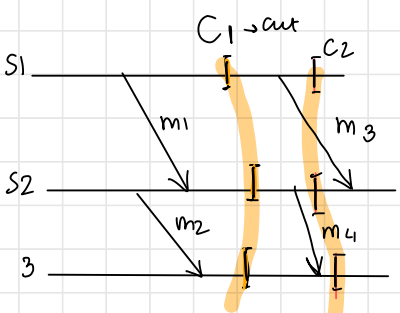
Is global state consistent?

↳ If every send has rcv ⇒ strongly consistent

How to implement strong consistency?

↳ Not practically possible as stopping to record affects performance.

Hence, we strive for only consistency

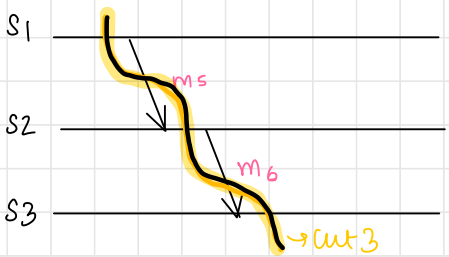


for C2

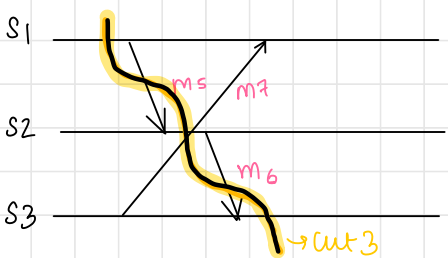
LS_1 = {send (m_3)}
LS_2 = {send (m_4)}
LS_3 = {recv (m_4)}

receive (m_3)? is in transit
⇒ not strongly consistent

Consistent global state: each receive should have corresponding send
↳ has msg in transit (only send recorded)

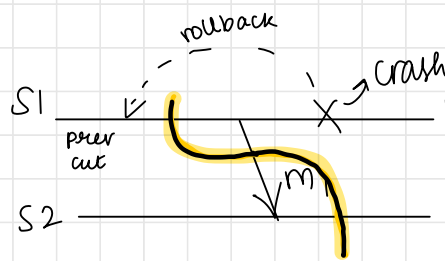


$LS_1 = \{-\}$
$LS_2 = \{rcv(m5)\}$
$LS_3 = \{rcv(m6)\}$
} no corresponding send.
⇒ inconsistent state
ie, rcv recorded before send
logically incorrect - problem1

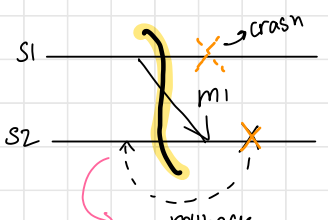


$LS_1 = \{-\}$ → inconsistency
$LS_2 = \{rcv(m5)\}$
$LS_3 = \{send(m7), rcv(m6)\}$
↓ in transit

NOTE: Even if 1 instance of inconsistency ⇒ Global state inconsistent
ie, inconsistency holds priority

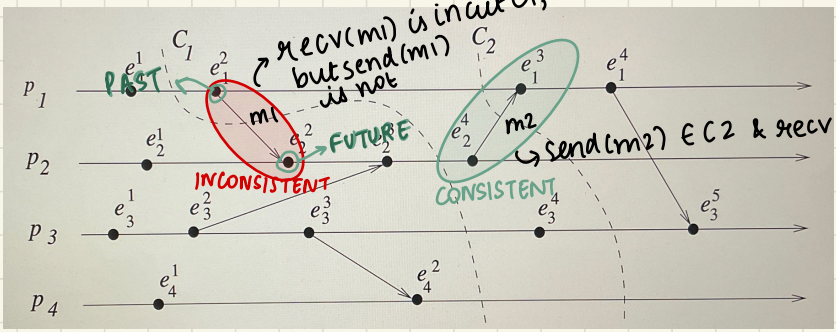


But w/ rollback, no record of sending m1 } problem 2



Cuts → split space-time diagrams & thus the events into PAST & FUTURE
(PAST ↙ left of cut; FUTURE ↘ right of cut)

Every cut corresponds to a global state



NOTE: A cut is consistent if all msgs that cross the cut from the PAST to FUTURE are in transit in corresponding frs

In C_1, $e_1^2 \rightarrow$ Past of C_1 but $e_2^2 \rightarrow$ FUTURE $\in C_1$

Past Cone of an Event

An event e_j is only affected by events in its past ie, $e_i \rightarrow e_j$

$$Past(e_j) = \{ e_i \mid \forall e_i \in H \wedge e_i \rightarrow e_j \}$$

Let $Past_i(e_j)$ be set of events in $Past(e_j)$ that are on Process P_i

$Past_i(e_j)$ is a totally ordered set by relation $\rightarrow_i$

$max(Past_i(e_j)) \rightarrow$ last event on P_i to affect e_j

$$max\ past(e_j) = \cup_{\forall i} (max(Past_i(e_j)) \rightarrow$$ set of last events on every process to affect e_j

Future Cones

Future of event e_j is set of all events e_i that are causally affected by e_j, ie $e_j \rightarrow e_i$

$$Future(e_j) = \{ e_i \mid \forall e_i \in H, e_j \rightarrow e_i \}$$

All event after max past(ej) & before min future(ej) are concurrent with ej ie, H - Past - fut.

$min(Future_i(e_j)) \rightarrow$ first event on P_i to be affected by e_j

$min(Future(e_j)) = \cup_{\forall i} (min(Future_i(e_j))) \rightarrow$ set of all first events on processes to be affected by e_j

Models of Process Communication

↳ Synchronous → blocking type; sender sends msg & blocks until receiver rcvs msg
→ sender resumes execution only after it learns abt rcv

↳ Asynchronous → non-blocking type; sender & rcvr dont synchronize, ie sender doesn't wait after sending msg
→ msg is buffered by system until rcvr is ready to rcv.

Asynch → needs more complex buffer mgmt but provides higher degree of parallelism. Difficult to design & implement. Better performance

---

Message in Transit : transit (LS_i, LS_j) [msg in transit b/w P_i & P_j on channel SC_{ij}]
$= \{m_{ij} \mid send(m_{ij}) \in LS_i \wedge rec(m_{ij} \notin LS_j\}$

FIFO model → each channel acts as FIFO message queue, ie, order preserved
NON FIFO model → channel acts as set in which sender adds msgs & rcvr removes msgs from it randomly

For causality,

if $send(m_{ij}) \rightarrow send(m_{kj})$ then
$rcv(m_{ij}) \rightarrow rcv(m_{kj})$

Chandy Lamport Algorithm

↳ uses control msg, "marker" whose role in a FIFO system is to separate msgs in channel

↳ after a node takes SS, send marker to all outgoing channels before next msg

↳ Marker separates msgs in channel into those to be and not included in SS

↳ 2 rules

Marker Sending Rule for process i

1. Process i records its state.
2. For each outgoing channel C on which a marker has not been sent, i sends a marker along C before i sends further messages along C.

Marker Receiving Rule for process j

On receiving a marker along channel C:

if j has not recorded its state **then**
Record the state of C as the empty set
Follow the "Marker Sending Rule"

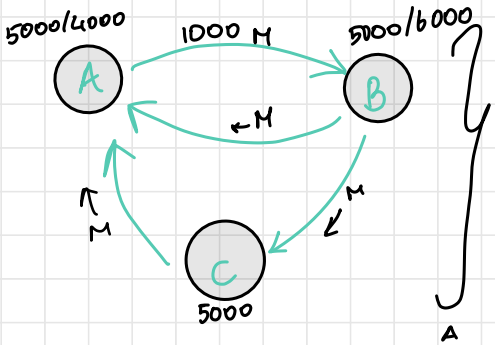
else
Record the state of C as the set of messages received along C after j's state was recorded and before j received the marker along C

↳ Algorithm terminates after each process received a marker on all of its incoming channels

Global Snapshot Recording Algorithm / Chandy Lamport

Starting process records state

M → when rcv marker, snapshot state & pass marker to outgoing channel



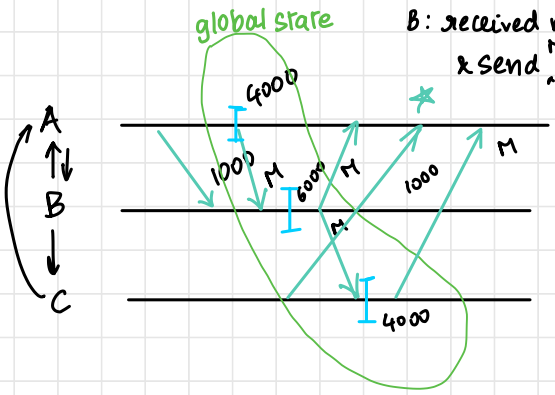
sender ss → snapshot & broadcast M

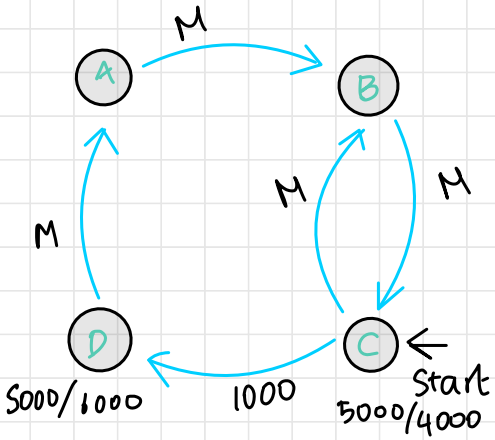
B: received marker, check if msg waiting? Yes, 1000. so integrate & send M all ss not been taken before

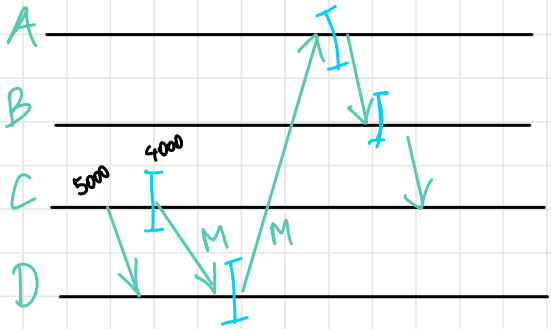
at A: ss already taken but msg waiting, record in local state & integrate in next global state

at C: rcv M from B, no msg waiting so take ss & send M to A

at A: rcv M from both channels, so end







Any node can start Algorithm.

