Mutual exclusion

- only one process is allowed to execute the Critical section (CS) at any given time.
- In distri systems shared variable (semaphores) cannot be used to implement mutal exclusion (Message passing is the only way)
- ✓ these algorithms is complex ∵ these have to deal with ~~un~~ unpredictable message delays & incomplete knowledge of system states.

(a) Token based approuch:

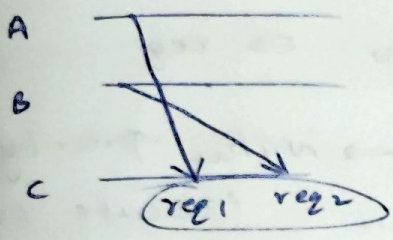
- uniqUe token (= privilege msg) is shared among sites
- site is allowed to enter CS if it possesses token & it continues to hold token until execution of CS is over.
  ∴ token is unique mutual exclu is ensured
- algorithms differ in way ~~site~~ processes seaches for token.

(b) Non-token based approuches:

- 2 or more successive rounds of messages are exchanged among sites to find which site will enter CS Next.
- site enters CS $\overset{\text{assersion}}{\wedge}$ if local variables → true
- ∴ assertion becomes true only at one site at any given time, mutual exclu ensured

Quorum based approaches:

- each site requests permission to execute CS from subset of sites (quorum)
- quorums are formed such a way that when 2 sites concurrently req access to CS atleast one site receives both req & this site is responsible to make sure that only one req execute CS at any time.



accept only one req

System model:

- N sites $S_1, S_2 \cdots S_n$, process at site S_i is P_i
- All processes communicate asynchronously over underlying communication N/w.
- A process wishing to enter CS req all other or subset of processes by sending 'req' msgs & waits for appropriate replies before entring into CS.
- while waiting the process is not allowed to make further req to enter CS
- Site can be in 3 states
  - → req CS
  - → in CS
  - → neither (req CS) nor (in CS) (idle)

- In token based algo a site can hold token but not in cs such state is idle token state.
- site may have several pending req for cs (site queues all these & serves at a time)
- we assume channels reliably deliver all msgs & sites do not crash
- Many algos uses Lamports style logical clock to assign time stamp to cs req
  - lower time stamp → higher priority to execute cs

Requirements of mutual exclusion algos:

① safety property:

- at any instant only one process can execute cs.

② Liveness property:

- absence of deadlock & starvation
- i.e, 2 or more sites should not endlessly wait for msgs that will never arrive.
- a site should not wait indefinitly to execute cs while others are repeatedly executing cs
- i.e, every site req cs should get opportunity to execute cs in finite time.

Fairness : (each process gets fair chance to execute CS)

- CS execution req are executed in order of their arrival in system.

(A) → necessary (B)(C) → important

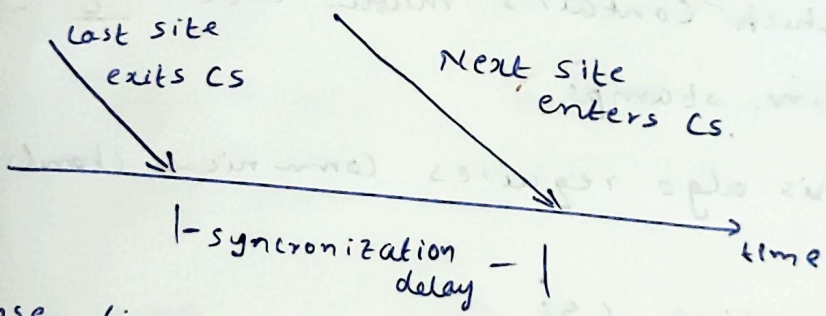
Performance metrics:

(A) message complexity:

- Num of msgs required per CS execution by a site

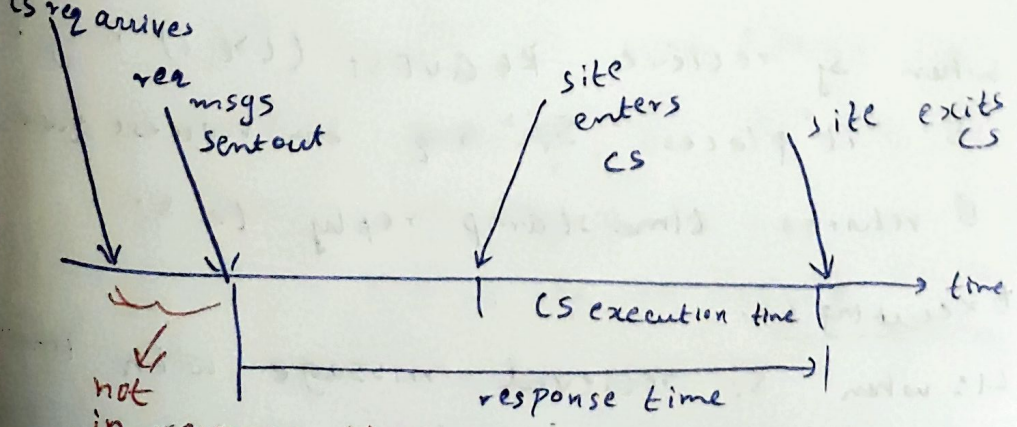
(B) synchronization delay:

- after site leaves CS, it is time required before next site enters CS.



(C) response time:

- time interval a request waits for its CS execution to be over after its requ msgs have been sent out



④ System throughput:

- rate at which sys executes reqests for CS

SD → synchronization delay

E → avg critical section execution time

$$\text{sys throughput} = \frac{1}{(SD+E)}$$

Lamport's algorithm:

- when site posesses req for CS, it assigns timestamp to req
- executes CS req in ↑ order of timestamp
- every site S_i keeps queue, request_queue which contains mutu exc req ordered by time stamps.
- This algo requires communi channels to be FIFO

requesting CS:

- S_i want to enter CS, it broadcasts REQUEST (ts_i, i) msg to all other sites & places request on request_queue$_i$
- when S_j recieves REQUEST (ts_i, i) msg from S_i it places S_i's req on request_queue$_j$ & returns time stamp reply to S_i

Executing CS:

4: when S_i recieved mssage with timestamp larger than (ts_i, i) from all sites

i.e. S_i's request is at top of request-queue;

Releasing Critical section:

- Site S_i upon exiting CS, removes its req from top of its request-queue; & broadcasts timestamped RELEASE message to all other sites
- when site S_j recieves RELEASE msg from S_i it removes S_i's req from request-queue.

- when site gets REQUEST, REPLY or RELEASE msg it updates its clock using time stamp in msg.

Correctness:

- Lamport's algo achieves mutual exclusion

Proof: (contradiction)

- Suppose 2 sites S_i, S_j executing CS concurrently for this to happen L1 & L2 must hold both sides concurrently i.e, both S_i & S_j have their own requests at top of their req-queue
- assume S_i's req has < timestamp than S_j's
- from L1 & FIFO property, so S_i's req is in $req\text{-}queue_j$ when S_j is in CS.
- this implies S_j's own req is top of it's ow $req\text{-}queue_j$ when smaller time stamp S_i's req is present in $req\text{-}queue_j$ contradiction ---

- lamport's algo is fair

Proof:

- fair → req of cs are executed in order of timestamps

by contradiction

- suppose S_i's req has smaller time stamp than any other S_j, & S_j is able to execute cs before S_i

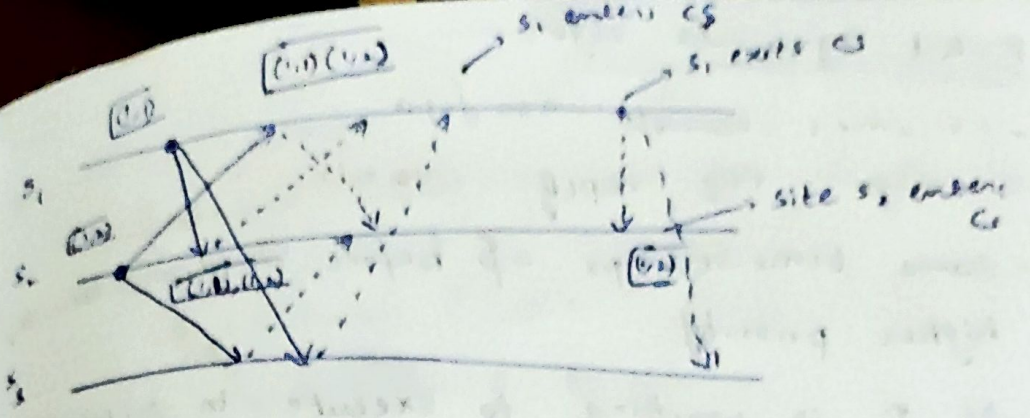
---

eg: S_1, S_2 got replies msgs from all others

S_1 has its req at top of req_queue$_i$

but S_2 has (not) " " req_queue$_j$

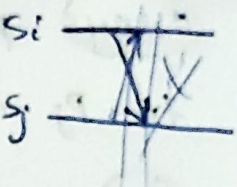
- so S_1 executes cs & sends release msg to all other
- S_2 has recieved REPLY from all other sites & also recieved RELEASE msg from S_1
- S_2 updates its req_queue$_j$ & its req will be top of its req_queue$_j$



Performance:

for each CS (N-1) req
(N-1) replies
(N-1) release

$\Rightarrow 3(N-1)$



optimization:

reply messages can be ommitted in certain situations.

eg: If S_j recieves req from S_i after it has sent its own req msg with timestamp higher than time stamp of S_i's req then S_j need not send reply message to site S_i

∵ S_i reciceves S_j's req with timestamp higher than its own (S_i), S_i can conclude that S_j don't have smaller timestam req which is still pending.

