INTRODUCTION ->

Here's a brief explanation of each topic:

1. Computer Architectures:

- **Tightly Coupled Systems**: Multiple processors share a common memory for parallel processing.
- **Loosely Coupled Systems**: Processors have their own memory and communicate via a network, used in distributed systems.

2. Distributed Computing System (DCS):

- Independent computers connected by a network, appearing as a single system where communication happens via message passing.

3. Distributed Computing System Models:

- Different models like Minicomputer, Workstation, and Processor-pool models describe how resources are shared among processors in distributed systems.

4. Factors for Emergence of DCS:

- DCS arose due to the need for distributed applications, better resource sharing, performance improvements, and scalability.

5. Distributed Operating Systems:

- **Network OS**: Users see a group of computers.
- **Distributed OS**: Appears as one system, managing tasks across different computers dynamically with higher fault tolerance.

6. System Issues:

- **Transparency**: Makes multiple computers appear as one system.
- **Reliability**: Achieved through fault tolerance and recovery mechanisms.
- **Scalability**: Systems are designed to handle growth without central bottlenecks.

7. Security:

- Distributed systems face security challenges due to decentralized control, addressed using encryption and secure communication protocols.

8. Kernel Models:

- **Monolithic Kernel**: Includes most OS services within one large kernel, offering speed but limited flexibility.
- **Microkernel**: Minimalist kernel with modular design, making it more flexible but slightly slower due to message passing between components.

MESSAGE PASSING

Here’s a more detailed breakdown of the key topics from the **"Message Passing"** PDF for better exam preparation:

1. Communication Primitives:

- **Synchronous Systems**: Communication and processing are guaranteed to happen within a specific time frame. The sender knows if the receiver has failed if no acknowledgment is received within the expected time.
- **Asynchronous Systems**: There are no time guarantees for communication or processing. Delays can occur indefinitely, making it hard to distinguish between a slow processor and a failed one.

2. Message Passing:

- **Shared Data Approach**: Information is placed in a shared memory area accessible to all processes.
- **Message Passing Approach**: Information is copied from the sender’s address space to the receiver's. This method supports process isolation.

3. Message Passing System (MPS):

- Provides protocols for IPC, MPS hides complex network protocol details and ensures compatibility across platforms. It supports building higher-level systems like **Remote Procedure Call (RPC)** and **Distributed Shared Memory (DSM)**.

4. Synchronization:

- **Blocking Send/Receive**: The sender or receiver process pauses until an acknowledgment or message is received.
- **Non-blocking Send/Receive**: The sender or receiver process continues execution without waiting for acknowledgment or message delivery, reducing waiting time but requiring polling or interrupt mechanisms for message handling.

5. Buffering:

- **Null Buffering**: No message buffering. The sender waits for the receiver to be ready. If the receiver isn’t ready, the message must be retransmitted.
- **Single-Message Buffer**: A buffer with the capacity to store only one message at a time, useful in synchronous communication.
- **Unbounded Capacity Buffer**: Used in asynchronous communication. It stores unreceived messages but can lead to buffer overflow issues.

6. Message Delivery:

- **Explicit Addressing**: The sender directly names the process with which it wants to communicate.
- **Implicit Addressing**: The sender doesn't specify a process but instead refers to a service that many processes might provide.

7. Reliability in Communication:

- **Four-Message Reliable IPC Protocol**: Sender waits for acknowledgment and reply, ensuring reliable delivery.
- **Three-Message Reliable IPC Protocol**: A reply from the receiver doubles as an acknowledgment, reducing the communication overhead.
- **Two-Message Reliable IPC Protocol**: The receiver's acknowledgment is implicit if the message is processed before the timer expires.

8. Fault-Tolerant Communication:

- This handles message losses, failed requests, or system crashes by using retransmissions and acknowledgment mechanisms to ensure message delivery.
- **Idempotency**: Operations are designed to produce the same result if repeated, essential for handling duplicate requests.

9. Multicasting:

- **Atomic Multicast**: Ensures that messages are either delivered to all members of a group or to none, maintaining consistency.
- **Reliability in Multicasting**: Different levels of reliability, such as 0-reliable (no acknowledgment) or all-reliable (all recipients acknowledge message reception).