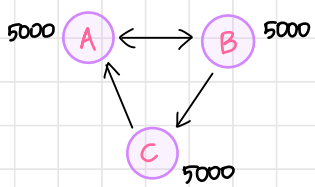
On send: • if msg on channel, integrate
(initiator) • send marker M through all outgoing channels

On receive: • record states
• follow send rule

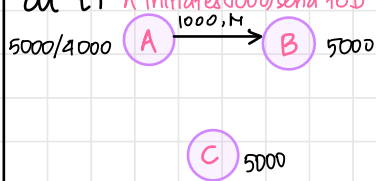
For ending algorithm: if initiator receives marker from all incoming channels

Step-by-step example

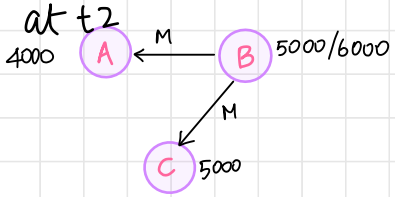
at t0:



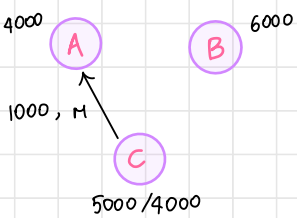
at t1: A initiates (1000) send to B



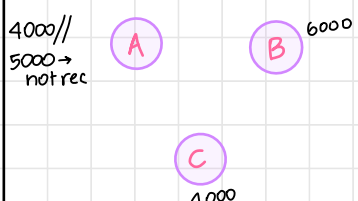
at t2



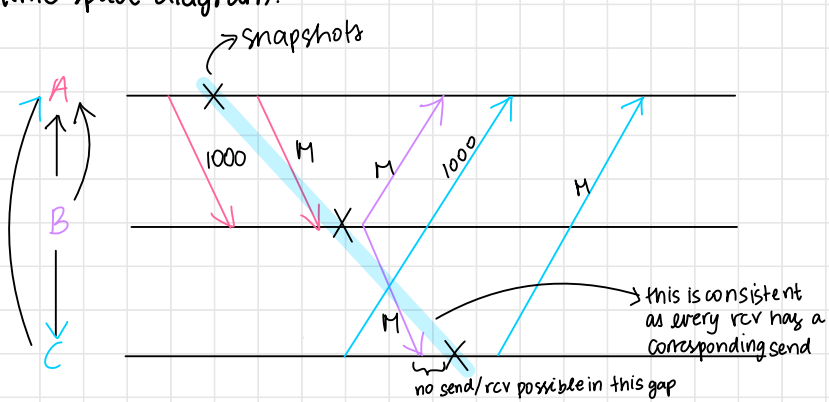
at t3



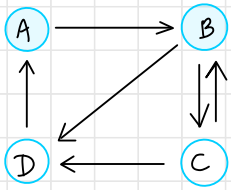
at t4



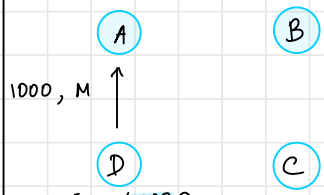
time-space diagram:



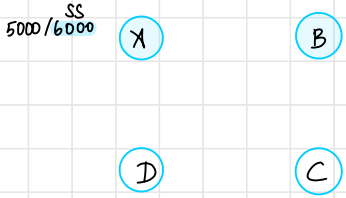
Example:



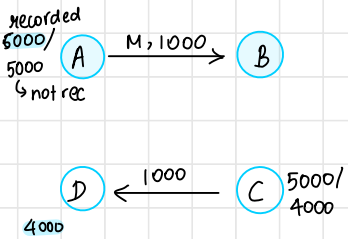
at t1



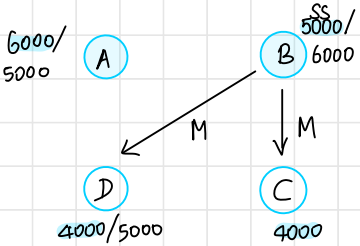
at t2 : SS taken by A



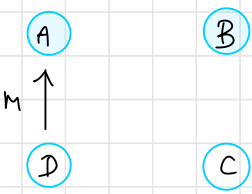
at t3 :



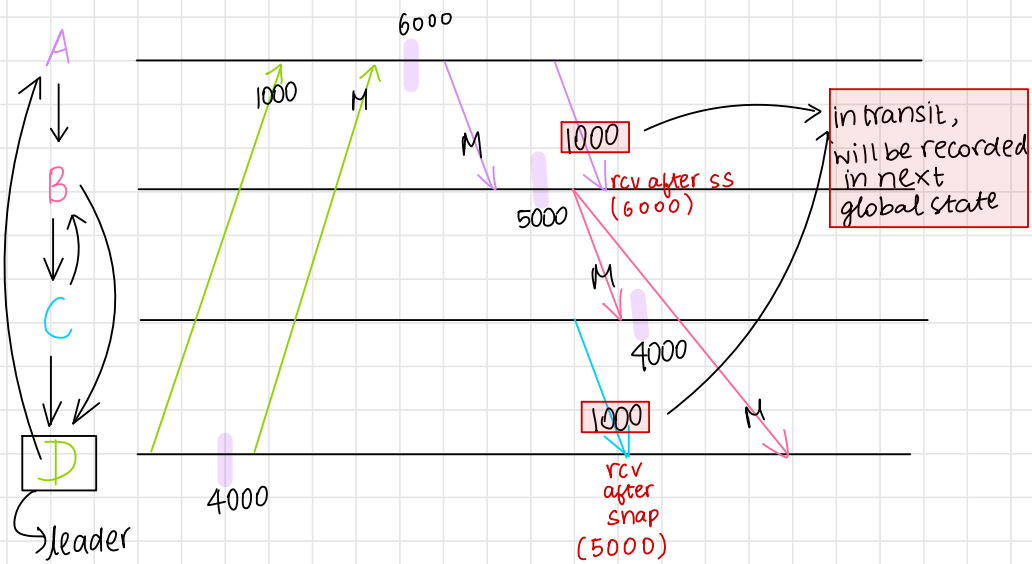
at t4



at t5

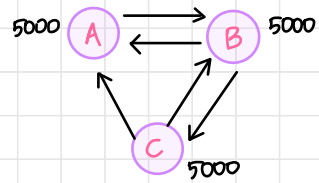


A rcvs marker from all incoming channels

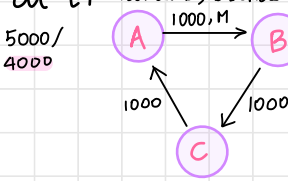


Example

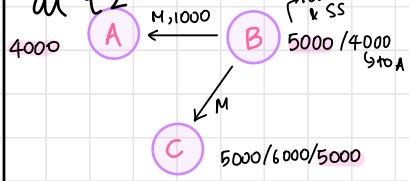
at t0:



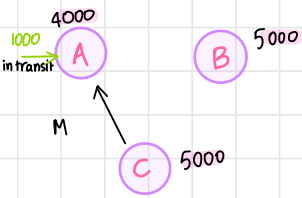
at t1: A sends B, B sends C, C sends A

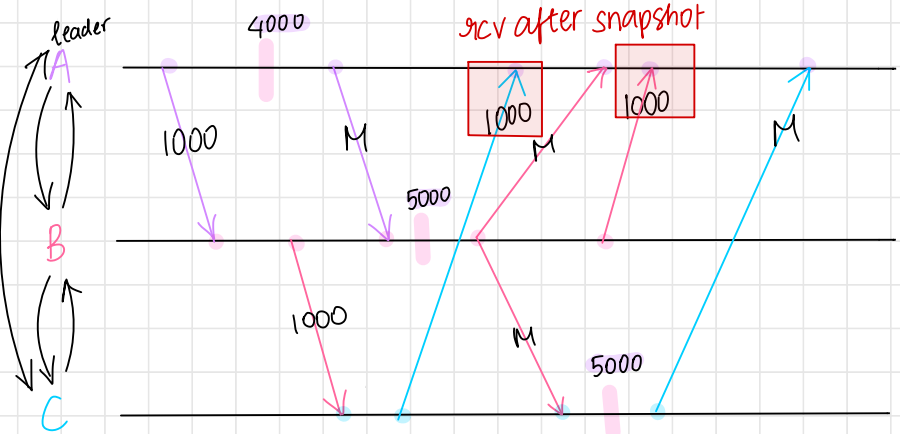


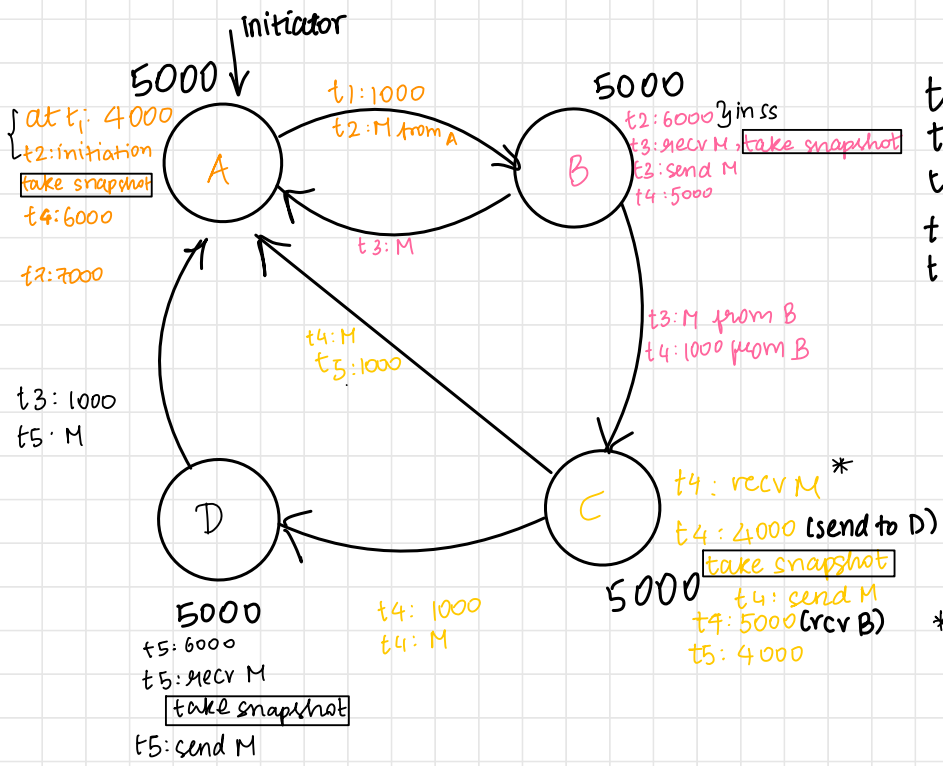
at t2

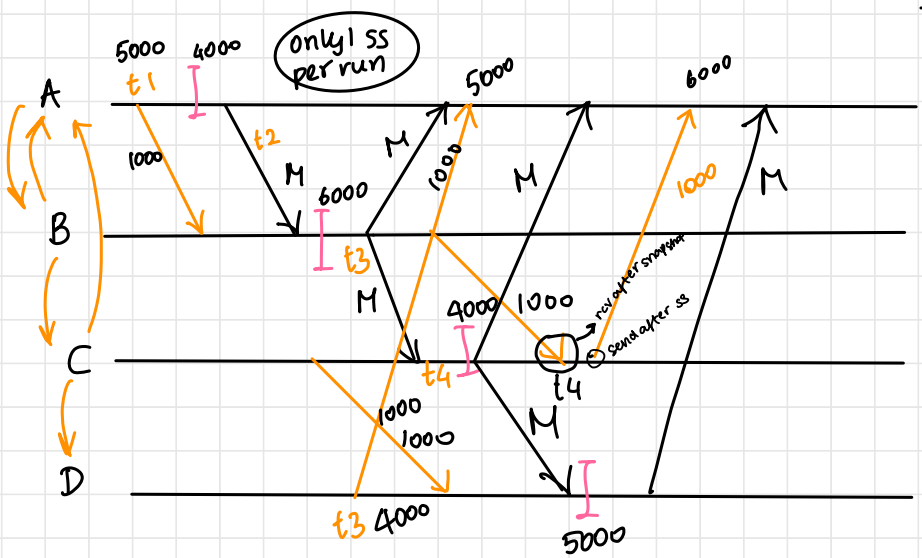


at final time

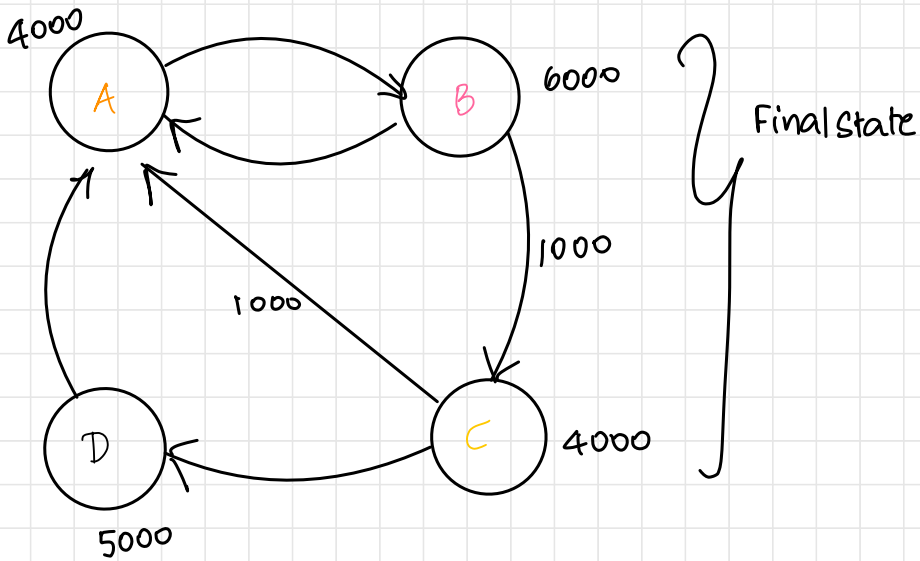


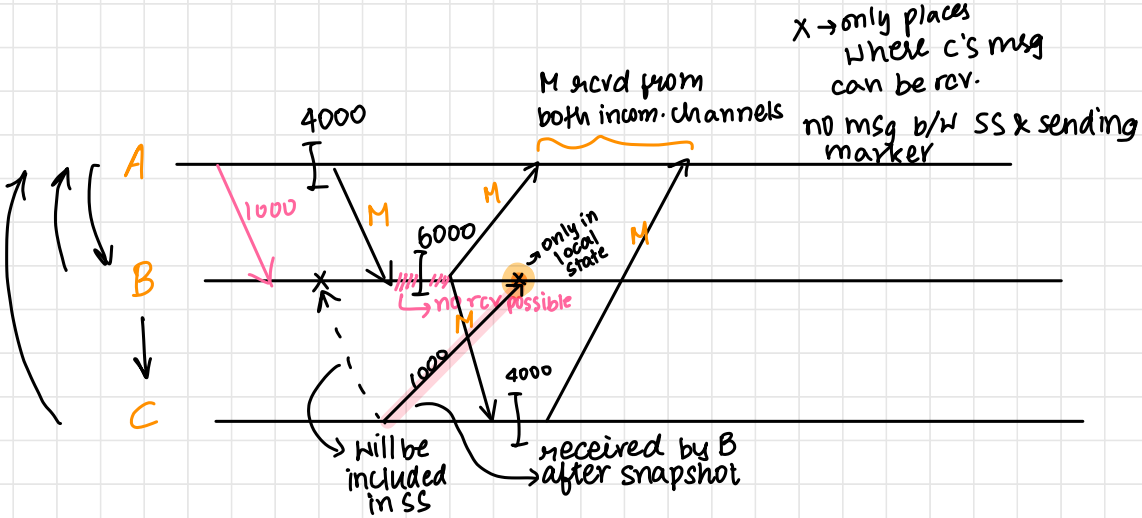






initiator has to rcv marker msgs from all incoming channels.

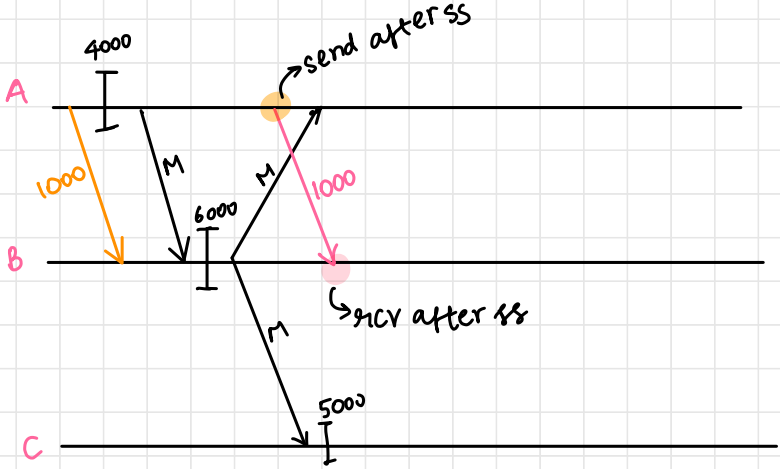




final states : A → 4000

B → 6000 .channel → 1000

C → 4000



A → 4000 , channel → −1000
B → 6000 , channel → 1000
C → 5000

By rec Global state : ↳ deadlocks prevented

Unit-3

Mutual Exclusion: Concurrent access to a shared resource by a process is done in a mutually exclusive manner
ie, only 1 process is allowed to execute the CS at given time

In a distributed system, shared variables (semaphores)/local kernel can't be used to implement mutex

Message passing is the sole means to implement DMutex

System Model

System has N sites: $S_1, S_2 \dots S_N$
We assume that 1 process (P_i) is running at each site (S_i)
↳ A site can be in
- ↳ idle: executing outside CS
- ↳ in CS
- ↳ requesting for CS: site is blocked, no further req for CS allowed
  So, even if site has multiple req for CS, it queues them up & serves them 1 at a time

---

Requirements for Mutex Algorithms

- ↳ Safety Property: at any instant, only 1 process can execute CS
- ↳ Liveness Property: absence of deadlocks & starvation, ie, no site should wait endlessly
- ↳ Fairness Property: every process gets fair chance to execute CS, ie, CS req served in the order they arrive

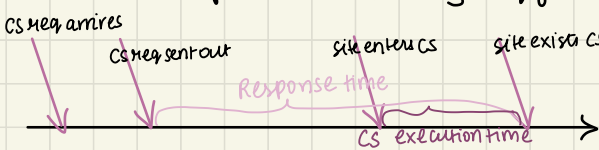
---

Performance Metrics

i) **Message Complexity:** no of msgs required per CS execution by a site

ii) **Synchronization delay:** time taken b/w 1 site leaving CS & the next site entering CS

iii) **Response Time:** time taken from a site sending req for CS & completing its CS



iv) System Throughput: rate at which system executes requests for CS

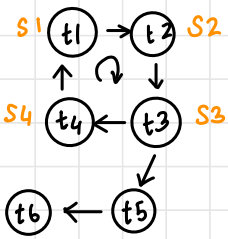
$$= \frac{1}{\text{Synchronization delay} + \text{execution delay}} = \frac{1}{SD + E}$$

↳ Loads
- low load → rarely >1 concurrent req for CS
- high load → always a pending req for CS at a site

Knapp's Classification

i) Path Pushing
ii) Edge Chasing (AND deadlocks)
iii) Diffusion Computation (OR deadlocks)
iv) global state based - GSRA algorithm

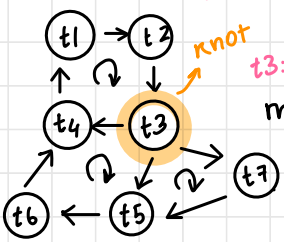
OR based deadlocks - can request for many resources but can proceed when it rcvs one of it



although cycle present, no deadlock.
as t3 rcvs from t5
t6 releases resource → used by t5 & released → given to t3

} min. starvation occurs

What is a knot?



t3: needs t4 or t5 or t7
multiple cycles are present

If a node is a knot then

↳ All outgoing edges, must lead to a cycle
ie, no lose edges
↳ It is a subgraph within same graph - all edges lead back to same node

Not all cycles have knots but all knots lead to cycles

AND request model cycle for Deadlocks
OR request model knot for deadlocks

Lamports Algorithm

↳ requests for CS are executed in increasing order of timestamps
↳ Every site has a request queue to hold mutex req ordered by timestamps
↳ Communication channels deliver msgs in FIFO order

3 Phases of Algorithm

↳ Request Phase:

- When site S_i wants to enter CS, broadcast request(seqno, process id) & places it on its request queue (ts_i, i)

- When site S_j receives request(ts_i, i), it places it on its req queue & returns reply(ts_i, i) → Reply Phase

↳ Executing Critical section

Site S_i enters CS when the following conditions hold

- S_i's request is at the top of the queue → L2
- S_i rcvd reply messages from all other sites → L1

↳ Release Phase

- After S_i finishes execution, it deletes request(ts_i, i) from queue & broadcasts a release msg to all other sites
- When S_j recvs release(ts_i, i) from S_i; S_j removes req(ts_i, i) from its queue

---

Theorem: Lamport's algorithm achieves mutual exclusion.
Proof:

- Proof is by contradiction. Suppose two sites S_i and S_j are executing the CS concurrently. For this to happen conditions L1 and L2 must hold at both the sites *concurrently*.
- This implies that at some instant in time, say t, both S_i and S_j have their own requests at the top of their *request_queues* and condition L1 holds at them. Without loss of generality, assume that S_i's request has smaller timestamp than the request of S_j.
- From condition L1 and FIFO property of the communication channels, it is clear that at instant t the request of S_i must be present in $request\_queue_j$ when S_j was executing its CS. This implies that S_j's own request is at the top of its own *request_queue* when a smaller timestamp request, S_i's request, is present in the $request\_queue_j$ – a contradiction!