3(N-1) to 2(N-1) per CS execution

Ricart-Agrawala algo:

- assumes channels as fifo
- 2 msgs, req, reply (No relays)
- same timestamps as before & lower id has higher priority.
- If P_i is waiting to execute in critical sec recieves REQUEST msg from P_j then (if) priority of P_j's request is lower then P_i defer delays REPLY to P_j & sends REPLY after to P_j after executing CS for its pending req.

  (else)
  P_i sends reply immediatly to P_j

∴ If several processes are req execution of CS, highest priority request succeeds in collecting all needed REPLY msgs to get into CS.

- each process P_i maintains request-deffered array, RD_i (size num of processes in system)
- initially $\forall i \forall j\ RD_i[j] = 0$
- whenever P_i defers request sent by P_j then $RD_i[j] = 1$ & after it has sent REPLY to P_j it sets $RD_i[j] = 0$.
- site recieves msg it updates its clock with timestamp in msg.
- so next highest priority site gets all REPLYs.

requesting critical section:

- when S_i wants to enter CS, it broadcasts timestamped REQUEST message to all other sites
- when S_j recieves REQUEST msg from S_i it sends REPLY msg to S_i

if S_j ((neither requesting nor executing CS) or ((S_j is requesting) & (S_i's req timestamp is < S_j's own req timestamp)))

else reply defered $RD_j[i] = 1$

executing critical sections:

- site S_i enters CS after it recieved REPLY msg from every site it sent REQ msg

releasing CS:

- when site S_i exit CS it sends defered REPLY msgs $\forall j$ if $RD_i[j] = 1$ sends to all S_j & sets $RD_i[j] = 0$

Correctness:

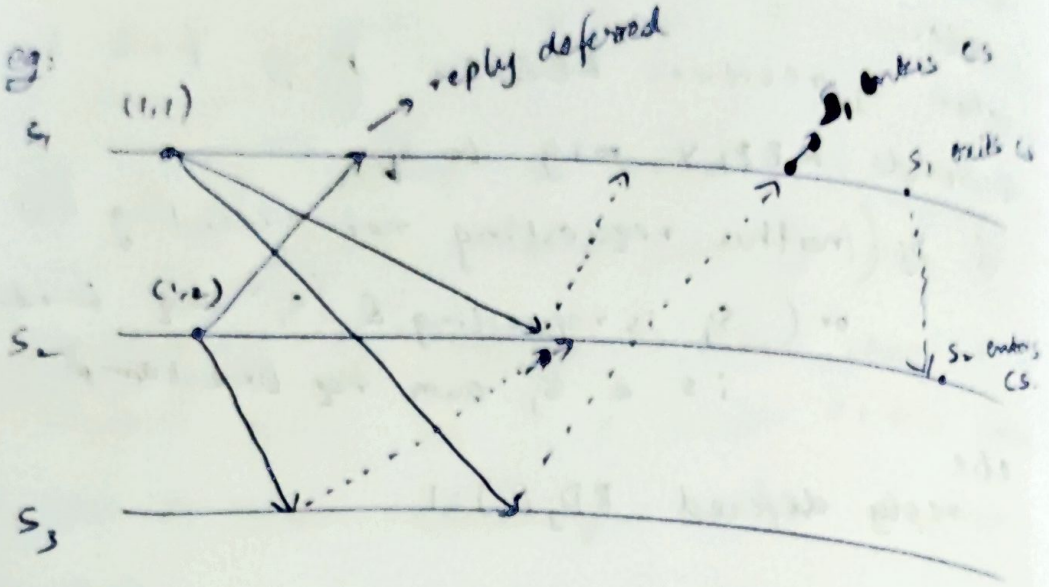
- achieves mutual exclusion

Proof: (contradition)

- eg S_i & S_j executing concurently CS. S_i's req has high prority than req of S_j so S_i recieves S_j's req after S_i made it own req, in order to S_j concurently enter CS if & only if S_i sent reply to S_j befor S_i exits CS. However this is impossible ∴ S_j's (lower priority

- for every pair of sites requesting for CS higher priority site will always defer req of lower priority site

Eg:



Performance:

N-1 req & N-1 replies for each CS execution

$\therefore$ 2(N-1) msgs for each CS execution

Singhal dynamic information structure algorithm:

- most mutual exclu algo use static approach to invoke mutual exclusion. i.e, they always take same course of actions to invoke mutual exclusion no matter state of system.

eg: if some sites asking freq to CS some very less freq then freq invoking site need not ask permission of less freq invoking site every time it want to go to cs. (only take permission from freq sites)

- information structure of algo evolves with time as sites learn about state of system through msgs.

- FIFO, network reliable, no ~~chan~~ crashes

Data structures:

- Information structure at site S_i consists 2 sets,

R_i → request set, contains sites from which S_i must aquire permission to get into CS

I_i → inform set, contain sites to which S_i must send its permission to execute CS after ~~its~~ executing its CS.

- each site maintains 3 boolean variables to denote state of site :

requesting → true if site is requesting

executing → true " " " executing

My priority → true if pending req of S_i has priority over current incoming req.

Initialization:

For each site S_i ($i = 1$ to n)

{

$R_i = \{S_1, S_2 \cdots S_{i-1}, S_i\}$

$I_i = S_i$

$C_i = 0$

}

requesting = executing = false

∴ initially S_i req only to $S_i, S_{i-1}, \cdots S_1$

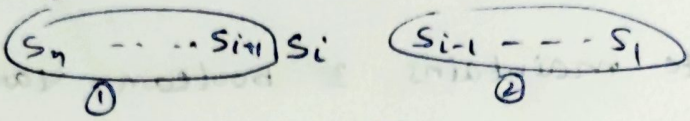
$S_n, S_{n-1} \cdots S_i, S_{i-1}, \cdots S_1$

$S_n, S_{n-1}, S_{n-2} \cdots S_i, S_{i-1}, \cdots S_1$

intial system has 2 properties

① every site requests permission from all sites to its right & no site to its left vice versa (All sites to its left asks per. no site to it right asks)

∴ every site S_i divides all sites into 2 disjoint sets



all sites in group ① req perm from S_i & S_i requests all sites in group ②

(ii) Cardinality of R_i will decrease from left to right (staircase pattern)

Description:

- S_i executes 3 steps to invoke mutex cs
- REQUEST msg handler at site processes incoming REQUEST msgs.
- It takes actions such as updating info structure & sending REQ/REPLY msgs to other sites.
- REPLY msg handler processes incoming REPLY msgs.

step-1: (Request critical section)

requesting = true

$C_i = C_i + 1$

Send REQUEST (C_i, i) msg to all sites in R_i

wait until $R_i = \phi$

requesting = false.

step-2: (Execute cs)

Executing = true

Execute cs

Executing = false

step-3: (Release cs)

for every site S_k in I_i (except S_i)

{

$I_i = I_i - \{S_k\}$

Send REPLY (C_i, i) msg to S_k

$R_i = R_i + \{S_k\}$ ✗ S_k may not be in cs in future

}

REQuest msg handler (S_i handling msg REQUEST (C_j, j))

$C_i = \max(C_i, C_j)$

Case

Requesting = True:

{

if 'my priority true'. // if pendi- req of S_i has priority over the incoming req

{

$I_i = I_i + \{S_j\}$; // ?

}

else

{

send REPLY (C_i, i) msg to S_j

if ($S_j \notin R_i$)

{

$R_i = R_i + \{S_j\}$

send REQUEST(C_i, i) msg to S_j

}

}

}

Executing = true

$I_i = I_i + \{S_j\}$

Executing = false $\wedge$ Requesting = false

{

$R_i = R_i + \{S_j\}$

send reply (C_i, i) msg to S_j...

}

REPLY msg handler

// site S_i handling msg REPLY (C_j, j)

{

$C_i = \max(C_i, c)$

$R_i = R_i - \{S_j\}$ // ~~No need to~~ ask

}

- S_i acquires permision from req set R_i & it relases CS by sending reply msg to all in set I_i
- If S_i itself requesting CS if higher priority req msg comes from S_j then S_i takes
  (i) S_i sends back reply to S_j
  (ii) If S_j is not in R_i then also S_i sends reply to S_j & S_i places S_j in R_i

• else (req of S_i has more priority than S_j)

S_i places an entry of S_j into I_i so that S_i can reply back to S_j after execution of CS.