10. Ordered Message Delivery:

- **Absolute Ordering**: All messages are delivered in the exact order they were sent.
- **Consistent Ordering**: Messages are delivered in the same order to all receivers, though this order may differ from the sending order.
- **Causal Ordering**: Ensures messages are delivered in a causally related order. Messages dependent on one another are delivered in the correct sequence.

11. Group Communication:

- **One-to-Many**: Single sender communicates with multiple receivers (multicast).
- **Many-to-One**: Multiple senders communicate with a single receiver.
- **Many-to-Many**: Multiple senders communicate with multiple receivers, requiring mechanisms to handle message sequencing and delivery order.

By understanding these key points, you'll be prepared to answer questions about message passing, synchronization, buffering, reliability, and multicasting in distributed systems.

Message Passing - Blocking & Non-Blocking API

Here is a summary of the **"Message Passing - Blocking & Non-Blocking API"** PDF for exam purposes:

1. Synchronous vs Asynchronous API:

- **Synchronous (Blocking)**: The thread waits (blocks) until the task, like reading data from a network, is complete. The system goes into a sleep state until the data is returned.
- **Asynchronous (Non-blocking)**: The thread continues execution without waiting. The system uses an event-driven model or callbacks to handle data once it's available.

2. Why Non-blocking I/O?:

- Reduces the number of threads needed, saving resources (like memory for each thread).
- Prevents bottlenecks in handling large numbers of I/O-bound requests (e.g., network or file I/O).

3. Blocking I/O Types:

- **CPU-bound Blocking**: The CPU is busy with tasks, causing delays.
- **I/O-bound Blocking**: Waiting for data from external sources like networks or disks, causing a pause.

4. Event Loop in Non-blocking I/O:

- Uses an infinite loop that checks for events (e.g., data availability) using system-level APIs like **epoll** (Linux) or **kqueue** (BSD). The event loop optimizes I/O handling and scales to many connections.

5. Server Architectures:

- **Thread-based**: Uses multiple threads, each handling a connection, which can lead to overhead from context switching.
- **Event-driven**: Uses a single thread with an event loop that handles events (like I/O operations) sequentially, minimizing resource usage.

6. Key Optimizations:

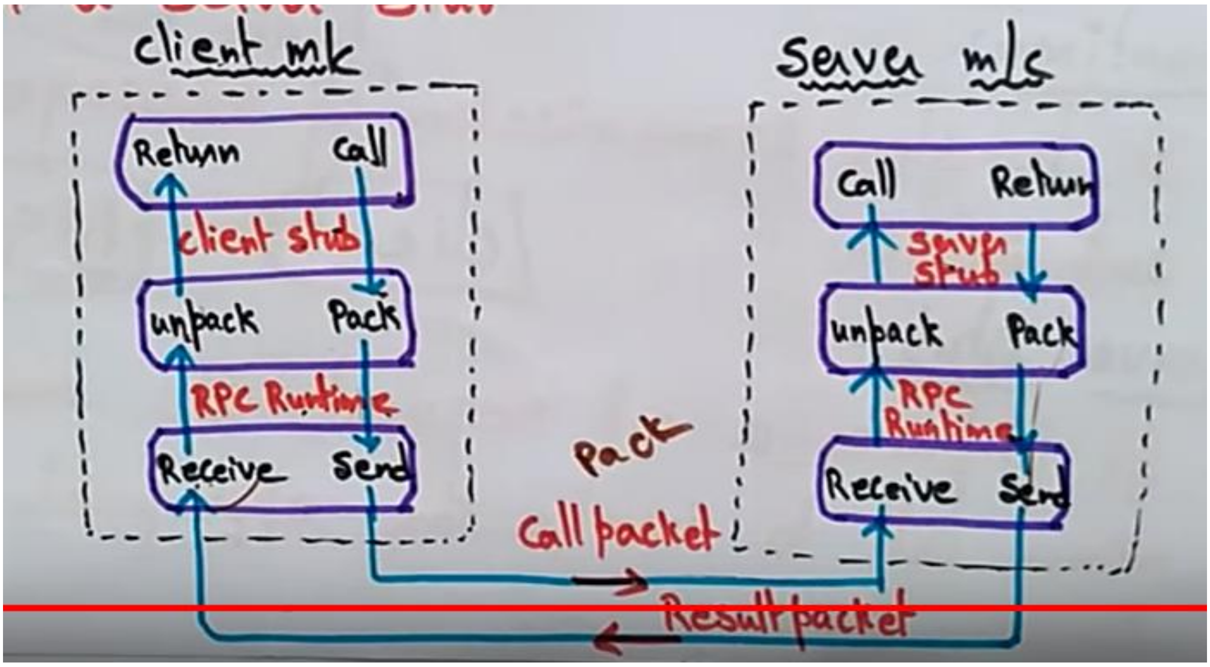
- Event-driven architectures are enhanced with kernel-level APIs like **epoll**, **io_uring**, and **IOCP**, allowing efficient management of large numbers of file descriptors (e.g., network connections).