Theorem: Lamport's algorithm is fair.
Proof:

- The proof is by contradiction. Suppose a site S_i's request has a smaller timestamp than the request of another site S_j and S_j is able to execute the CS before S_i.
- For S_j to execute the CS, it has to satisfy the conditions L1 and L2. This implies that at some instant in time say t, S_j has its own request at the top of its queue and it has also received a message with timestamp larger than the timestamp of its request from all other sites.
- But *request_queue* at a site is ordered by timestamp, and according to our assumption S_i has lower timestamp. So S_i's request must be placed ahead of the S_j's request in the $request\_queue_j$. This is a contradiction!

Lamports requires $3(N-1)$ msgs per CS invocation

- In Lamport's algorithm,REPLY messages can be omitted in certain situations. For example, if site S_j receives a REQUEST message from site S_i after it has sent its own REQUEST message with timestamp higher than the timestamp of site S_i's request, then site S_j need not send a REPLY message to site S_i.
- This is because when site S_i receives site S_j's request with timestamp higher than its own, it can conclude that site S_j does not have any smaller timestamp request which is still pending.
- With this optimization, Lamport's algorithm requires between $3(N-1)$ and $2(N-1)$ messages per CS execution.

Distributed Mutual Exclusion

In DS, states are inconsistent

Fairness & liveliness are required in a DS

↳ To solve < Token based algo (eg: Ring Topology)
Non-token based algo

---

i) **Token based**: arrange nodes cyclic order. Pass token once used to next node

ii) **NonToken based**: request is broadcasted

↳ Node which possesses resource should ID who needs resource next

↳ Same resource shouldnt be given

Lamports Algorithm

3 Phases → i) **Request** - node wants access to critical section. Should know which req (seq. no) is being served needs process id & seq no (current req+1). Place on self queue, queue manager broadcasts

ii) **Reply** - everyone replies (consent) back to req in order of recv. But if some node objects, replies immediately

iii) **Release** - release critical section & broadcast to all nodes

---

(1,1)* at release

(1,1) at release req from P1

(1,1) at release from P1

time	P1	P2	P3
1:	Req(1, 1) (PID, Seq no)	broadcast	→ Phase 1 Request
2:		Rcv Req(1,1)	Rcv Req(1,1)
3:		Reply Req(1,1) has no objection	Reply Req(1,1)
4:	Reply rcvd (1,1) from P2		→ Phase 2 Reply
5:	(1,1) from P3		
6:	execute critical section		
7:	Release Req(1,1) & delete req(1,1)		→ Phase 3 release
8:		Recv rel(1,1)	Recv rel(1,1)
9:		delete req(1,1)	del req(1,1)

Case 2

P1	P2	P3
~~(1,1)~~ | (2,2) |	(1,1) | (2,2) |	(1,1) | (2,2) |
req (1,1)		
	rcv req (1,1)	rcv req (1,1)
	req (2,2) Concurrently 2 req CS	rep req (1,1)
rcv req (2,2) → No reply	rep req (1,1) Yfairness as P1 req first	rcv req (2,2)
rcv rep(1,1) from P3		rep req (2,2)
rcv rep (1,1) from P2	rcv rep(2,2) from P3	
Exec. Critical Section		
del req(1,1) & release		
rep req (2,2)	rcv rel. req(1,1)	rcv rel req (1,1)
	del req(1,1)	del req (1,1)
	rcv rep (2,2) from P1	
	Exec. Critical Section	
	del req (2,2) & release	
rcv rel req (2,2)		rcv rel req (2,2)
del req (2,2)		del req (2,2)

Case 3

P1	P2	P3
(1,1) | (1,2)	(1,2) | (1,1) forced to shuffle	(1,1) | (1,2)
req (1,1)	req (1,2)	
rcv req (1,2) won't reply as P1 req at top of its queue	rcv req (1,1) no reply as req(1,2) at front of queue	rcv req (1,1) rcv req (1,2)
		rep req (1,1) rep req (1,2)
rcv rep (1,1) from P3	rcv rep (1,2) from P3	
	Starvation So order according to Proces ID P1 > P2 > P3 rep req (1,1)	
rcv rep (1,1) from P2 Exec. critical Section Release req (1,1)		
	rcv rel (1,1) del req (1,1)	rcv rel (1,1) del req (1,1)
rep req (1,2)		
	rcv rep (1,2) from P1 Enter critical section & continue w. release & broadcast	

NOTE: To ensure fairness, only one outstanding req allowed per process

If comm. problem, req (1,1) $\not\rightarrow$ P3 (doesn't reach)

after timeout, req (2,1) again by P1

if after re-request (2,1), (seq no+1) req (1,1) reaches P3 $\Rightarrow$ discard if rcvd req (seq) no < prev req (seq) no

for 1 cycle: $3(n-1)$ msgs needed $\rightarrow$ msg complexity

Ricart Agarwala Algorithm

↳ assumes communication channels are FIFO

↳ uses 2 types of messages Request & Reply (same as in Lamport)

↳ request are served in order of timestamp

↳ each process has a Request-Deferred Array, RD

for Process P_i & P_j

$\forall i, \forall j$ $RD_i[j] = 0$ initially

if P_i defers request from P_j

$RD_i[j] = 1$

after P_i replies to req from P_j

$RD_i[j] = 0$

Phases

i) Request

- When site S_i wants to enter CS, it broadcasts $req(ts_i, i)$ to all sites
- If a site S_j rcvs req from S_i,

  if it is idle / req CS with $ts_j > ts_i$ ⇒ reply to S_i

  else set $RD_j[i] = 1$

ii) Execute CS

- Site S_i enters CS after rcving reply from all other sites

iii) Release CS

- When S_i exits CS, send reply to all deferred msgs

  ie, if $RD_i[j] = 1$, reply to S_j & set $RD_i[j] = 0$

Theorem: Ricart-Agrawala algorithm achieves mutual exclusion.

Proof:

- Proof is by contradiction. Suppose two sites S_i and S_j are executing the CS concurrently and S_i's request has higher priority than the request of S_j. Clearly, S_i received S_j's request after it has made its own request.
- Thus, S_j can concurrently execute the CS with S_i only if S_i returns a REPLY to S_j (in response to S_j's request) before S_i exits the CS.
- However, this is impossible because S_j's request has lower priority. Therefore, Ricart-Agrawala algorithm achieves mutual exclusion.

Requires $2(N-1)$ msgs per CS execution

Optimization: if P_i got Reply from P_j for prior CS access, P_i doesnt need to send requests to P_j for further CS access until P_j requests CS access

reduces no of msg to $[0 - 2(N-1)]$

Ricart - Agarwala's algorithm (D-mutex)

↳ reduces msg complexity to 2(n-1)

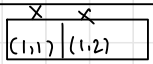
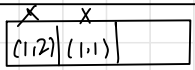
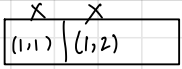
NOTE: In CSE-B ppts, instead of storing req in queue, store 1 if req is deferred, else zero.

Queue size = no of process (fixed)

P1	P2	P3
(1,1) | (2,2) X X	(1,1) | (2,2) X	(1,1) | (2,2) X X
req (1,1)		
	rcv req (1,1) req (2,2)	rcv req (1,1) rep req (1,1)
rcv req (2,2) rcv rep (1,1) from P3	rep req (2,2)	rcv rq (2,2) rep req (2,2)
rcv rep (1,1) from P2	rcv rep (2,2) from P3	
Enter critical section del Req (1,1) & send rel (1,1) to P2 & P3	Lamport's Algorithm	} ignore for this Algo
now, immediately after responses from P2 & P3, delete (1,1) from queue. after critical section, send response to P2 & delete (2,2) from queue	delete (1,1) after sending response to P1 This ensures no release msg is needed	delete (1,1) & (2,2) after responding to P1 & P2

since (1,1) is at top of queue, response given to P1

NOTE:
Self requests are deleted after executing critical sections
Other requests are deleted after sending response to it

P1	P2	P3
		
req (1,1)	req (1,2)	
recv req (1,2)	recv req (1,1)	recv req (1,1) recv req (1,2) rep req (1,1) & delete rep req (1,2) & delete
rcv resp (1,1) from P3	rcv resp (1,2) from P3 process id takes priority ⇒ rep req (1,1) & delete	
rcv resp (1,1) from P2 critical section delete (1,1) from queue	(1,1) from queue	
rep req (1,2) & delete (1,2) from queue	rcv resp (1,2) from P1 Enter critical section delete (2,2) from queue	

To ensure fairness,

unlike in lampost, req (on receiver side) deleted immediately after response

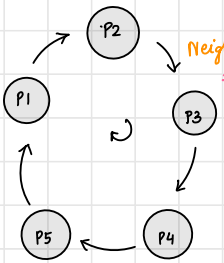
⇒ fairness may be violated?

To ensure fairness, req not deleted from sender until after execution

Token based D-Mutex (Suzuki - Kasami's algorithms) (CSE-B ppt refer)

↳ Arrange nodes in circular order

resources passed around sequentially



Neighbour nodes are predefined
⇒ if P1 has token P5 needs,
it still has to pass through all nodes b4 P5 gets it
node w. token doesn't even know who needs token

bad mutex mgmt;
leads to starvation
↳ resource free, but not able to be used by reqd node

Problem: Starvation; know demands & pass token to that node

Data structures : i) request vector, R at each node (size → n (n: total no of nodes)

ii) token vector, T (only 1 copy / instance; available at node which last used token)

Request

Inc $R_i[i]$ (i^{th} value of requesting node's R)

broadcast R_i

Response

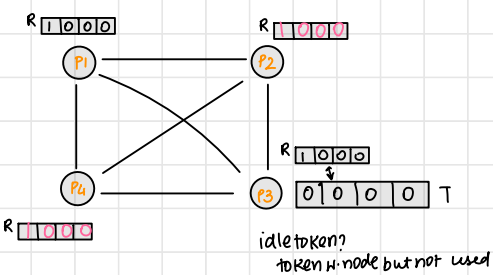
on rcv broadcasted msg.