- If S_i recieves req msg from S_j when S_i is in CS then it simply puts S_j in I_i (so that S_i can reply back to S_j after CS)

- If S_i recieves req msg from S_j when S_i nor req nor executing CS then it places entry of S_j in R_i & sends reply to S_j

* Site to execute CS last, places it self at right end of staircase pattern.

Correctness:

Quorum based mutual exclusion algorithms:

(i) - site does not request permission from all other sites but only from subset of sites
- req set of sites are choosen such that $\forall i \forall j : 1 \le i, j \le N \quad R_i \cap R_j \neq \phi$ i.e, every pair of sites has site which mediates conflict b/w that pair.

(ii) in quorum based mutual ex algo site can send out only one REPLY msg at any time, a site can REPLY msg only after it has recieved a RELEASE msg ~~from~~ for prev REPLY msg.
$\therefore S_i$ locks all sites in R_i in exclusive mode before executing its CS.

Coteries:

coterie 'C' is defined as set of sets where $g \in C$ is called quorum.

(i) intersection property:
If $g, h \in C \Rightarrow g \cap h \neq \phi$

eg. $\{1,2,3\}$ $\{2,5,7\}$ $\{5,7,9\}$ cannot be quorums in coterie $\because S_1 \cap S_3 = \phi$

(ii) minality property:
there should not be quorums g, h in coterie C $\Rightarrow g \supseteq h$
eg: $\{1,2,3\}$ $\{1,3\}$ ✗

Let 'i' be site in quorum 'A', if 'i' wants to invoke mutual exclusion it requests permission from all sites in quorum 'A'.

Due to intersection property quorum 'A' contains atleast one site permission to only one site at any time

∴ mutual exclusion is garunteed.

Maekawa's algorithm:

request sets of sites (quorums) are constructed to satisfy

M1 $(\forall i\ \forall j : i \neq j \quad 1 \leq i,j \leq N :: R_i \cap R_j \neq \phi)$

M2 $(\forall i : 1 \leq i \leq N :: S_i \in R_i)$

M3 $(\forall i : 1 \leq i \leq N :: |R_i| = k)$

M4 any site S_j is contained in k number of R_is $1 \leq i, j \leq N$

$|R_i| = \lceil \sqrt{N} \rceil$ $\quad (N = k(k-1)+1)$

- from M_1 every pair has one common site which can handle conflits.
- A site can only one REPLY msg at any time i.e, it grants permission to an incoming Req if it is not granted permission to some other site.

* it algo req in order delivery of msgs)

- M_1 & M_2 are necessary
- M_3 → size of all R_i's = k
- M_4 → exactly same Num of sites should request permission from any site, i.e., all sites have "eq probability" in granting permission to others.

Requesting CS:

ⓐ site S_i req access to CS by sending REQUEST(i) msgs to all sites in R_i

ⓑ S_j recieves REQUEST(i) msg, sends a REPLY(j) msg to S_i (provided it hasn't sent REPLY msg to site its recipent of last RELEASE msg) else it queues up REQUEST(i) for later.

Executing CS:

- site S_i will execute CS if it has recieved all reply msgs from R_i set sites.

Releasing CS:

- After exec of CS is over, S_i sends RELEASE msg to every site in R_i
-

Correctness: (Mutex algo achieves mutex exclusion)

Proof (by contradiction)

Suppose 2 sites S_i & S_j concurrently executing CS, i.e. S_i recieves replies from all R_i & S_j recieved replies from all sites in R_j

if $R_i \cap R_j = \{S_k\}$ then S_k must have to sent REPLY msg to both S_i & S_j which is contradiction.

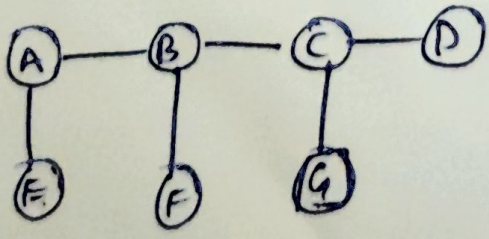
Performance:

Token Based algo:

- unique token is shared among sites
- site allowed to enter CS if it has token
- use seq Num instead of time stamps
- every request for token contains seq num & seq numbers of sites advance independently.
- site increments seq num counter every time it makes req for token.

Raymond's tree based algo:

- uses spanning tree of compu N/w to reduce num of msg exchanges per critical section execution.
- algo exchanges $O(\log N)$ msg under light load & 4 msgs under heavy load (N → Num of nodes on N/w)
  to execute CS.
- Nodes are vertices of graph, & links b/w nodes are edges.
- minimum spanning tree (minimum cost)



- each node needs to hold information about & communicate only to its immediate

neighbouring nodes.

i' needs to know holds infor— about

ii B, D, G

This algo uses privilege to enter into cs

i.e. only one node can have that at any time. (called privileged Node)

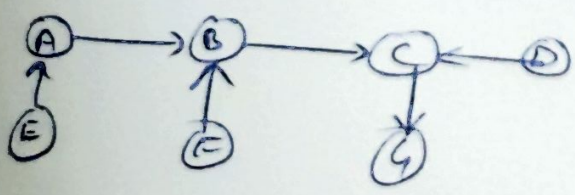
except when privilege is in transit from one node to other via Privilege msg.

Holder variables.

- node stores in HOLDER variable the identity of node that it thinks has privilege or leads to privilege node

- HOLDER$_x$ = Y we can redray X — Y into X → Y

- If G holds privilege

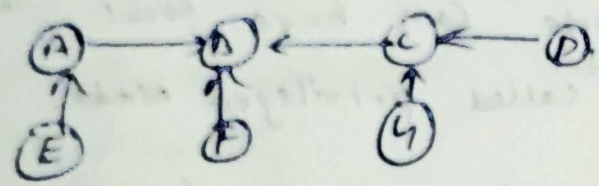


HOLDER$_A$ = B
" B = C
" F = B
" C = G
" D = C
" E = A
G = self

- B which does not hold privilege want to execute CS

→ B sends Req msg to HOLDER$_B$ (i,C) which in turns to G (HOLDER.)

- If G don't need privilege then sends PRIVILEGE msg to neighbour C' reverse path of (B to G) & changes $HOLDER_G = C$ & $HOLDER_C = B$



TERMINATION Detection

- Detection of termination is not trivial in DS ∵ no process has knowledge of global state & global time

① At any time a process can be only one of 2 states (active → busy) when it is doing local computation & (idle → passive) temporarily finished execution of its local computation. & will be reactivated only on receipent of of msg from another process.

② an active become idle at any time (when it completed local ~~execution~~ compun & has proceded all recieved msgs)

* msg used in underlying compution are basic msgs
* mgs " for purpose of termination detection Control msgs.

Definition of termination detection:

Let $P_i(t)$ denote state (active or idle) of process P_i at time t & $C_{i,j}(t)$ is num of msgs in transit in channel at time t from P_i to P_j, DS is said to be termin at t_0 iff:

$$(\forall i :: P_i(t_0) = idle) \wedge (\forall i,j :: C_{i,j}(t_0) = 0)$$

Termination detection using distributed snapshots:

- consistant snapshot of DS is taken after distributed computation has terminated: then snapshot will capture termination of computation.

* Communication is reliable but NON FIFO. Msg delay is arbitary but fenite.

→ main idea is when computation terminates there must exit, a unique process which became idle last

- when process goes from active to idle it issues req to all process to take snapshot & it also req itself to take local snapshot.

Description:

- logical clock to order req^{s}
- each process has logical clock 'x' = 0 initially
- process ↑ x by 'i' of each time it becomes idle
- basic msg sent by process at its logical time 'x' is form B(x)
- control msg sent by process 'i' at logical time 'x' is of form R(x,i)
- besides 'x' every process maintains variable 'k' when process is idle

(x,k) is max value of all R(x,k) ever recieved or sent by process.

⇒ $(x,k) > (x',k')$ iff $(x > x')$ or $((x = x')$ and $(k > k'))$

R1: when process 'i' is active it may sent basic msg to Pj at any time by

send B(x) to j

upon receiving B(x') ~~from~~ process P_i does

let $x = x' + 1$

if (i is idle)

→ go active

when process 'i' goes idle

$x = x + 1$

$k = i$

send msg R(x, k) to all others;

take a snapshot for req by R(x, k)

upon recieving msg R(x', k') process 'i' does

if ((x', k') > (x, k) and (i is idle)) // sender of R(x', k') terminates after this process.

