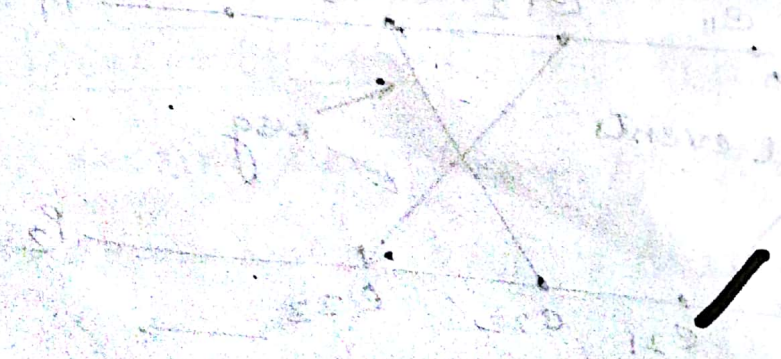
# Unit-1 (Distributed Systems)

## Advantages of Distributed System:-

1. Lower cost and higher System throughput
2. Shared resources (both h/w & s/w)
3. fault Tolerance

## Issues in D.S

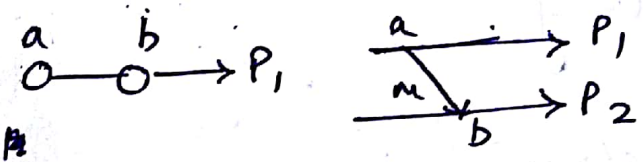
1. No Shared memory
2. No global clock
3. arbitrary msg delays
4. Scalability
5. Mutual exclusion for Shared resources
6. Deadlock detection etc



# (Logical Clock)

## Happened Before Relation

- Assume two events 'a' & 'b'.
- We say a → b ('a' happened before 'b') if either of the following are true:
  - a & b are events in the same process
  - a is an event of sending message 'm' at one process, & b is the event of receiving msg m at another process.

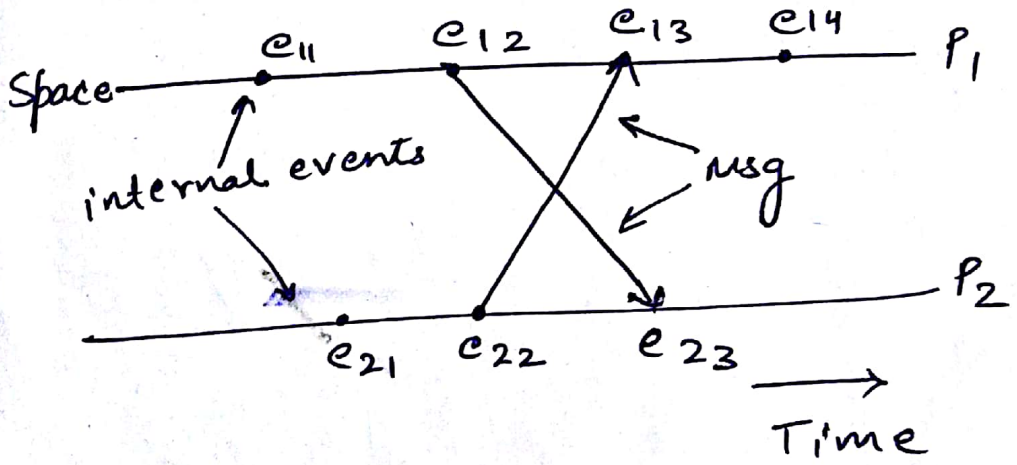



→ Transitive relation :- if a → b & b → c, then a → c

→ Causally ordered events: a → b, a affect b

→ Concurrent events: a || b a ↛ b & b ↛ a

## Space time diagram

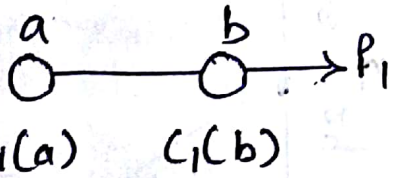


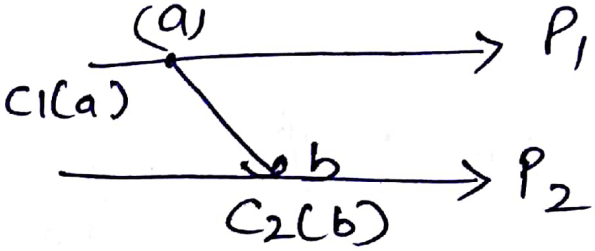
## Lamport Clock: Scalar Logical Clocks.