$R_j[i] = \max(R_j[i], R_i[i])$ (update i^{th} place of receiver node's R vector)

if node has T, then

↳ if $T[k] == R_j[i] - 1$, pass the token

↳ explanation:



idle token?
token w. node but not used

i) P1 wants token

$R_1 = 1000$ & broadcast

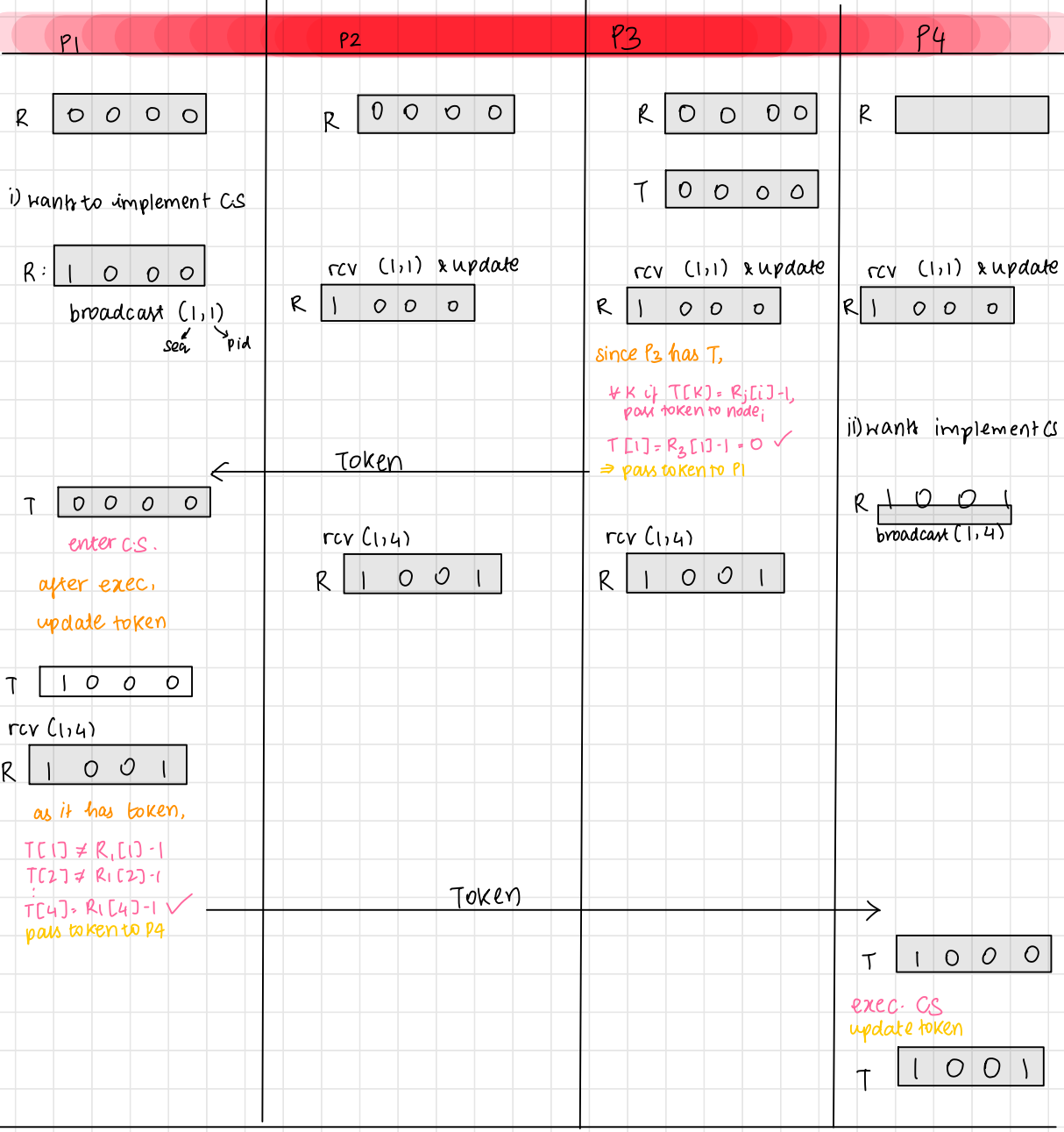
⇒ $R_2 = R_4 = 1000$ as $\max(R_j[i], R_i[i])$

↳ Why max?
- rcvd req may be outdated as a req may have been resent. To ignore old req, take max

for P_3, (with T)

$\forall k$, $T[k] == R_3[i] - 1$ (i: i^{th} value, j: j^{th} node)

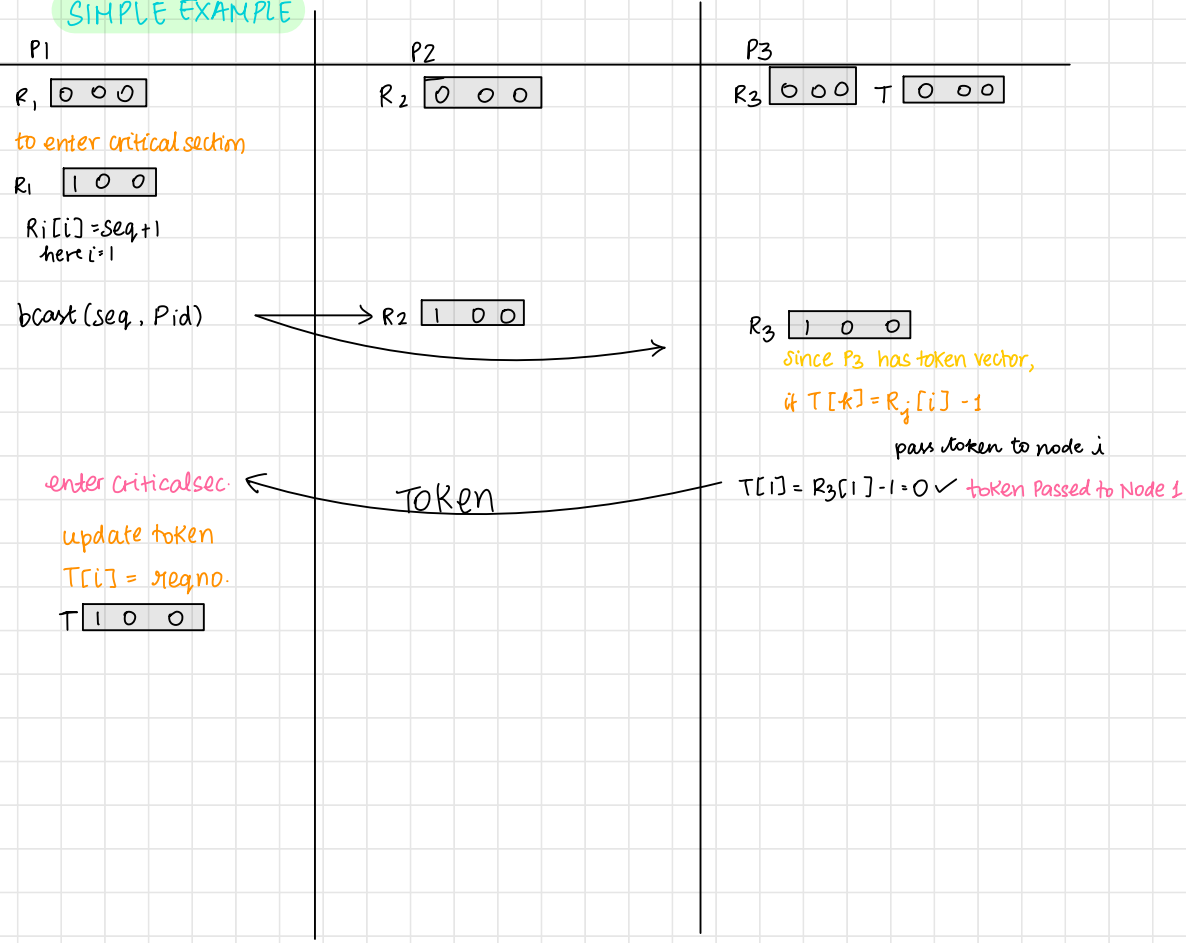
for N1, condition satisfied,
so pass token to N1
scan token L to R, first value to satisfy, pass token to that node.



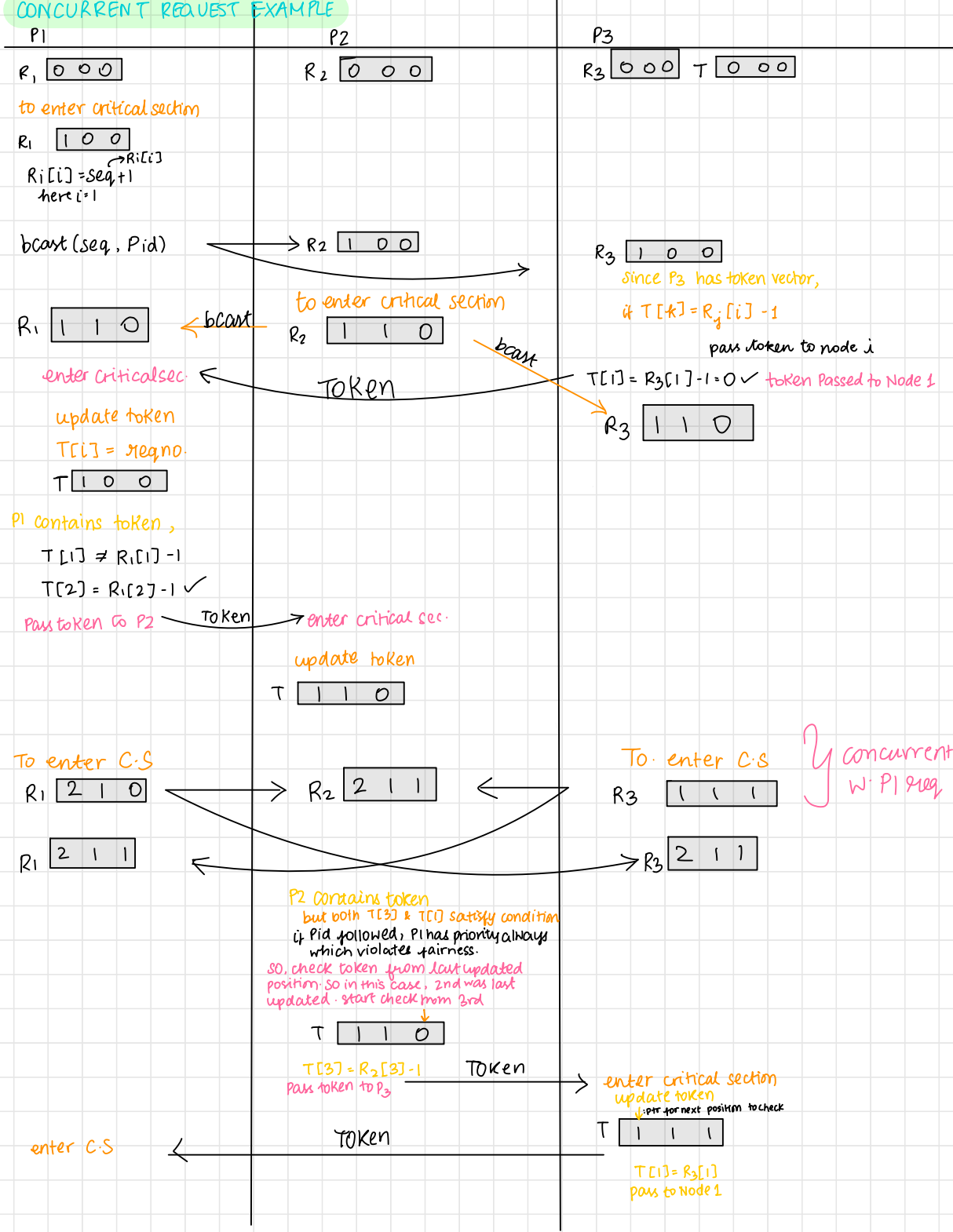
How to ensure fairness (no node requests < 1 times)

↳ maintain copy of local req.