{ (x, k) = (x', k')

take local snapshot for req R(x', k')

}

if ((x', k') < (x, k) and (i is idle))

{

do nothing (req is ~~sent~~ started before P_i took its snapshot so no need to take again)

}

if (i is active)

x = max (x', x)

② Termination detection by weight throwing:

- Process called controlling agent monitors computation.
- extra communication channel b/w (controlling agent & all process) & (every pair of process)

Basic idea:

- intially all processes are idle & weight of each process is zero. & weight of controlling agent = 1
- Computations starts when controlling agent sends basic msg to one of processes & a non-zero weight 'w' is assigned to each process when in active state

when process sends msg to other it sends some weight, when proc— recieves msg it adds that weight to its weight

∴ sum of weights of all processes & weights on msgs in transit = 1

when process becomes passive it sends weight to controlling agent. in control msg. controlling agent concludes on termination if it weights becomes '1'

B(Dw): basic msg 'B' is sent as part of computation where Dw is weight assign—

C(Dw): control msg 'C' is sent from proc— to controlling agent

Rule-1:

Controlling agent or active process may send a basic msg to one of processes say p' by splitting its weight 'w' into w_1 & w_2 such that $w_1 + w_2 = w$ $(w_1 > 0, w_2 > 0)$ it assigns $w = w_1$ & sends $B(DW = w_2)$ to p'

Rule 2:

Correctness:

A: set of all weights of all active process

B: " " " on all basic msgs in transit

C: " " " " all control " " "

w_c: weight on controlling agent

I_1: $w_c + \sum_{w \in (A \cup B \cup C)} w = 1$ (sum of all wei = 1)

I_2: $\forall w \in (A \cup B \cup C)$ $w > 0$ (active pr wei of weight on all transit msgs > 0)

$\therefore w_c = 1$

$\Rightarrow \sum_{w \in (A \cup B \cup C)} = 0$ (from I_1)

$\Rightarrow A \cup B \cup C = \phi$ (by I_2)

$\Rightarrow A \cup B = \phi$

it tells computation is terminated

$\Rightarrow A \cup B = \phi$ msg delay is finite

$w_c + \sum_{w \in C} w = 1$

after some time $w_c = 1$ $\therefore$ algo detects termination in finite time.

Distributed Shared Memory:

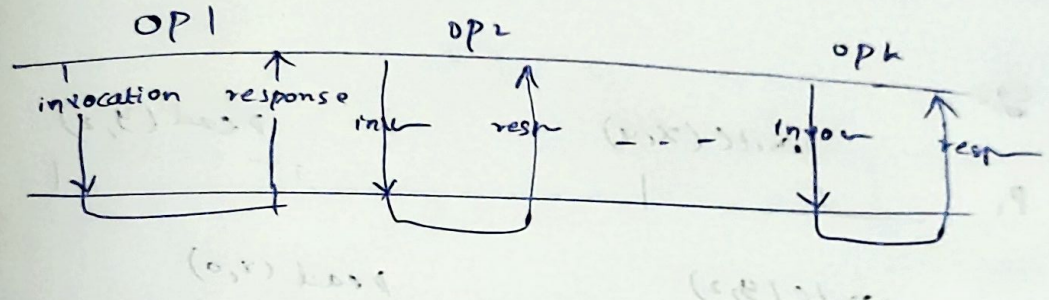
Memory consistent models:

Memory coherence is ability of system to execute memory operations correctly.

Assume n processes, s_i memory operations per P_i (process)

$$\frac{(s_1 + s_2 - - - + s_n)!}{s_1!\ s_2! \cdots s_n!}$$ } total possible permutations or interleavings

- Problem is to identify which is correct



- when we write 'a' to x → Write (x, a)
- read 'x' & it returned 'a' → Read (x, a)

Strict consistency / Linearizability / Atomic consistency

- read should return most recent value written per global time axis
- if opers do not overlap as per global time it is obviously sequential
- operations that overlap as per global time

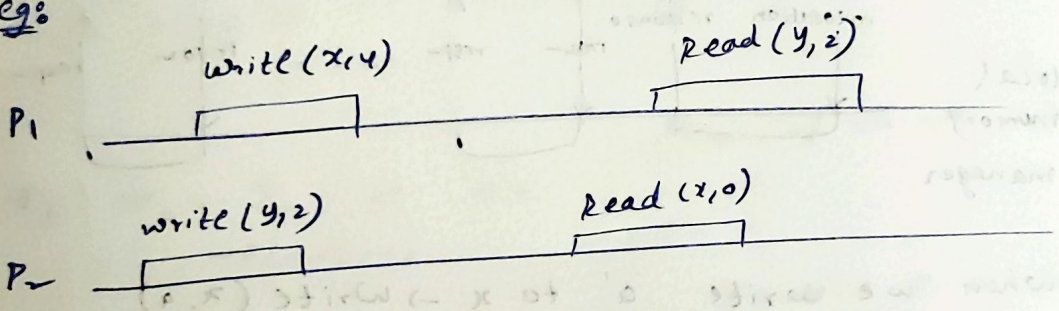
(i) all processes see same ordering of events equivalent to global time ordering of non-overlapping events

(ii) all operations appear to be atomic & sequentially executed.

Sequential consistency:

- result of execution is same as if all processeors were executed in seq
- operations of each individual processor appear in seq in local pgm order.
- even if 2 diff processors do not overlap in global time scale they may appear in reverse order in common seq order seen by all.

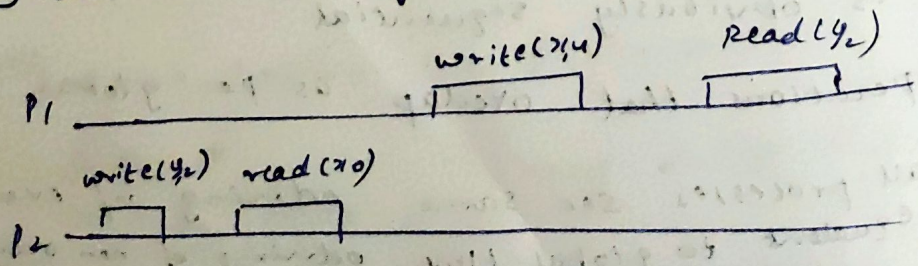
eg:

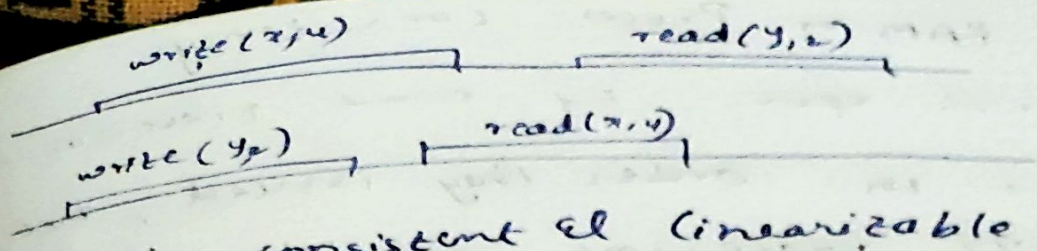


ANS sequentially consistent but not linearizable.

→ P_2 read begins after P_1 write (x,4) but read returns value existed before write(x,4)

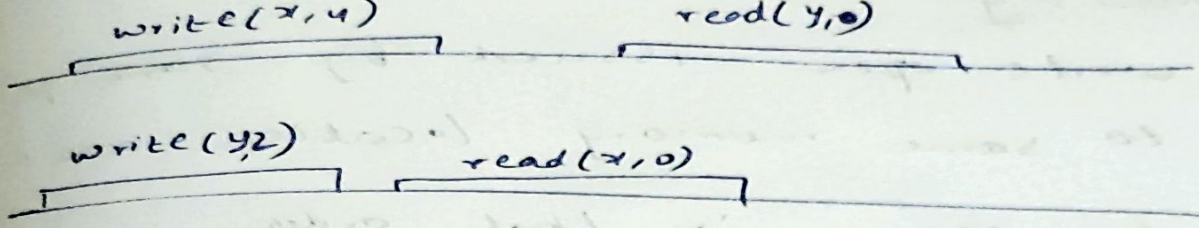
- sequential is because we can reorder global reorder by





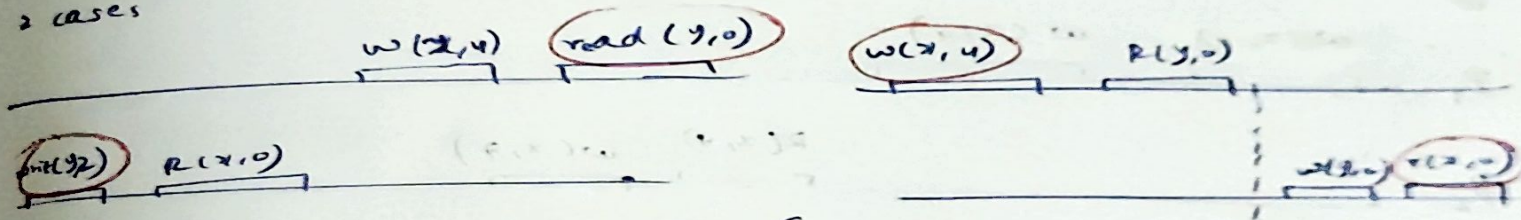
sequentially consistent & linearizable as well

*If it is linearizable then 100% it is seq



Not sequentially consistent (∵ not linearizable)

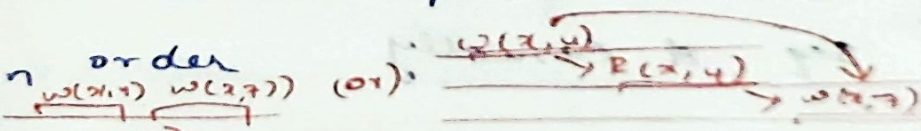
2 cases



in both cases it is failing.