→ Monotonically increasing integer.

→ Condition to be satisfied by the logical clock

- if a & b are two internal events in $P_i$ such that $a \rightarrow b$, then $C_i(a) < C_i(b)$
- if a is the event of sending msg m at $P_i$ and b is the event of receiving msg m at $P_j$ then $C_i(a) < C_j(b)$




## Implementation of L.C (Rules

R1: for two successive internal events,

$C_i = C_i + d$ $\quad C_i = C_i + 1$ $\quad (d > 0)$

R2: for msg m sent from $P_j$ at timestamp $t_m$, the value of $C_i$ at $P_i$ = $\max(C_i, t_m) + 1$

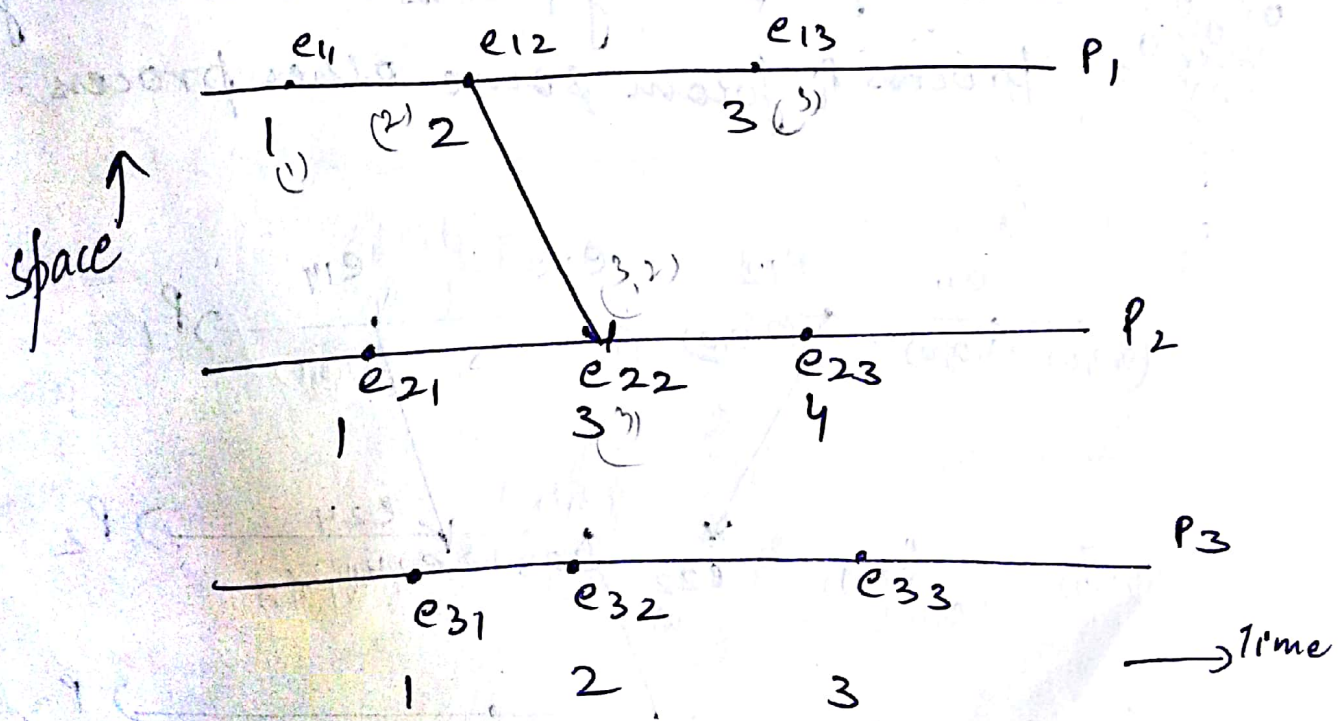
$P_i = \max(C_i, t_m + d)$ $\quad (d > 0)$

# Causal ordering of Events across Processes

(1) Given any two events at two different processes we want to be able to say whether the two events are causally related or not.