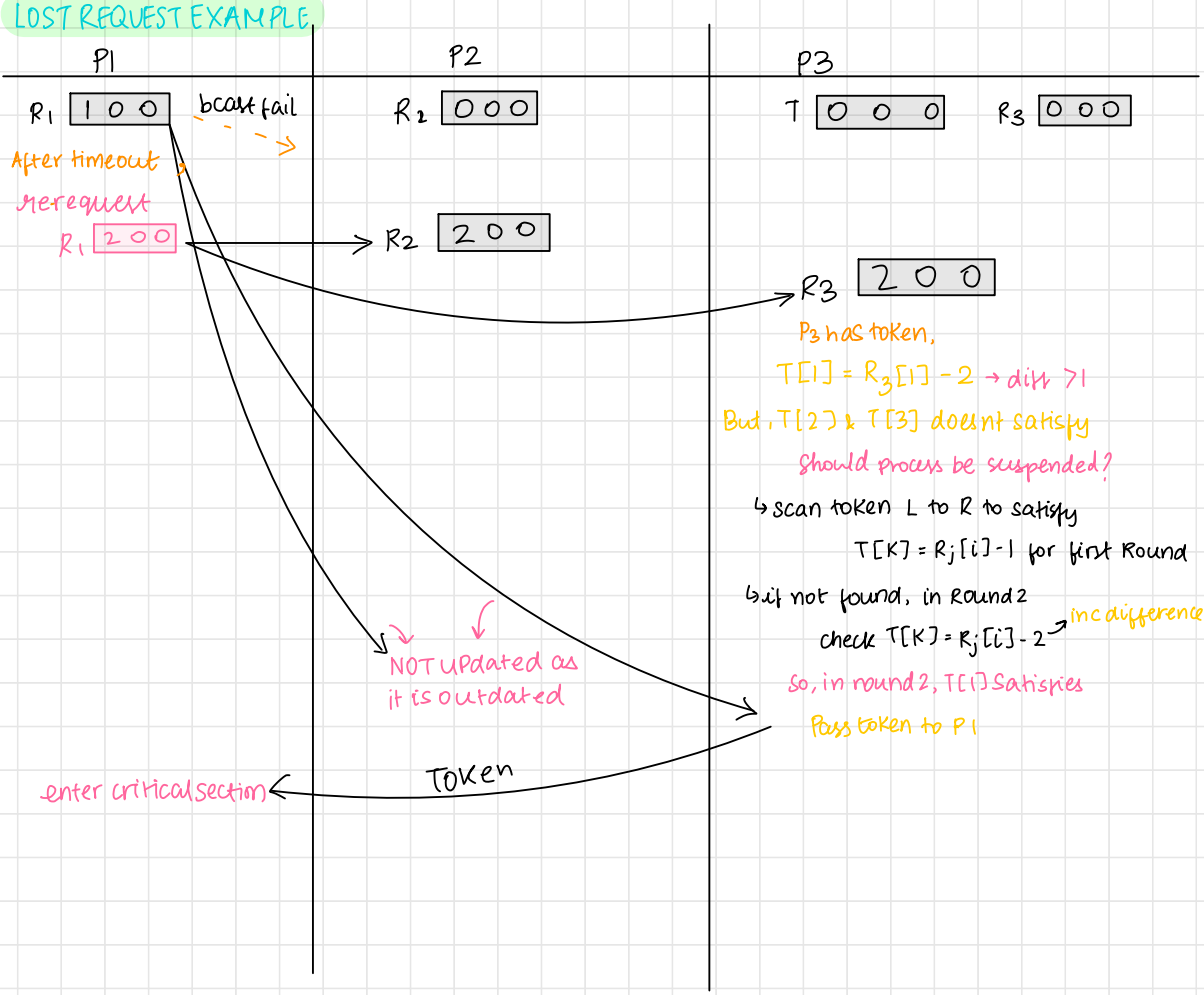
SIMPLE EXAMPLE



CONCURRENT REQUEST EXAMPLE

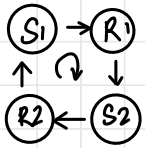


LOST REQUEST EXAMPLE



Distributed Deadlocks

Deadlock example:

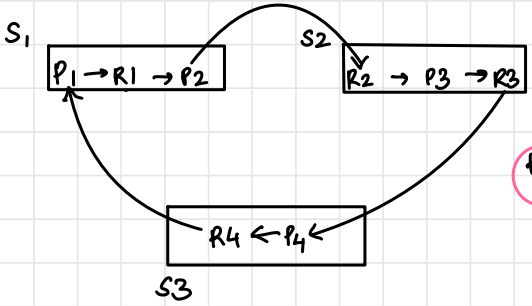


↳ central leader not suitable for DS
any node can initiate a task

Phantom Deadlock: deadlock doesn't exist currently but DS mistakenly assumes or vice versa

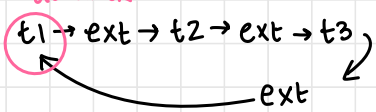
Occurs when there is delay in comm. channel

Path pushing algorithm



transaction: abstract word to imply waiting

deadlock



Deadlock Detection in DS

Deadlock: a state where a set of processes request resources that are held by other processes in the set

Deadlocks occur if sequence of resource allocations aren't controlled

If system with async processes $P_1, P_2 \cdots P_2$ (each on diff processor) communicate, since there is no global clock, processes have instantaneous access

Assumptions

- ↳ All resources are reusable
- ↳ Processes are allowed only exclusive access to resources
- ↳ Only one copy of each resource

States of a process

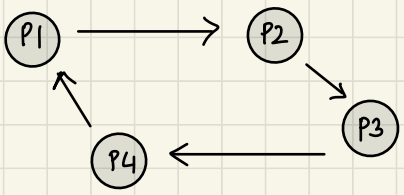
i) Running → has all reqd resources & is either executing / ready
ii) Blocked → waiting for resource

Wait For Graph (WFG)

↳ nodes are processes
If there is a directed edge b/w nodes P_1 & P_2
$\Rightarrow$ P_1 is blocked & is waiting for a resource that P_2 holds

deadlocks from WFG → if there is a cycle / knot

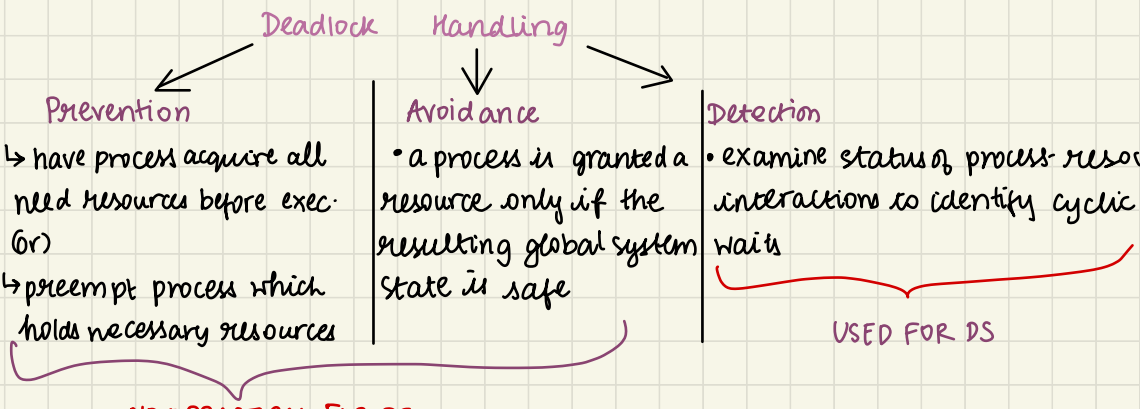
WFG example with deadlock cycle



with knot

Deadlock Handling

- Complicated in DS because no site has accurate knowledge of current state of system & communication involves finite/unpredictable delay



Deadlock detection → maintain WFG & search for cycles / knots
↙ AND deadlock
↘ OR deadlock

A deadlock detection algorithm must satisfy

↳ Progress (No undetected deadlocks)
- Algorithm must detect all existing deadlocks in finite time
  ie, after all wait-for dependencies are formed, algo shouldn't wait for events

↳ Safety (no false deadlocks)
- Shouldn't report Phantom deadlocks

Deadlock Resolution → breaking wait-for dependencies to solve deadlocks
→ involves rollback of deadlocked processes & reassigning resources to blocked Processes

Different Models

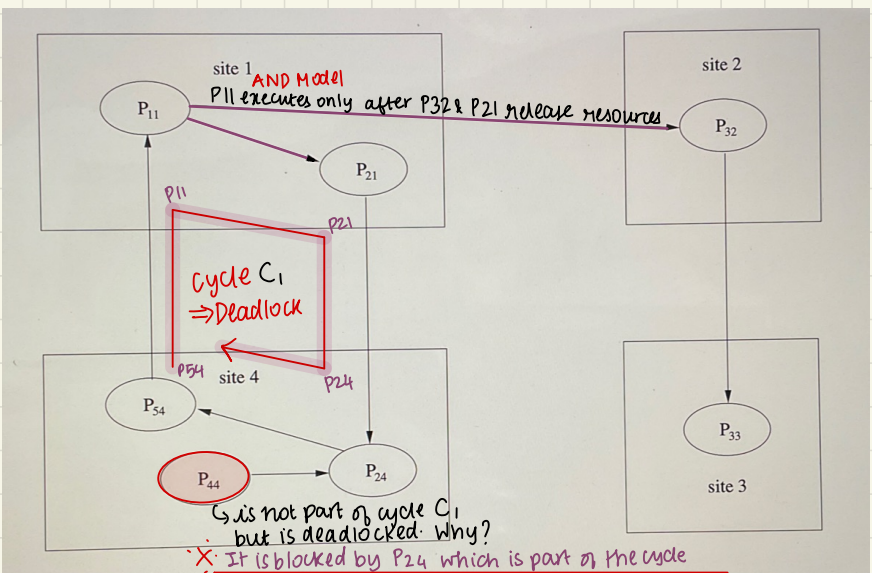
↳ Single Resource Model

- one process can have at most 1 outstanding req for a resource, ie, outdegree of WFG for a node = 1
- Presence of cycle ⇒ deadlock