Casual consistency:

- In seqm consis write opern should be seen in common order (or):



- In casual consistency only casually related writes should be seen in common order.

→ at a process local events are casually ordered

→ write casually precedes read issued by another process if read returns value written by write

→ transtive closure of above 2 orders is casual order.

Pipelined RAM or Proces...

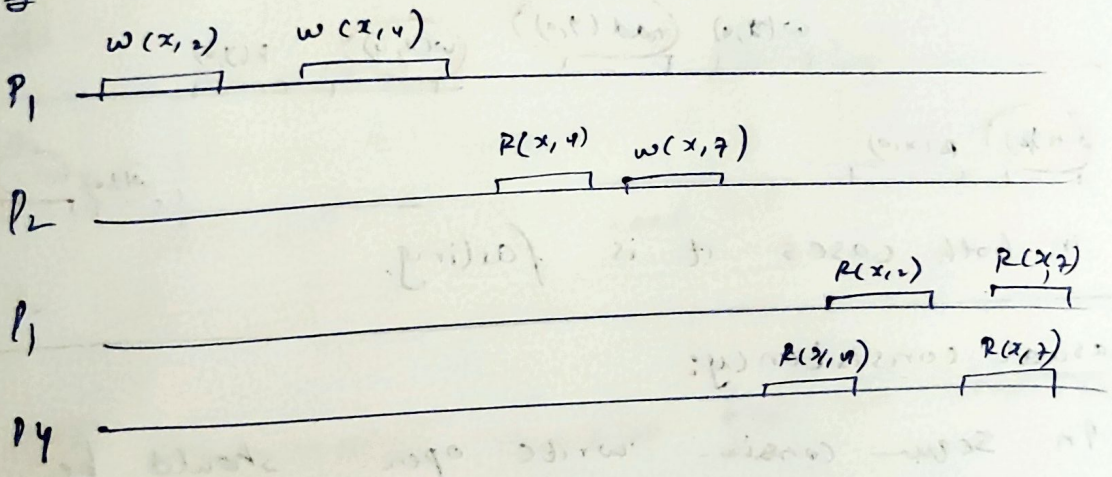
only write oper by same processor are seen by others in order they issued, but writes from diff processors may be seen by other processors in diff order.

Slow memory:

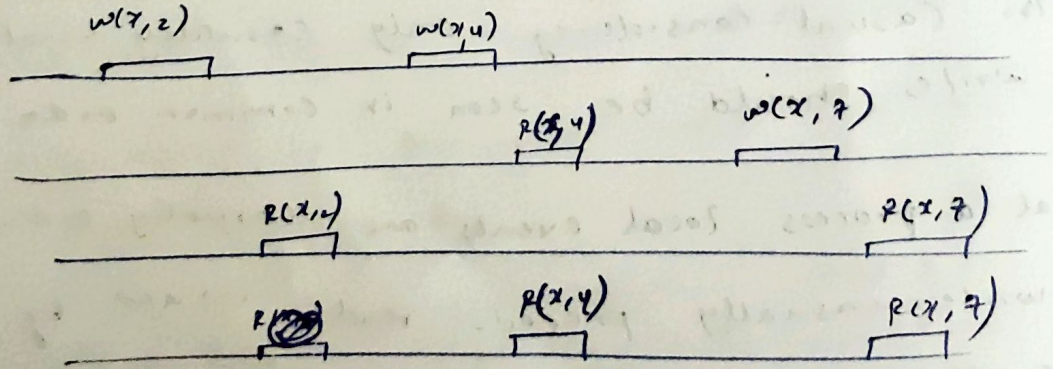
only write oper issued by same processor and to same memory location must be seen by other in that order.

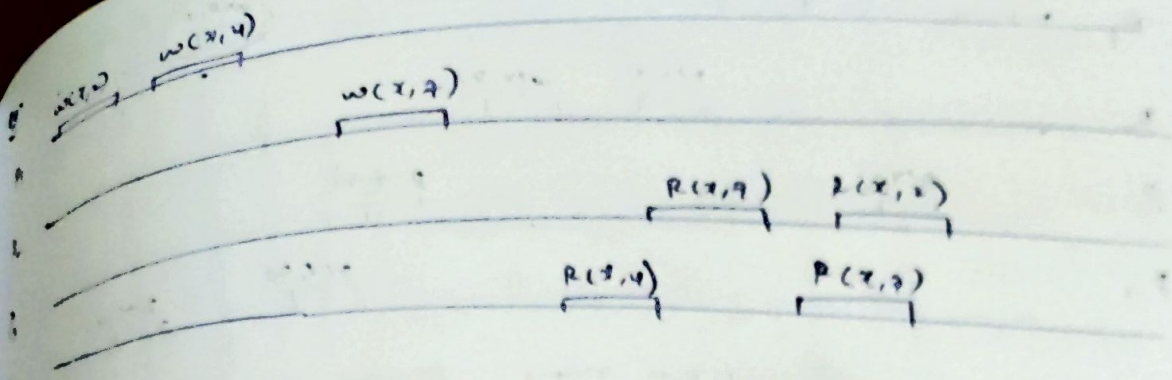
---

eg:

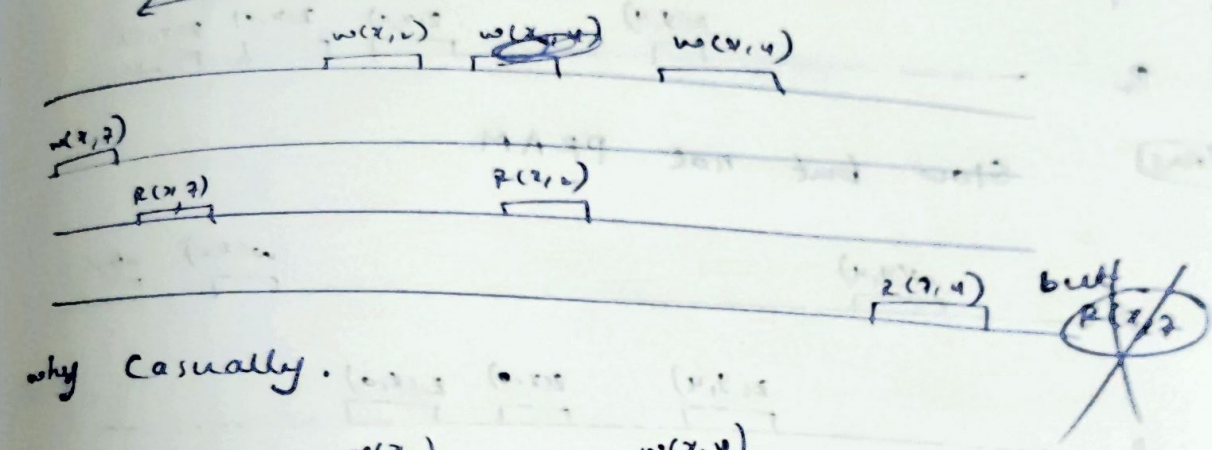


ANS seq & Casually

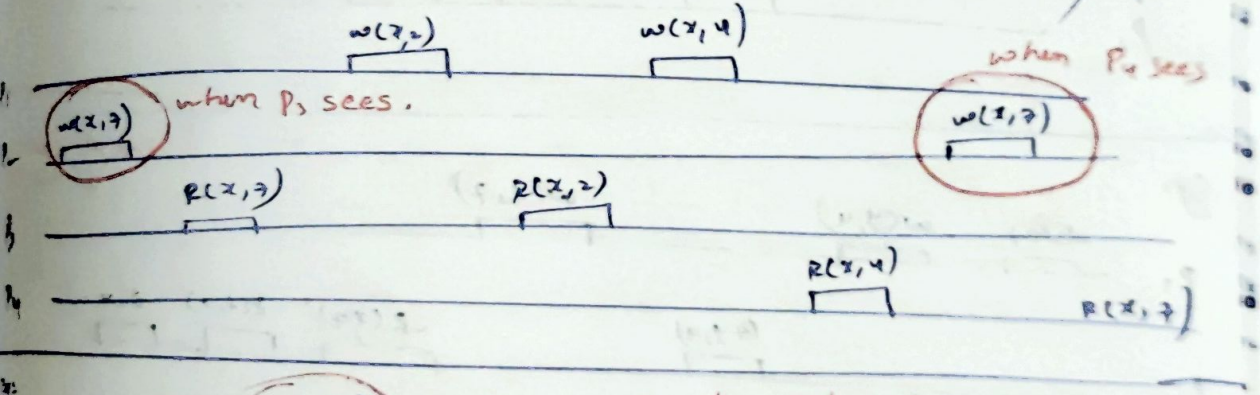


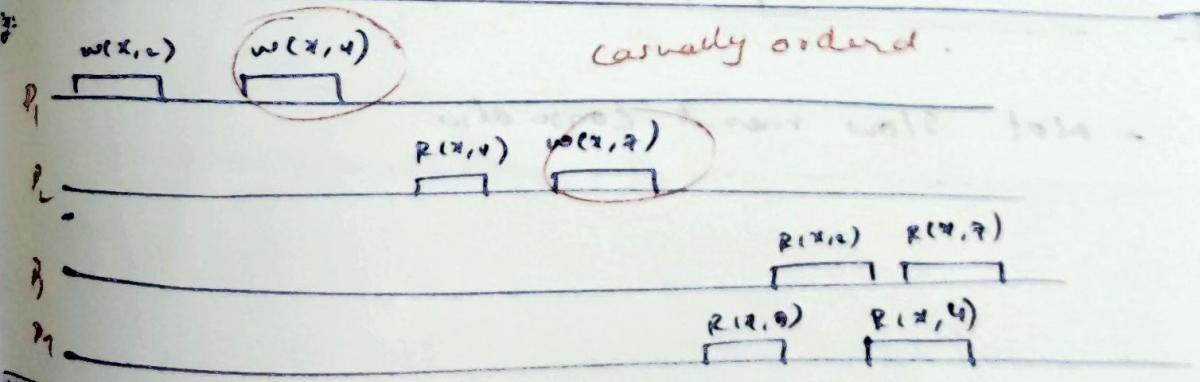


casually but not sequentially.



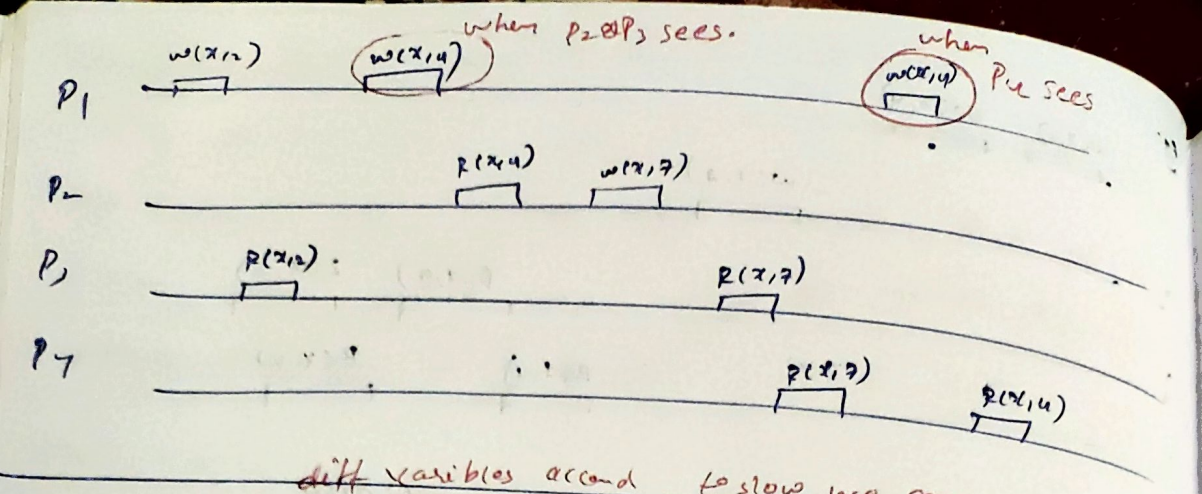
why casually.





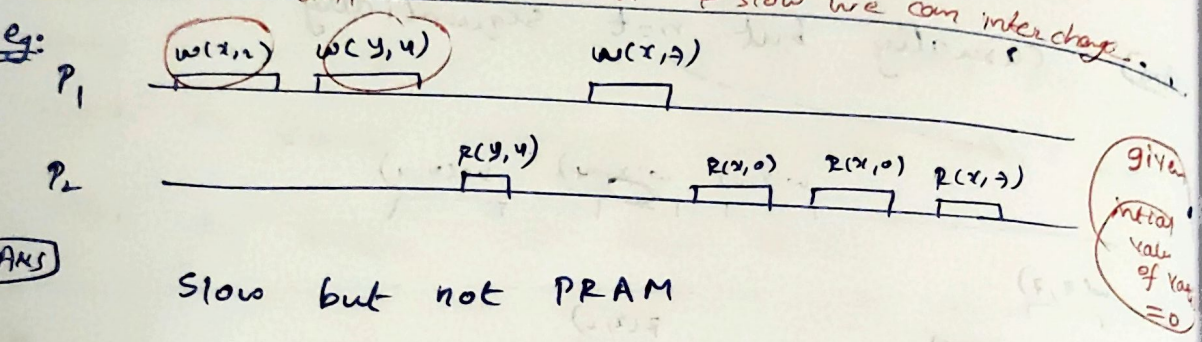
Not casual, Not seq, PRAM

So P_3 only must see $w(x,7)$ after $w(x,4)$ in casual consistent



diff variables accord to slow we can interchange...

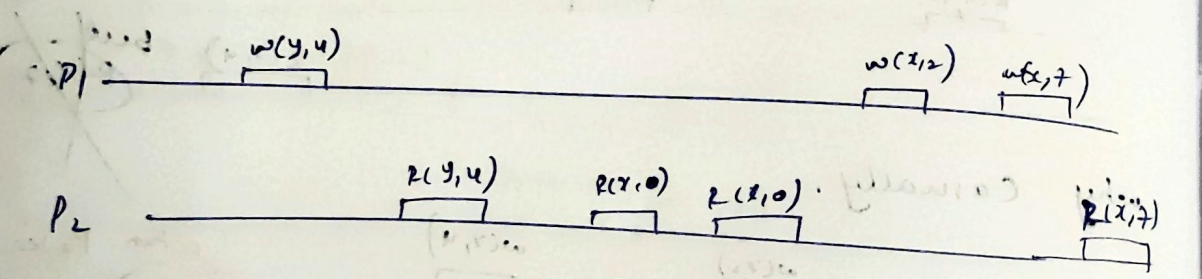
eg:



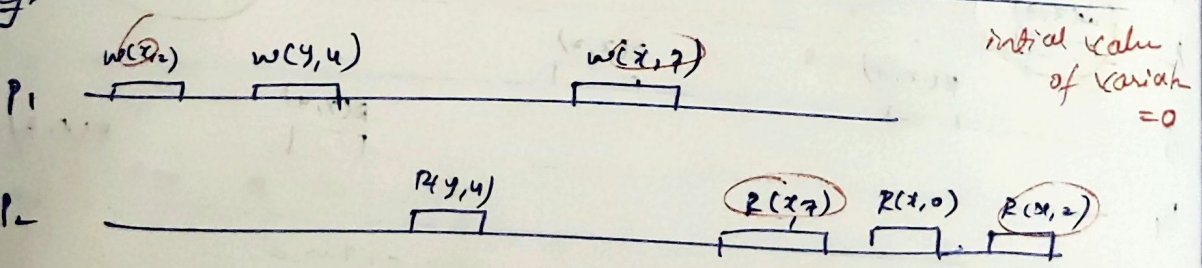
given initial value of var = 0

ANS

Slow but not PRAM

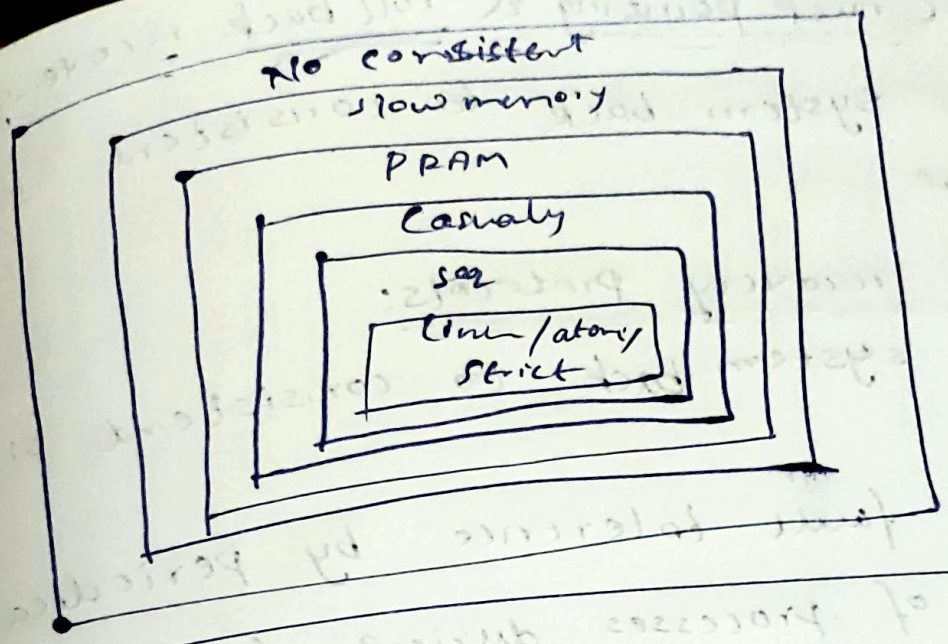


eg:



initial value of variable = 0

- Not slow ment consumption


No consistent
slow memory
PRAM
Casualy
seq
Linearizable/atomic
Strict

Check pointing & roll back recovery:

- restore system back to consistent state after failure

Roll back recovery protocols:

- restore system back to consistent state after failure
- achieve fault tolerence by periodically saving state of processes during failure free execution
- treats DS application as collection of processes over N/w.