7. Callbacks and Performance:

- Non-blocking I/O relies on callbacks, which are functions that get called when data is available. This reduces idle time but makes debugging and control flow harder.

In short, non-blocking I/O improves server performance by handling multiple connections with fewer resources, using event-driven models instead of thread-per-connection methods.

Remote Procedure Calls (RPC)



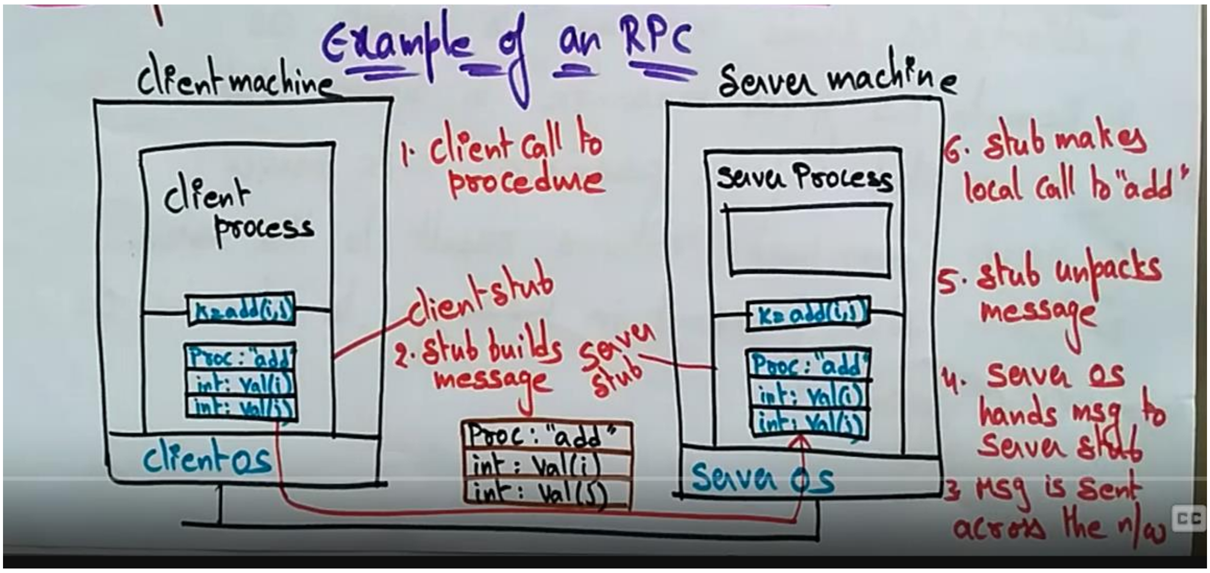
Steps of a Remote Procedure Call :-

1. client procedure calls client stub in normal way
2. client stub builds message, calls local OS
3. client's OS sends message to remote OS
4. Remote OS gives message to server stub
5. Server stub unpacks parameters, calls server
6. Server does work, returns result to the stub
7. Server stub packs it in message, ~~to~~ client's OS calls local OS.

Server's OS sends message to client's OS

client's OS gives message to client stub

0. stub unpacks result, return to client.