↳ AND model

- process can have >1 resource req simultaneously & satisfied only after all resources rcvd; ie, execution happens only after node rcvs all resources
  outdegree of WFG for a node ≥ 1
- presence of cycle ⇒ deadlock

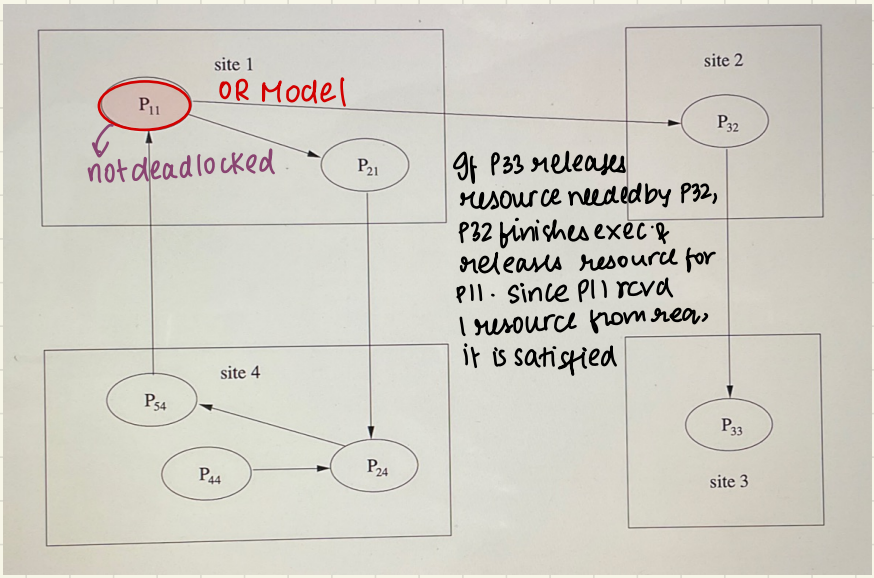
NOTE: AND model is more general than single resource model



process can be deadlocked if
↳ part of cycle (or)
↳ depends on a deadlocked process

↳ OR Model

- process can have >1 outstanding req for resource simultaneously, but begins execution if even 1 of the resources is granted
  (begins execution = req satisfied)
- presence of cycle ⇏ deadlock
- presence of knot ⇒ deadlock



↳ AND-OR Model

- generalisation of both AND & OR models
- process specifies kind of request as (and) / (or)
  eg: req can be x and (y or z)
- can't use WFG.
- have to use repeated application of the test for OR-model deadlock

↳ $\binom{p}{q}$ model

- p out of q model allows process to obtain any k (p) resources out of pool of n (q) resources
- same expressive power as AND-OR but more compact request
  Every $\binom{p}{q}$ request can be expressed as AND-OR & vicevera

- AND req $\equiv \binom{p}{p}$
- OR req $\equiv \binom{p}{1}$

↳ Unrestricted model

- no assumptions about requests other than deadlock is stable
- concerns ab. properties of problem separated from DS computations
  (stability, deadlock) (msg passing vs. sync comm)

Distributed deadlock detection algorithms

↳ Path Pushing

- deadlocks detected using explicit global WFG
- each site has local WFG, if it performs deadlock computation, send WFG to neighbors
- repeated until some site has sufficient info abt global state to announce presence / absence of deadlocks

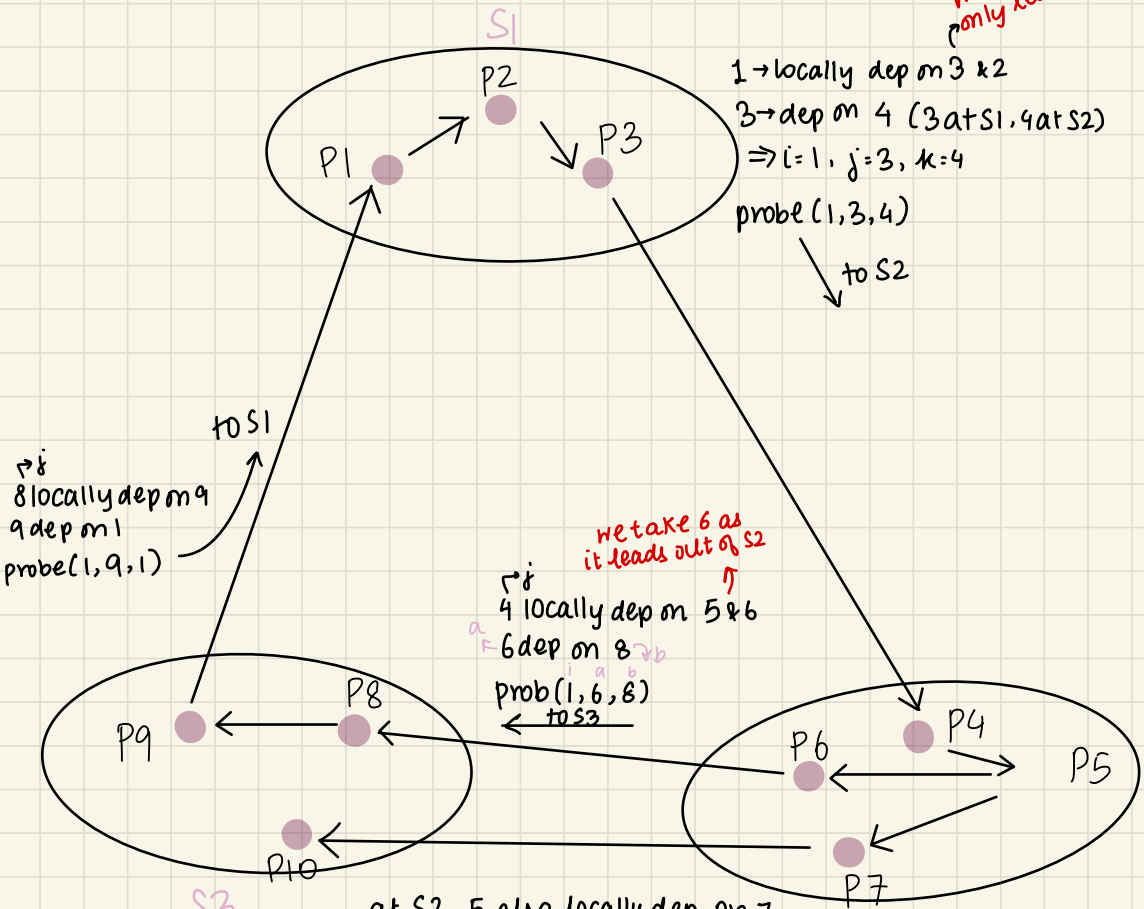
Drawbacks

- It detects phantom deadlocks
- algo sends $n(n-1)/2$ msgs to detect deadlock in n sites
- size of msg: $O(N)$
- delay in detecting deadlock : $O(n)$, so by the time its detected, it would be resolved

↳ Edge Chasing

- presence of cycle detected by propagating 'probes' along edges of graphs
- cycle detected when a node receives a probe identical to one it sent previously
- only blocked processes forward probe messages through outgoing edges
- Probes are fixed in size & short

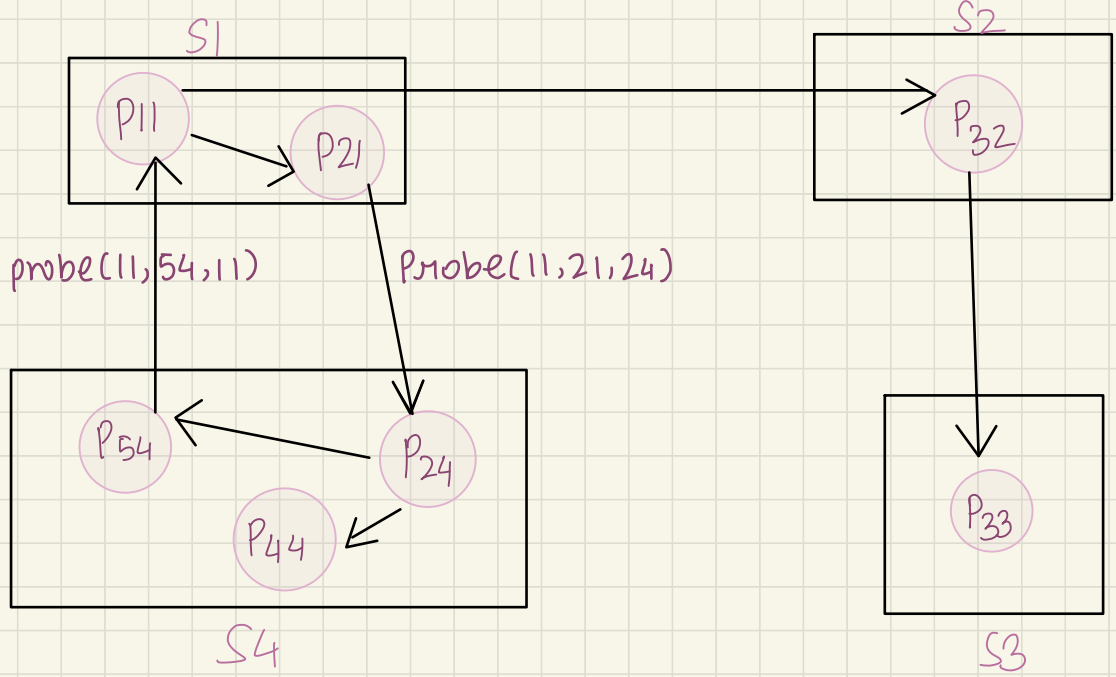
→ P_k is locally dependent on P_a
→ P_a is waiting for P_b
→ P_a, P_b are on different sites
send probe (i, a, b) to home site of P_b



at S2, 5 also locally dep on 7,
7 dep on 10
so, probe(1,7,10)
→ S3

but since 10 has no outgoing edges, it terminates

For m process at n sites, it exchanges $m(n-1)/2$ msgs.
Msg size → 3 int values
delay = O(n)