(2) with the Lamport's scalar logical clocks, we can say that for any two events a in $P_i$ and b in $P_j$, if $a \to b$, $C_i(a) < C_j(b)$

- But, if $C_i(a) < C_j(b)$, then we cannot say for sure whether $a \to b$ or not (i.e. we cannot determine the causal relationship b/w the two events)
- This characteristic is referred to as partial order.




even though $C(e_{11}) < C(e_{32})$, we can't say whether $e_{11}$ happened before $e_{32}$ or not

→ we can't establish causal relationship b/w these two events.

# Vector Clock (Total ordering)

- Vector is an array

$T^i = C_i[1-n]$ is a vector maintained at process $P_i$

where

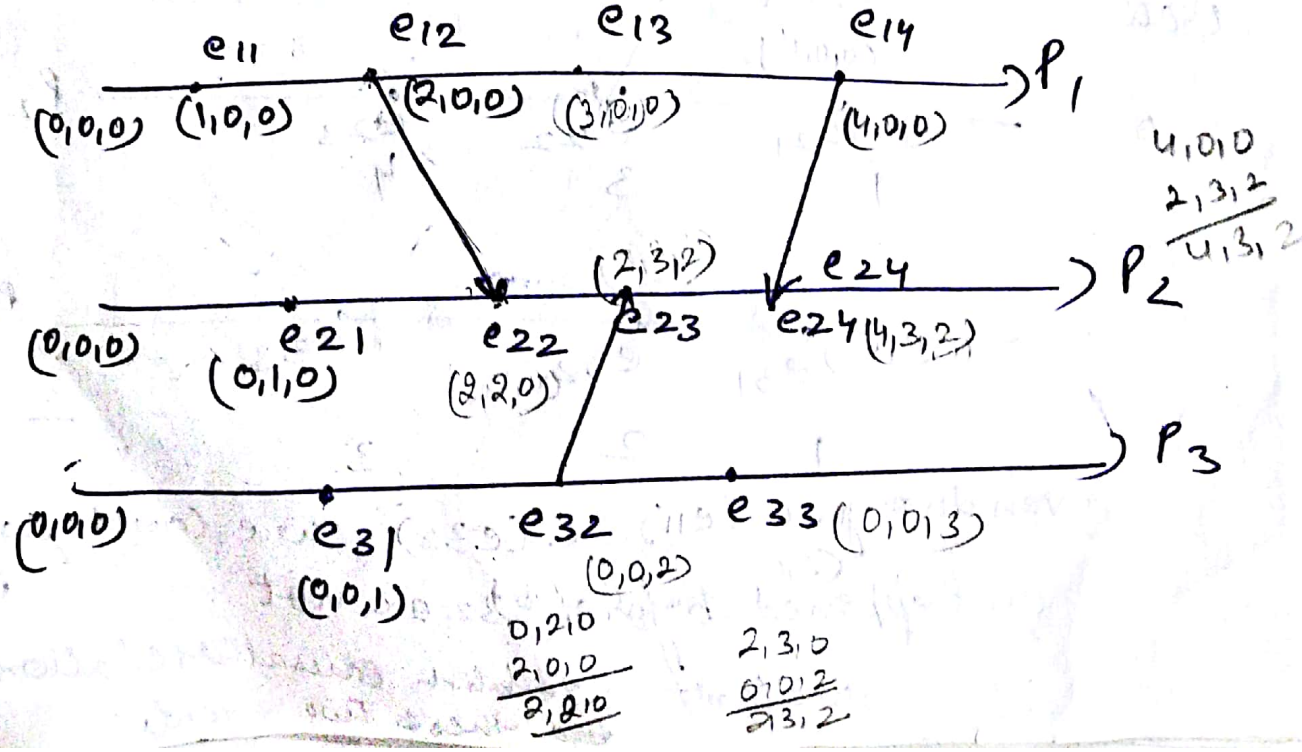
- $C_i[i]$ is the scalar clock value at process $P_i$
- $C_i[j != i]$ is the best guess of the scalar clock value of process $P_j$

## Rules for V.C.

R1: $C_i[i] = C_i[i] + 1$ for two successive internal events at $P_i$

R2: at process $P_i$, $C_i[j] = \max(C_i[j], tm[j])$ for all processes $P_j$, where $tm$ is the vector timestamp borne by process m received by process $P_i$ from some other process. (↓ sending)

0,2,0
2,0,0
2,2,0



## roperty of vector clock

**Total order :-** with vector clocks, we can clearly say that for any two events a & b in $P_i$ & $P_j$ respectively,

(1) if $a \rightarrow b$, then $T_a^i < T_b^j$

(2) if $T_a^i < T_b^j$ then $a \rightarrow b$

## Comparison of vector clocks

(1) Equal $(2,2,2) = (2,2,2)$ (entry match for each process)

(2) Not equal
$(2,1,2) != (2,2,2)$ (atleast one value not same).