Implementation issue with RPC

i)Connectionless Protocol – TCP used over lan client is bound to server and connection is established b/w them

ii)Connectionless Protocol – IP and UDP are easy to use and easily fit with Unix and network such as internet.

iii)Message Size – break the message into parts and use stop and wait or Go back n or selective repeat to send data into segments.

1. Introduction to RPC:

- RPC is a communication paradigm used in distributed systems to facilitate interaction between processes on different machines.
- It mimics local procedure calls but involves a network, making it more complex in handling communication, failures, and address spaces.

2. RPC Model:

- RPC follows a standard procedure call model:
    1. Caller passes arguments.
    2. The procedure is executed.
    3. The result is returned.
- The critical difference is that the called procedure may reside in a different address space, often on another machine, requiring message passing between caller and callee.

3. Transparency in RPC:

- **Syntactic Transparency**: RPC should resemble local procedure calls in syntax.

- **Semantic Transparency**: It's challenging to achieve, as network failures and delays make RPCs less predictable than local calls.

4. Differences between RPCs and Local Procedure Calls:

- **Disjoint Address Spaces**: RPC operates between processes in separate memory spaces.
- **Failure Handling**: RPCs are more prone to failure (network and machine failures).
- **Performance**: RPC is slower due to network overhead compared to local procedure calls.

5. Implementing RPC:

- **Stubs**: Act as proxies for the client and server to hide the complexity of network communication.
    - **Client Stub**: Sends procedure requests to the server.
    - **Server Stub**: Unpacks requests, calls the procedure, and sends the result back.
- **RPC Runtime**: Manages the communication (retransmission, encryption, etc.).

6. RPC Message Handling:

- **Call Messages**: Include procedure identifiers, arguments, and client identification.
- **Reply Messages**: Return results or indicate failure.

7. Marshalling and Unmarshalling:

- **Marshalling**: The process of packaging data (arguments and results) into a format that can be transmitted over the network.
- **Unmarshalling**: Decoding the data upon arrival.

8. Server Management:

- **Stateful Servers**: Maintain the state between calls, allowing for easier interaction but are less robust in case of failures.
- **Stateless Servers**: Handle each request independently, making them more fault-tolerant but less efficient.

9. Parameter Passing:

- **Call-by-Value**: Common in RPC; arguments are copied and sent over the network.
- **Call-by-Reference**: Harder to implement since the client and server do not share memory.
- **Call-by-Move/Object-Reference**: Advanced techniques for optimizing network data transfer.

10. Call Semantics: how a system handles the execution of a procedure call, especially in the face of potential failures

Call Semantics	Guarantee	Behavior	Use Case
Possibly or May-Be	No guarantees	Call may not be executed or no response is sent.	Periodic updates, low-priority tasks
At-Least-Once	Call is executed at least once	Client retransmits until a response is received, causing potential duplicates.	Idempotent operations like reading
At-Most-Once	Call is executed at most once	Ensures no duplicate execution; may not execute at all in case of failure.	Non-idempotent operations like transactions
Exactly-Once	Call is executed exactly once	Ideal guarantee, but hard to achieve in practice.	Critical operations like payments
Last-One	Only the last call's result is used	Retransmits calls, but only last successful call's result matters.	State updates, session management
Last-of-Many	Latest call's valid result is used	Handles multiple call requests, ignores orphans, ensures latest valid	Handling retries while avoiding orphans

11. Security Concerns:

- **Authentication**: The client and server must authenticate each other to prevent unauthorized access.
- **Data Security**: Sensitive data needs to be encrypted to prevent interception.

12. Special RPC Types:

- **Callback RPC**: Allows the server to call back the client, creating a peer-to-peer interaction model.
- **Broadcast RPC**: Sends a request to all available servers and processes the first reply.
- **Batch-mode RPC**: Sends multiple requests in one go to reduce network overhead.