Check points:
saved state of process.

why roll back recovery of DS is complicated?

AND Msgs induce inter process dependencies during failure free operation.

i.e, upon failure of one or more processes in system these dependencies may force some of processes that did not fail, to roll back this is rollback propagation.

eg: If P_1 sends msg to P_2 Now P_1 rolls back to state that preceds sending of msg 'm' P_2 should also roll back to state that preceds recieve of 'm' else it will be inconsistent.

This phenomenon of cascaded rollback is called

if each process takes checkpoints separately then system can't avoid domino effect.

this is independent or uncoordinated checkpointing

techniques to avoid domino effect:

1. Co-ordinated checkpointing
   - processes co-ordinate their checkpoints to form systemwide consistent state
   - in case of failure system state can be restored to such a consistent set of checkpoints (preventing roll back propagation)
2. Communication induced checkpointing:
   - forces each process to take checkpoints based on information piggybacked on application msgs it receives from other processes.
   - checkpoints are taken such a way that system wide consistent state always exist on stable storage
3. Log based rollback recovery:
   - combines checkpointing with logging of non deterministic events
   - relies on "piecewise deterministic (PWD) assumption

Local check point:

- All processes save their state at certain instances of time
- local check point is snapshot of state of process at given instance.
- Assumptions
  - process stores all local checkpoints on stable storage
  - process is able to roleback to any of its existing local checkpoints.
- $C_{i,k}$ → k^{th} local checkpoint at process P_i
- $C_{i,0}$ → process P_i takes ~~snapshot~~ checkpoint $C_{i,0}$ before its starts execution.

Consistency system states:

global state of DS:

- collection of all individual states of participating processes & states of communication channels.

Consistent global state:

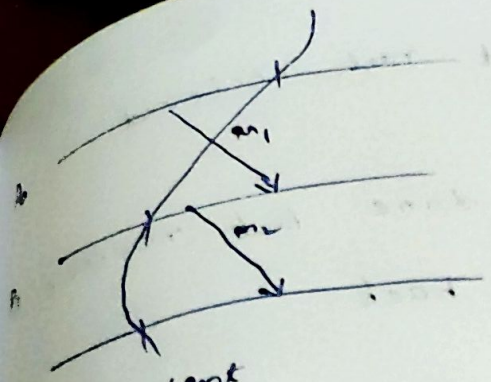
- If reciever is recorded then send must be also be recorded.

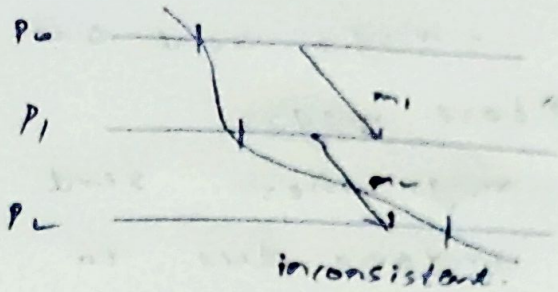
global check point:

- set of local check points one from each process

consistent global check point:

- No msg is sent by process after taking its local point that is recieved by another process before taking its snapshot.





Interactions with outside world:

to recieve i/p data or deliver o/p of computation.

If failure occurs outside world cannot be expected to roll back.

Outside world process: (OWP)

- special process that interacts with rest of system through msg passing

- before sending o/p to OWP, system must ensure that state from which o/p is sent will be recovered despite any future failure

Symbol "||":

interaction with outside world to deliver outcome of computation.

Diff Types of msgs:

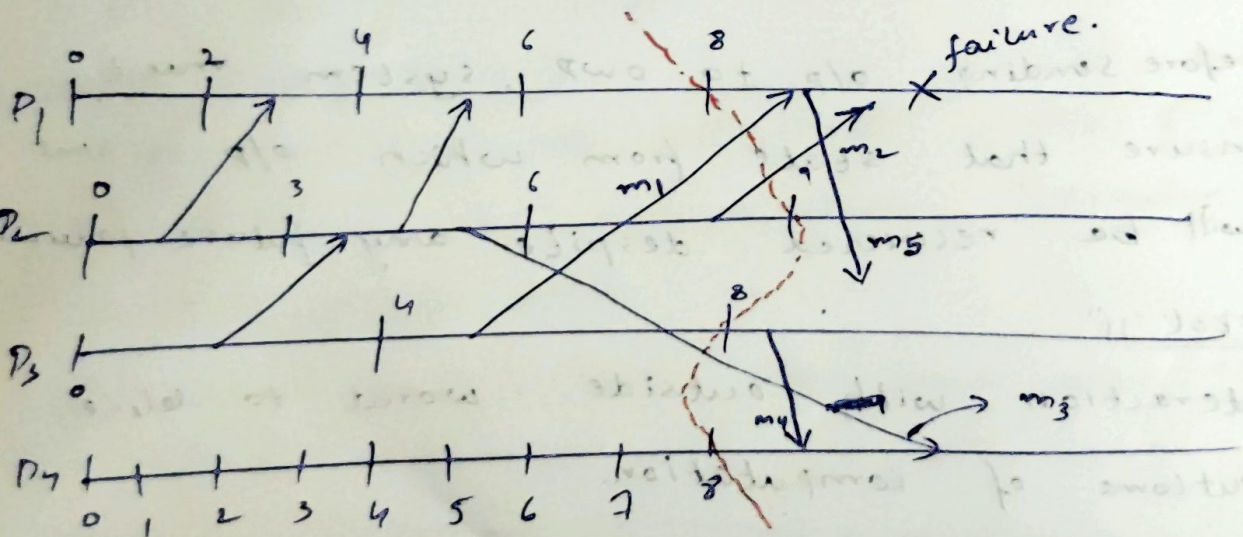
- At process failure & recovery can leave msgs that were executed before failure :
rollback of processes for recovery may rollback send & recieve operation of several msgs.

(i) in-transit msgs:
- msgs sent out but not recieved

(ii) Lost msgs:
- msgs whose sent is done but recieve is undone due to roll back

(iii) Delayed msgs:
- msgs whose recv was not recorded bcoz recieve process was either down or msg arrived after roll back

(iv) orphan msgs:
- msg with recieve recorded but send not recorded
- do not arise if roll back to consistent global state



in transit m_1, m_2

lost m_1 (prev it is recieved but undone)

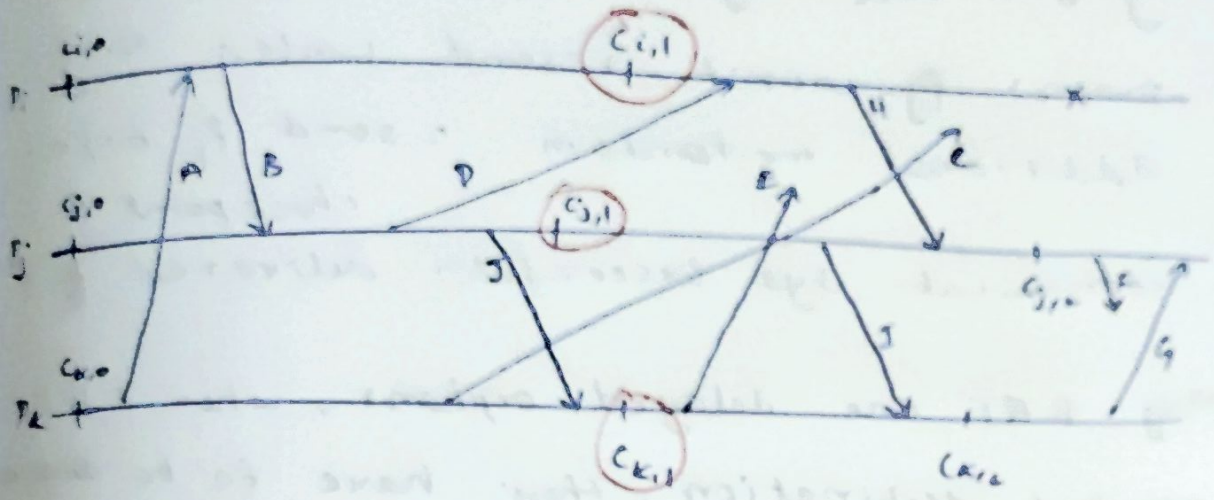
delayed m_2, m_5

orphan none

3) Duplicated msgs:

ues in failure recovery:

. we have to handle msgs that are left abnormally state due to failure & recovery



. restored global state is $\{C_{i,1}, C_{j,1}, C_{k,1}\}$ → consistent

why $C_{j,1}$ why not $C_{j,2}$

∵ msg 'H' will become orphan if $C_{j,2}$

∵ recv will be in $C_{j,2}$ but sent is undone

Now msg 'I' will become orphan if $C_{k,2}$

so P_k also have to roll back to $C_{k,1}$

→ A, B J msg no problem ∵ send & recv before $\{C_{i,1}, C_{j,1}, C_{k,1}\}$ (normal ~~completed~~ msgs)

→ I, H, G are completly undone (vanished msgs)

- Now C, D, E, F are problamatic

- Msg 'c' is in transit during failure & it is delayed msg.