(3) Less than or equal
$(2,1,2) \leq (2,2,2)$
left side < right side.
for every index one value.

(4) Less than
$(2,1,2) < (2,2,2)$
strictly less than
$Ta[i] < Tb[i]$

(5) Concurrent $(2,3,2) !< (3,1,2)$ for one only
$(3,1,2) !< (2,3,2)$

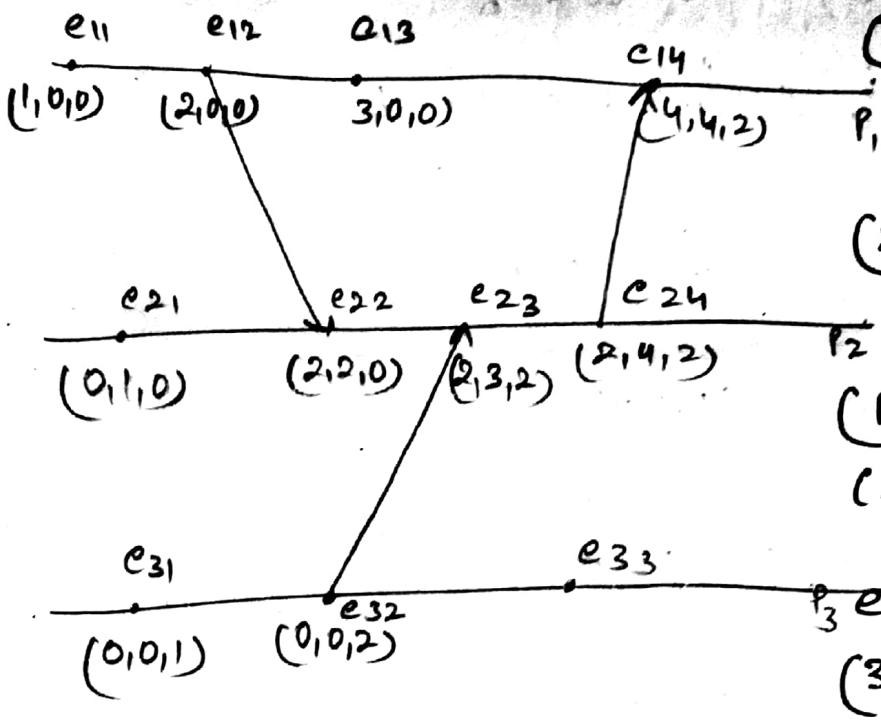
when we can't say one are less means they are $(2,3,2) \parallel (3,1,2)$ concurrent.

## Total order

with vector clocks, we can clearly s
for any two events a & b in $P_i$ & $P_j$ respec

if $a \rightarrow b$ then $T_a^i < T_b^i$ (process)

if $T_a^i < T_b^i$, then $a \rightarrow b$ → a happen before b.