① at site 1

↳ P11 locally dep on P21
↳ P21 dep on P24 at S4
probe (11, 21, 24) to S4
i j k

② at site 4

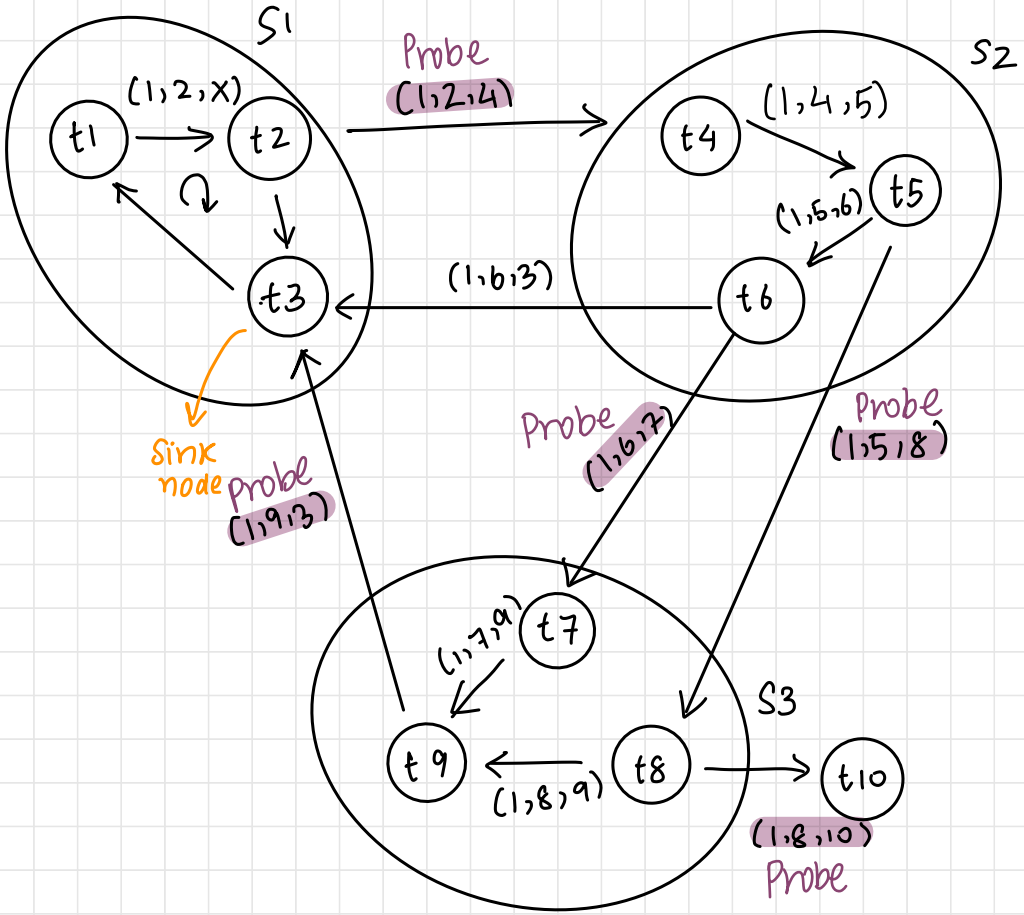
↳ P24 locally dep on P44 & P54
only P54 leads out of S4,
↳ P54 dep on P11
i a b
probe(11, 54, 11) to S1

↳ Diffusion Computation based

- deadlock detection is diffused through WFGs of the system
- make use of echo algorithm
- This computation superimposed over DS computation. If termination, initiator declares a deadlock
- To detect a deadlock, a process sends out a query through all outgoing edges in the WFG
- Queries are propagated through edges
- When a blocked process recvs first - query for a (specific) deadlock detection initiation, it only replies after all its own queries have received replies
- For subsequent queries from that same deadlock det. initiation, it replies immediately.
- Initiator of deadlock detection detects a deadlock when it recvs a reply for every query it sent out

initiator sends a query (if there is an outgoing edge)

(expects response)



Chandy Misra Haas Algorithm for AND model

↳ Based on edge chasing

↳ uses a probe (i, j, k) → deadlock detection initiated by P_i & is being sent by P_j at home site to P_k at another site

↳ if P_i receives a probe back, there is a deadlock.

Dependency: a process P_j is dependent on a process P_k if there is a seq $P_j, P_{i1}, P_{i2}, \cdots P_{im}, P_k$ such that

→ $P_{i1} \cdots P_{im}$ are blocked & P_i is waiting for resource from P_{i+1} & P_{im} waiting for P_k & P_j is waiting for P_i

ie, except P_k all processes are blocked & each process except P_j holds a resource for which the prev. process in seq is waiting

Local dependency: P_j is locally dependent on P_k if P_j depends on P_k & both P_j & P_k are in same site

Data Structures

↳ Each process P_i has an array dependent$_i$

dependent$_i(j)$ = true if P_i knows P_j depends on P_i

initially, dependent$_i(j)$ = false $\forall i$ & $\forall j$

y given in ppt but wrong

dependent$_i(j)$ ⇒ all P_j that holds a resource that P_i wants

Algorithm exactly same as Edge Chasing

Chandy Misra Haas Algorithm for OR model

↳ Based on diffusion computation

2 types of messages:

↳ query(i,j,K) & ↳ reply(i,j,K) } diffusion comp initiated by P_i, sent from P_j to P_K

↳ Initiated by a blocked process → sends query to all processes in its dependent set, (ie, all its outgoing edges)?

↳ If active node (node w. no outgoing edges), discard query

↳ When blocked process P_K, recvs query(i,j,K)

If this is the first query msg rcvd by P_K initiated by P_i → engaging query

- P_K propagates query to all processes in its dependent set
- sets local variable $num_K(i)$ = no of query msg sent

If it isn't engaging query,

- If P_K has been blocked since engaging query, send reply immediately
- else, discard

Process P_K has variable $wait_K(i)$ [true/false] to denote it has been blocked since query(i,j,K) from P_i

NOTE: Process sends a reply to an engaging query after it recvs replies to every query it sent out for this engaging query

↳ initiator detects deadlock when it rcvs replies to all queries it sent out

https://www.youtube.com/watch?v=qPINNeXtfUc

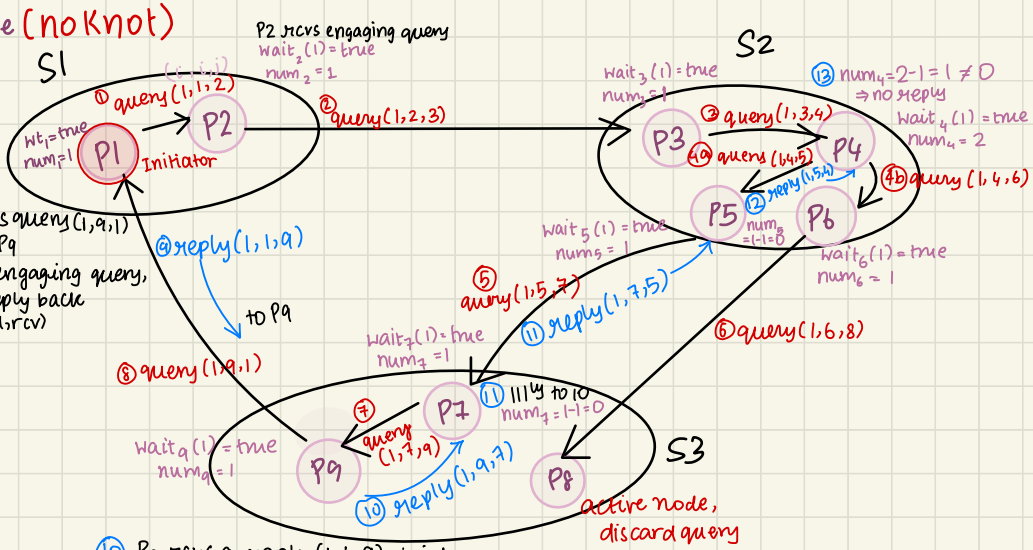
Algorithm:

1. Initiation by a blocked process Pi:
   send query(i,i,j) to all processes Pj in the dependent set DSi of Pi;
   (initator = sender = receiver)
   num(i) := |DSi|;
   waiti(i) := true;

2. Blocked process Pk receiving query(i,j,k):
   if this is *engaging* query for process Pk /* first query from Pi */
   then send query(i,k,m) to all Pm in DSk;
   numk(i) := |DSk|; waitk(i) := true;
   else if waitk(i) then send a reply(i,k,j) to Pj.
   ↳ from the process P_j that P_K rcvd query from

3. Process Pk receiving reply(i,j,k)
   if waitk(i) then
   numk(i) := numk(i) - 1;
   if numk(i) = 0 then
   if i == k then declare a deadlock.
   else send reply(i, k, m) to Pm, which sent the engaging query.

Example (no knot)



⑩ P9 rcvs a reply(1,1,9) i=j=1, K=9

$wait_K(i)$ = true ✓

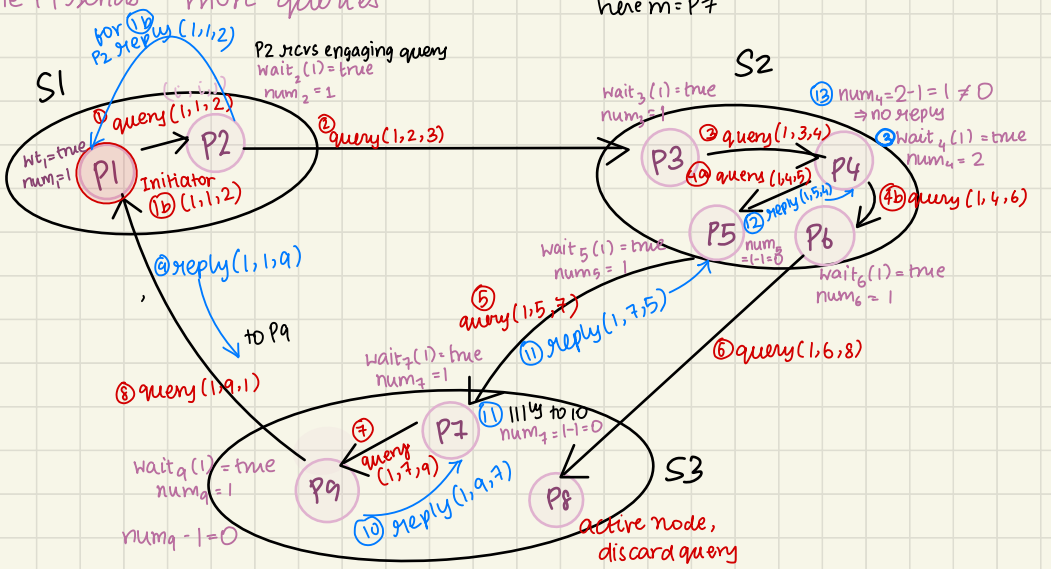
$num_9 - 1 = 0$

∵ num = 0, $K \neq i$,

so send reply to (i, K, m)

↳ process which sent engaging query, here m = P7

Assume P1 sends more queries



But, P1 sent 2 queries & received only 1 reply ⇒ algo cant terminate as should be no of queries = no of replies

Modify diagram by adding link b/w P8 → P9 creating a knot