  3 possibilites
  - 'c' might arrive at P_i before it recovers
  - it might arrive while P_i is recovering
  - " " " after P_i completed recovery

- Msg 'D' is lost msg ∵ recieve is undone
  - Process P_j won't resend unless any additional mechanism. ∵ send P_j before checkpoint & communicate sys successfully delivered 'D'

- Msg. E & F are delayed orphans, when E & F reached destination, they have to be discarded ∵ their sent is undone.

Deadlock detection in DS:

- Process may req resources in any order which may not be known a priori & a process can req resourse while holding others.

Assumptions:
- systems have only reusable resources
- processes are allowed to make only exclusive access to resources
- only one copy of resource.

→ Process can be in 2 states

(i) running or active:
- process has all needed resources & is either executing or is ready for execution

(ii) blocked:
waiting for acquire some resource.

---

wait for graph: (directed graph)

- represents state of system.
- nodes are processes & if directed edge b/w P_1 to P_2 if P_1 is blocked & is waiting for P_2 to release some resourse.
- System is deadlock if there exits a cycle in WFG

Deadlock handling strategies:

- 3 techniques for handling deadlocks
  - deadlock prevention
  - " avoidance
  - " detection.
- It is not trivial since DS msg will be having delays
- deadlock prevention is commonly achieved either by having a process acquire all needed recouses simultaneously before it begins execution or by preempting a process that hold needed resource.
- deadlock avoidance approch in DS is resource is granted to process if resulting global state is safe but in DS we can't do this
- Deadlock detection:

requires an examination of status of process resourse interactions of presence of cyclic wait.

- It is best approuch for DS

Issues in deadlock detection:

(a) Detection of deadlock:

- 2 issues → maintainence of WFG
  → seaching WFG for cycles (knot)

- cycle or knot may involve several sites, seach for cycles depends on how WFG of system is repres-

Correctness criteria

(i) progress
(ii) Safety.

Models of deadlock:

(i) single resource model:

- each process has at most one outstanding req for only one unit of resource.
  (out dgree is atmost 1)
- presence of cycle is deadlock in WFG

(ii) AND model:

- req more than one resource simultaneously
  dreq is satisfied if all resources are granted to process
- presence of cycle in WFG indicates deadlock in AND model
- If cycle in WFG for AND model ⇒ deadlock but vice versa not true.
  ∴ process may not be in cycle but in deadlock

Deadlock detection in DS:

- Process may req resources in any order which may not be known a priori & a process can req resourse while holding others.

Assumptions:

- systems have only reusable resources
- processes are allowed to make only exclusive access to resources
- only one copy of resource.

→ Process can be in 2 states

(i) running or active:
- process has all needed resources & is either executing or is ready for execution

(ii) blocked:
waiting for acquire some resource.

---

wait for graph: (directed graph)

- represents state of system.
- nodes are processes & if directed edge b/w P_1 to P_2 if P_1 is blocked & is waiting for P_2 to release some resourse.
- system is deadlock if there exits a cycle in WFG

Deadlock handling strategies:

- 3 techniques for handling deadlocks
  - deadlock prevention
  - " avoidance
  - " detection
- It is not trivial since DS may will be having delays

- deadlock prevention is commonly achieved either by having a process acquire all needed recourses simultaneously before it begins execution or by preempting a process that hold needed resource.

→ deadlock avoidance approach in DS is resource is granted to process if resulting global state is safe but in DS we cant do this

→ Deadlock detection:

requires an examination of status of process resourse interactions of presence of cyclic wait.

- It is best approach for DS

Issues in deadlock detection:

(a) Detection of deadlock:

- 2 issues → maintainence of WFG
  → seaching WFG for cycles (knots)
- cycle or knot may involve several sites. seach for cycles depends on how WFG of system is repres.

Correctness criteria

(i) progress
(ii) safety.

Models of deadlock:

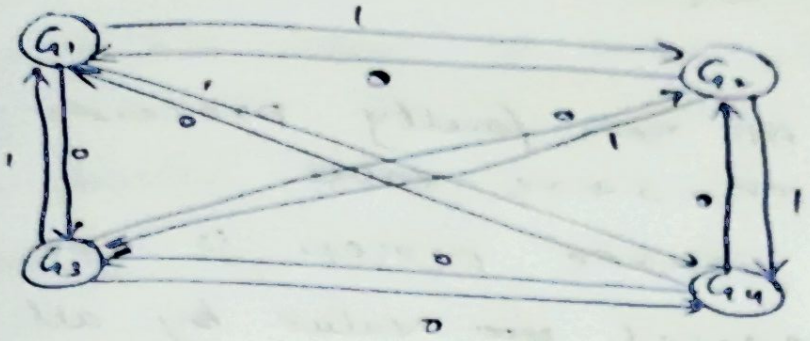
(i) single resource model:

- each process has at most one outstanding req for only one unit of resource. (outdgree is atmost 1)
- presence of cycle is deadlock in WFG

(ii) AND model:

- req more than one resource simultaneously
  dreq is satisfied if all resources are granted to process
- presence of cycle in WFG indicates deadlock in AND model
- If cycle in WFG for AND model ⇒ deadlock but viceversa not true.
  ∴ process may not be in cycle but in deadlock

Byzantine generals sending confusing msgs.



- 4 camps attacking army has to attack all at a time then only they will succeed.
- Asyncronus system is modeled b/w them which has unbounded delay to travel b/w the
- Lost msg is modeled by messenger being captured by enemy
- Byzantine process is modeled by general being a traitor. He will mislead informa to other generals.
- He may inform one general to attack at 12am & inform others at noon. or he might not send msgs to other generals.
- In figure various generals are conveying misleading values so we have to find which is correct.

Byzantine agreement problem:

- source process with an initial value to reach agreement with other processes about its initial value

(i) Agreement: All non faulty processes must agree on same value.

(ii) Validity: If source process is non faulty then agreed value by all non faulty must be same & equal to initial value of source

(iii) Termination: each non faulty process must eventually ~~agree~~ decide on value.

→ If source is faulty then non-faulty processes can agree upon any value

→ It is irrelevent what faulty process agree upon.

Consensus problem:

- difference is that each process has initial value & all correct processes must agree on single value.

(i) agreement: All non faulty agree on same value.

(ii) validity: If all non-faulty processes have same initial value then agreed value of all non faulty process must be same as that initial value

(iii) Termination: each non faulty process must eventually ~~agree~~ decided on value

Interactive consistent problem:

Difference is that each process has initial value, all correct processes must agree upon set of values, with one value of each process

(i) Agreement:
All non-faulty processes must agree on same array of values $A[v_1, v_2 \cdots v_n]$

(ii) Validity:
- If P_i is non fault & its intial value is v_i, then all non faulty processes must agy on v_i as ith element of array 'A'
- If P_j is faulty then non-faulty processes can agree on any value of $A[j]$

(iii) Termination:
every non faulty process must eventually decide on array 'A'.

Agreement in failure free system:

- consensus can be collected from diff. processes arriving at decision & distributing this decision in system.

each process broadcasts its value to all others, each process computes some function on recieved value

eg (max, min, majority)

Agreement in (msg-passing) synchronous systems with failures:

Consensus algo for crash failures:

- n process, up to f processes may fail in fail-stop model
- Here consensus variable 'x' is integer valued, each process has intial value x_i
- If up to 'f' failures are to be tolerated, then algo has 'f+1' rounds.
- In each round process P_i will send value of its variable x_i to all other processes if that value is not sent before.
- of all values recieved in each round & it own value x_i at start of round, process takes min & updates x_i, after 'f+1' rounds local value x_i is garunted to be consensus value

Algo:

global value int f;

int x; // local value.

Process P_i ($1 \le i \le n$) executes consensus algo. for up to 'f' crash failures

```
for round from 1 to f+1
{
    if current value of x is not broadcasted
    {
        broadcast(x)
    }
    y_j ← value (if any) received from P_j in this round
    x_i ← min_{v_j} (x_i, y_j);
}
o/p 'x' as consensus value
```

Correctness:

① aggrement condition is satisfied :: in 'f+1' rounds there must be atleast one round in which no process failed, say round 'r' in this round all processes that have not failed so far suscceed in broadcasting their values

consensus algo for Byzantine failures:

* system of 'n' processes, Byzantine agreement problem can be solved in synchronous system only if num of Byzantine processes 'f' such that $f \leq \lfloor n-1/3 \rfloor$