$e_{31} \rightarrow e_{14}$

$(0,0,1) < (4,4,2)$

$P_1$ $e_{12} \rightarrow e_{23}$

$(2,0,0) < (2,4,2)$

$P_2$ $e_{11} \,||\, e_{33}$

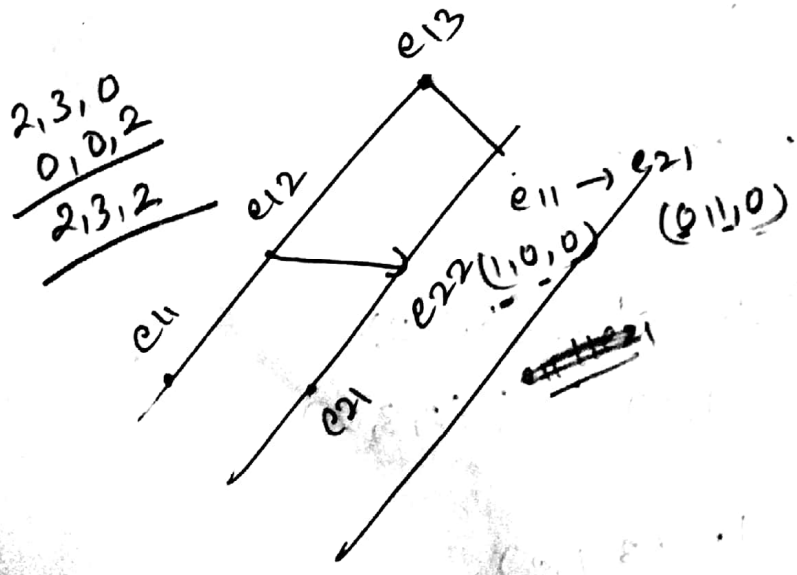
$(1,0,0) \not< (0,0,3)$

$(0,0,3) \not< (1,0,0)$

$P_3$ $e_{13} \,||\, e_{23}$

$(3,0,0) \not< (2,4,2)$

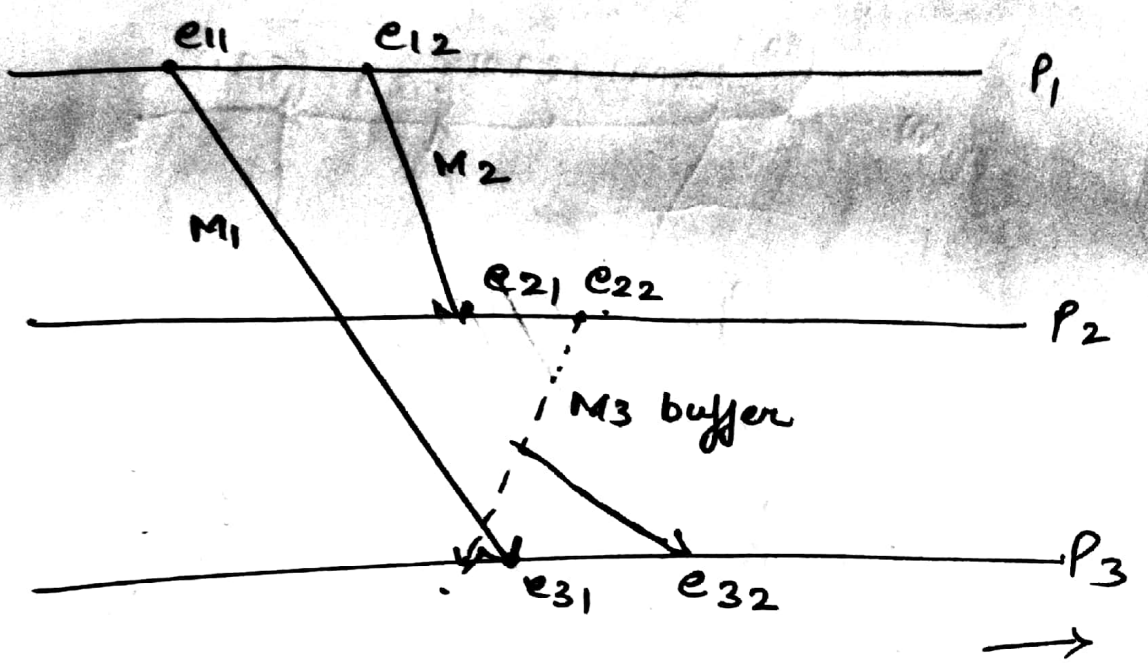
$(2,4,2) \not< (3,0,0)$




$e_{12} \rightarrow e_{22}$

$(2,0,0) < (2,2,0)$

## Causal ordering of Messages:-

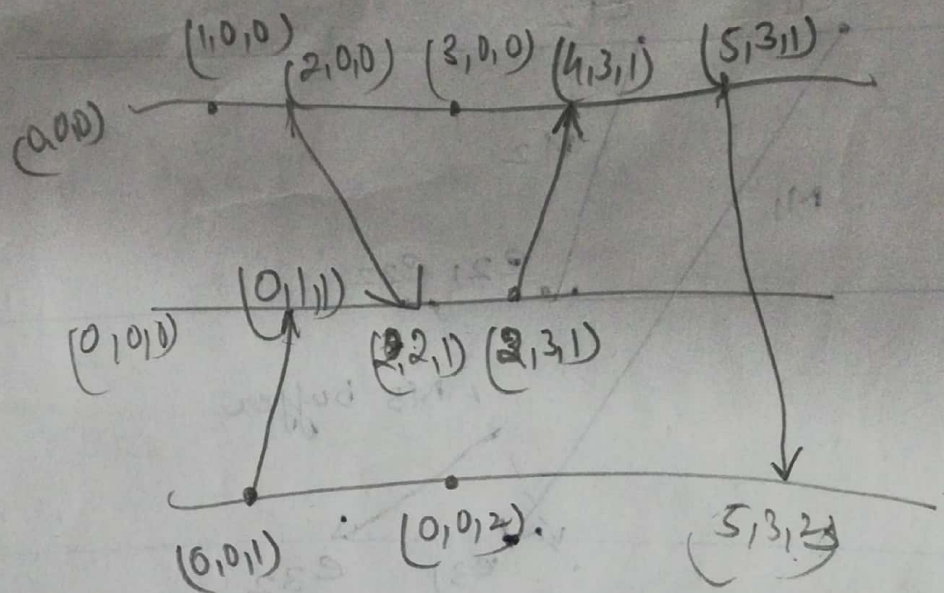



Send($M_1$) → Send($M_2$) → Receive($M_2$)
→ Send($M_3$)

So, Send($M_1$) and Send($M_3$) are causally related.

When two msg's sent to a process are causally related, they should be received in the same order.

10




5,3,1
0,0,3
5,3,2

2,3,1
4,0,0
4,3,1

0,0,1
0,1,0
0,1,1

2,0,0
0,2,1
2,2,1

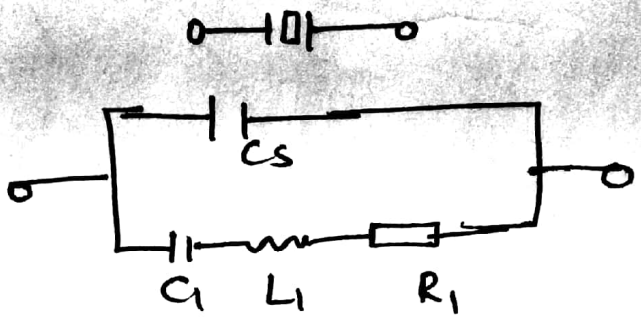
0,0,2
5,3,1
5,3,2

(11)

## RLC Circuit

## quartz Crystal

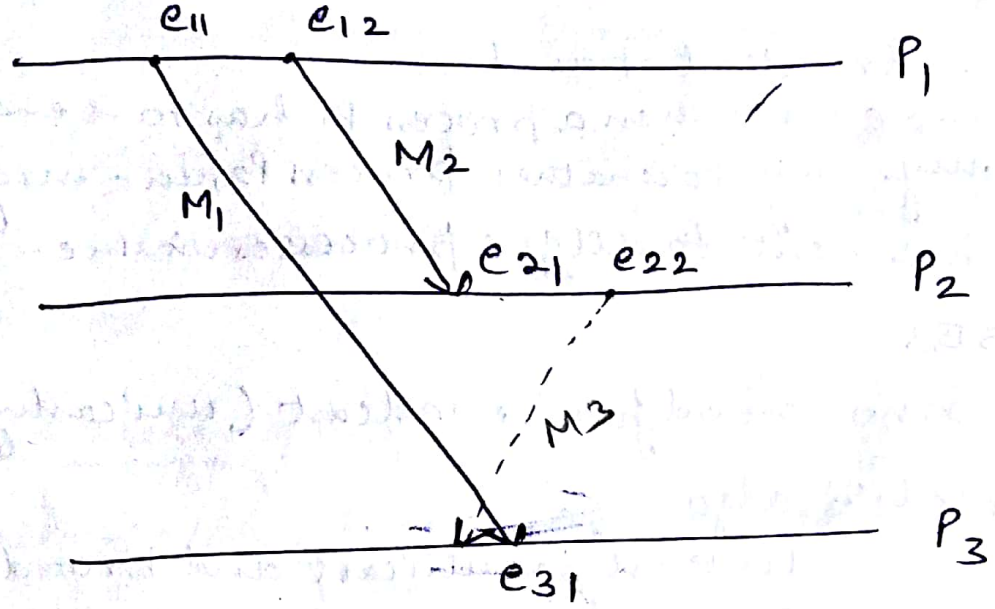
→ has a precise resonance frequency.

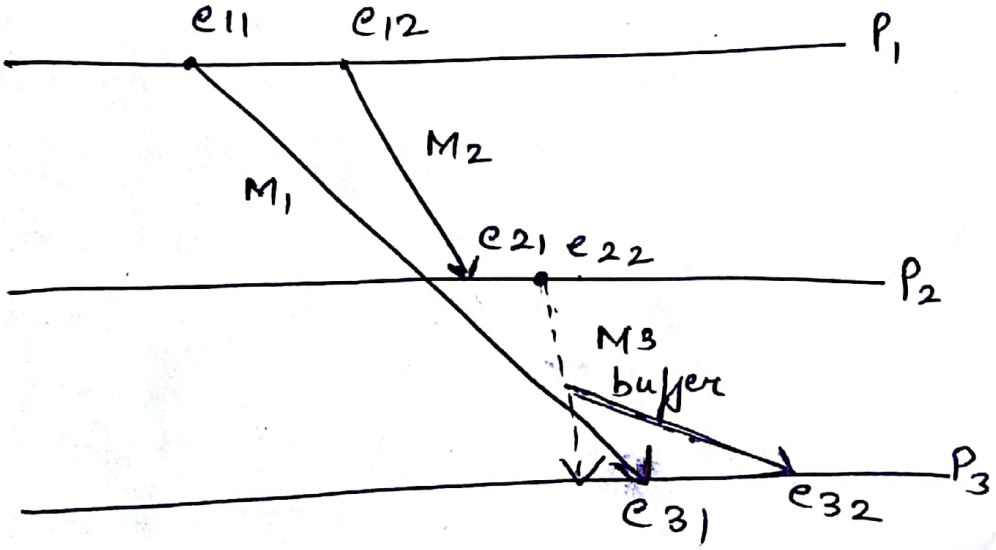



← RLC Circuit

IC mounted on motherboard:

## Causal ordering of Msg:-




Send ($M_1$) → Send ($M_2$) → Receive ($M_2$) →
=
Send ($M_3$)

So, Send ($M_1$) and Send ($M_3$) are Causally related. when two msg's sent to a process are Causally related, they should be receive in the same order.

## (Algorithm for Causal ordering of Messages)

(1) BSS

→ Broadcast-based

→ even when a process $P_1$ has to send a msg only to another process $P_2$, one msg has to be sent to all the processes in the system.

(2) SES

→ no need for broadcast (unicasting)

(3) Mattern algo

→ unicast, multicast and broadcast

Example BSS

→ no clock increment, upon receiving.

## Global State of a D.S.

① The State of a process $P_i$ ($LS_i$) is a collection of all events that have happened at the process.
- internal, msg send and msg receive

② The State of a Channel i-j ($SC_{ij}$) is a collection of all the messages that have been sent & not yet received.

③ Global State of a D.S. is a collection of the local states of all the processes and the states of all the common channels

C1: $send(m_{ij}) \in LS_i \Rightarrow m_{ij} \in SC_{ij} \oplus rec(m_{ij}) \in LS_j$ (⊕ = either ... OR ... received)

C2: $send(m_{ij}) \notin LS_i \Rightarrow m_{ij} \notin SC_{ij} \wedge rec(m_{ij}) \notin LS_j$ (∧ = and)

has to satisfy these two condition.

Consistent G.S:-
- A msg m sent by a process $P_i$ to a process $P_j$ and recorded in the local state of $P_i$ should be either recorded in the local state of receiver $P_j$ or the Channel $C_{ij}$
- A msg m sent by a process $P_i$ after recording its local state should not be recorded in the local state of receiver $P_j$ and also not in the state of Channel $C_{ij}$.

~~Strongly Consistent G.S~~

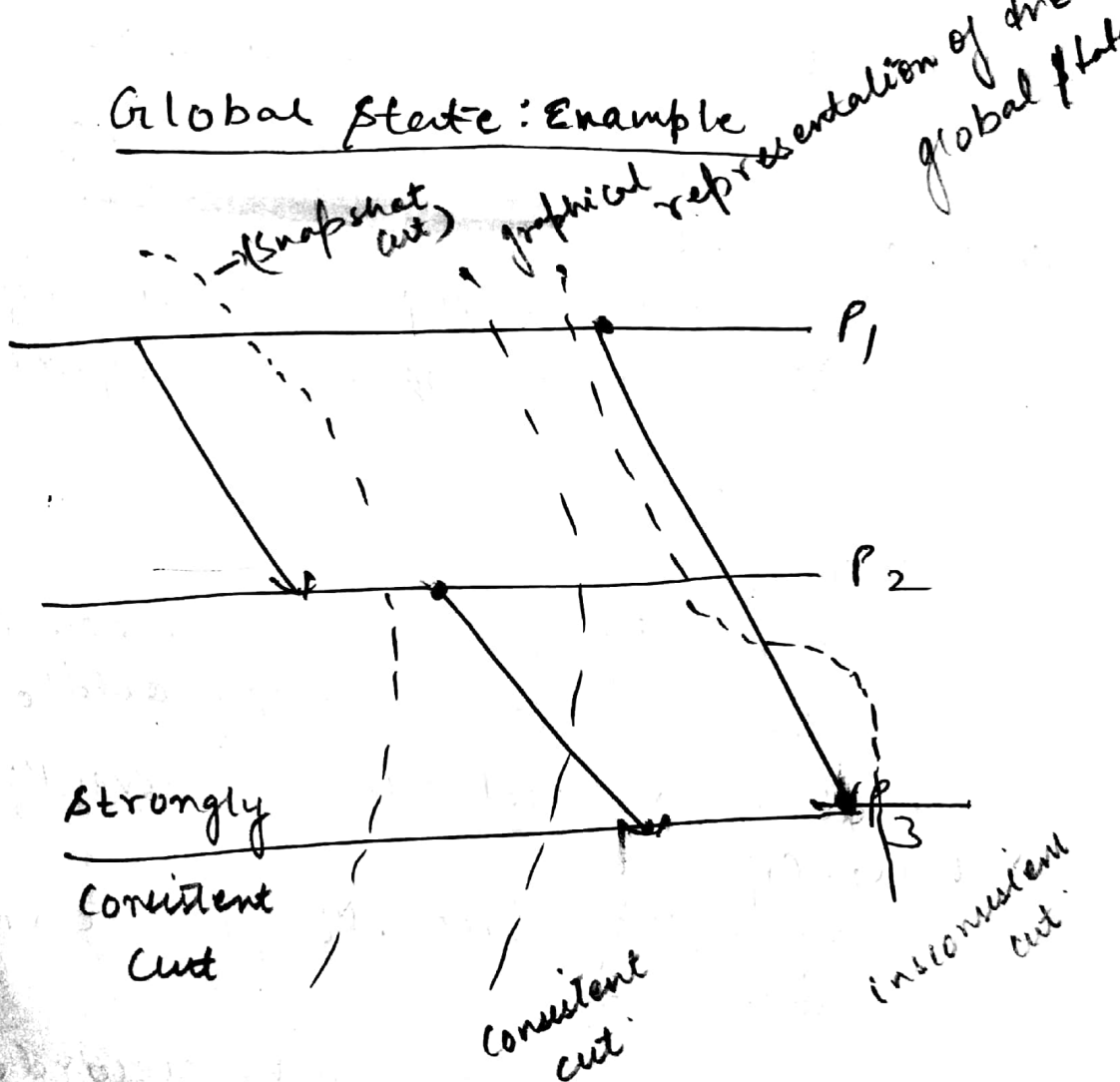
Strongly consistent G.S:-

→ A consistent G.S in which there are no msg's in transit.

Inconsistent G.S:-

→ that records the receiving of a ~~msg's and the sending of the msg~~ msg in the receiving process or as a transit message in the Channel; but the sending of the msg is not recorded.

Global State: Example




* cut is a graphical representation of a the global state

## Termination Detection :-

→ D.S consist of a set of cooperating processes which communicate with each other by exchanging messages.

→ it is important to know when the computation has terminated.

→ A D.S is globally terminated if every process is locally terminated and there is no message in transit b/w any processes.

→ Locally terminated state is a state in which a process is locally terminated & there is no message.

→ Locally terminated state is a state in which a process has finished its computation & will not restart any action unless it receives a msg.

→ termination detection algorithm is used for this purpose.

→ Msg in computation = Basic msg.
Msg in T.D = Control msg.
(Termination Detection)

## System Model:

At a given time process can be in

(active)
- local computation is going on.
- can send messages

(idle)
- where the process has finished the execution of its local computation. & will be reactivated only on the receipt of a msg from another process.

* A computation is said to have terminated if and only if all the processes are idle and there is no msg in transit.

**Basic Idea:** one of the cooperating processes monitors the computation and is called the controlling agent.

Huang's Termination Detection Algo