13. Lightweight RPC (LRPC):

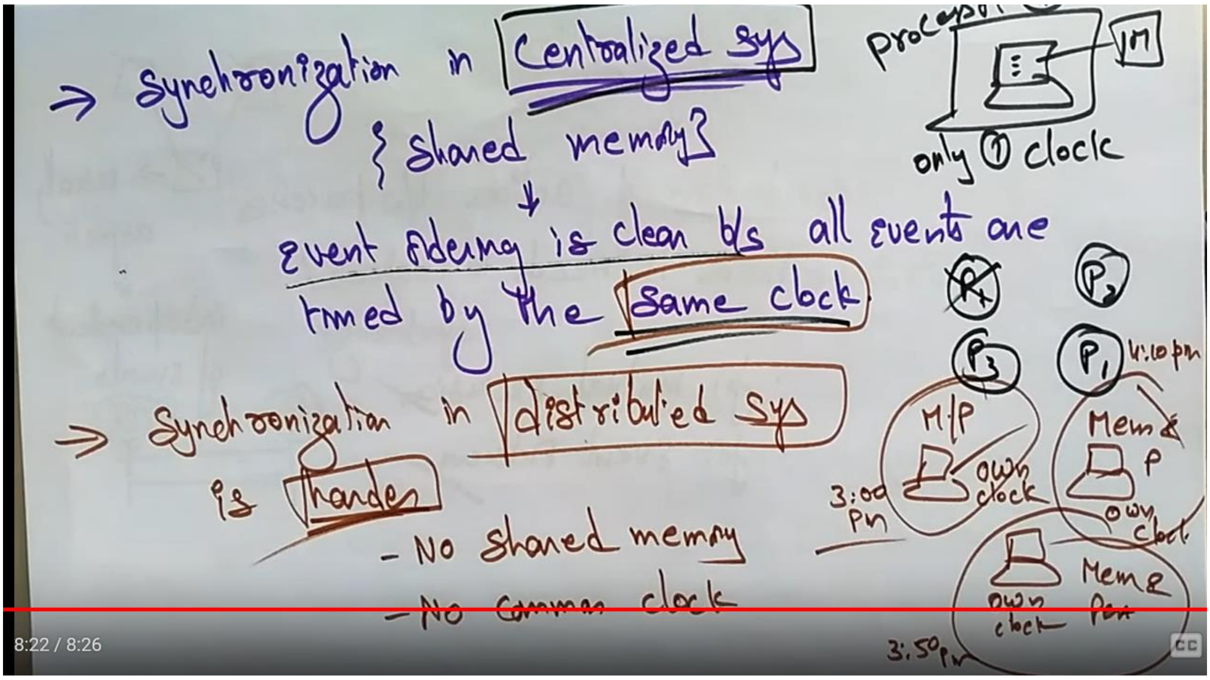
- **LRPC**: Optimized for cross-domain communication within the same machine, reducing overhead compared to traditional RPC.

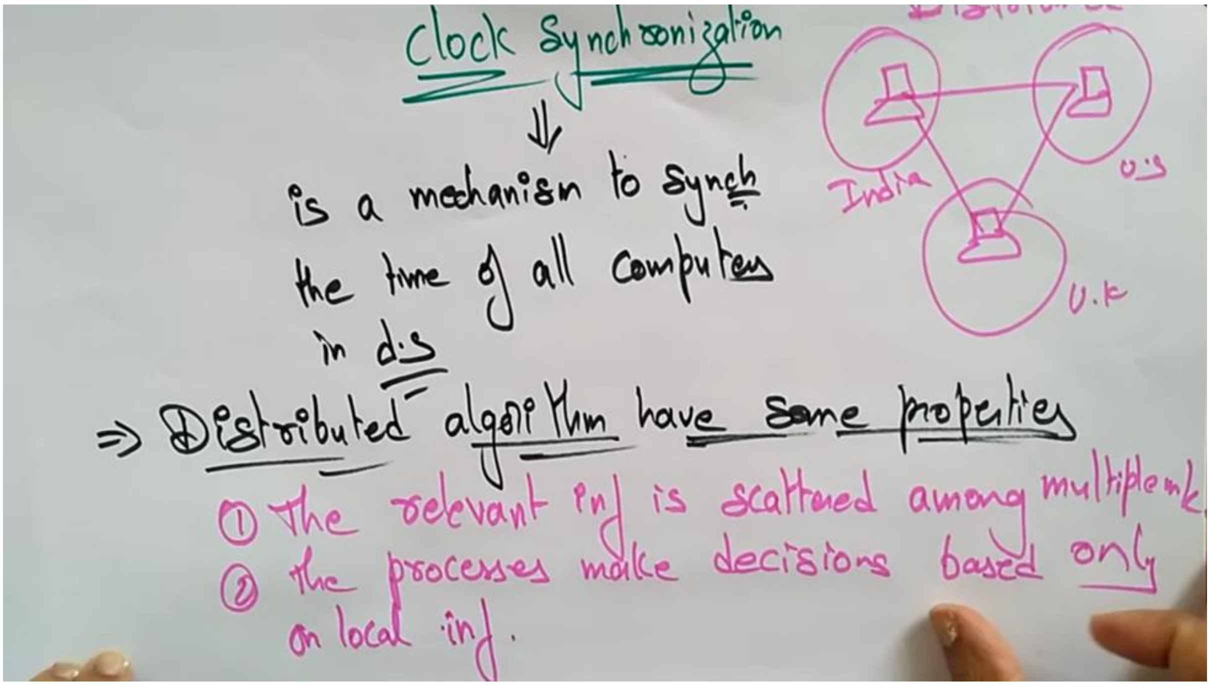
14. Optimizations:

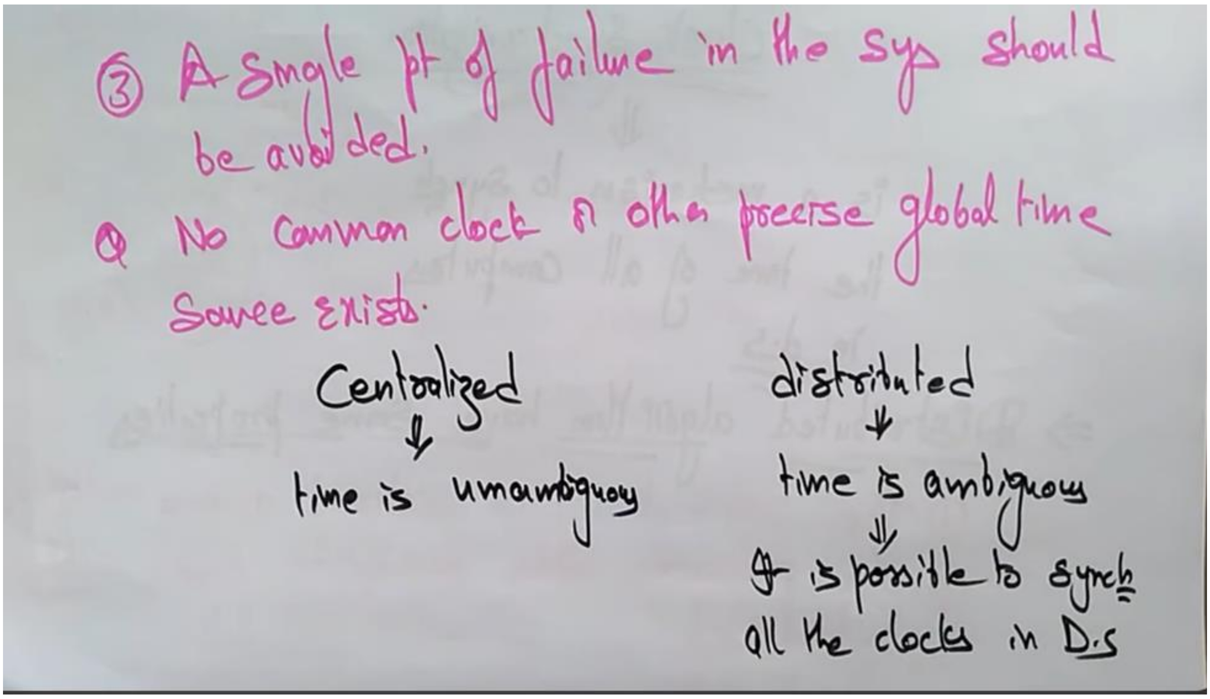
- Efficient handling of data transfer, concurrency, and reducing communication overhead are key to improving RPC performance.

These are the major points you should focus on for an exam. The document covers both fundamental concepts and advanced topics like server management, parameter passing, and special types of RPCs, all of which are essential for understanding distributed systems and RPC mechanisms.

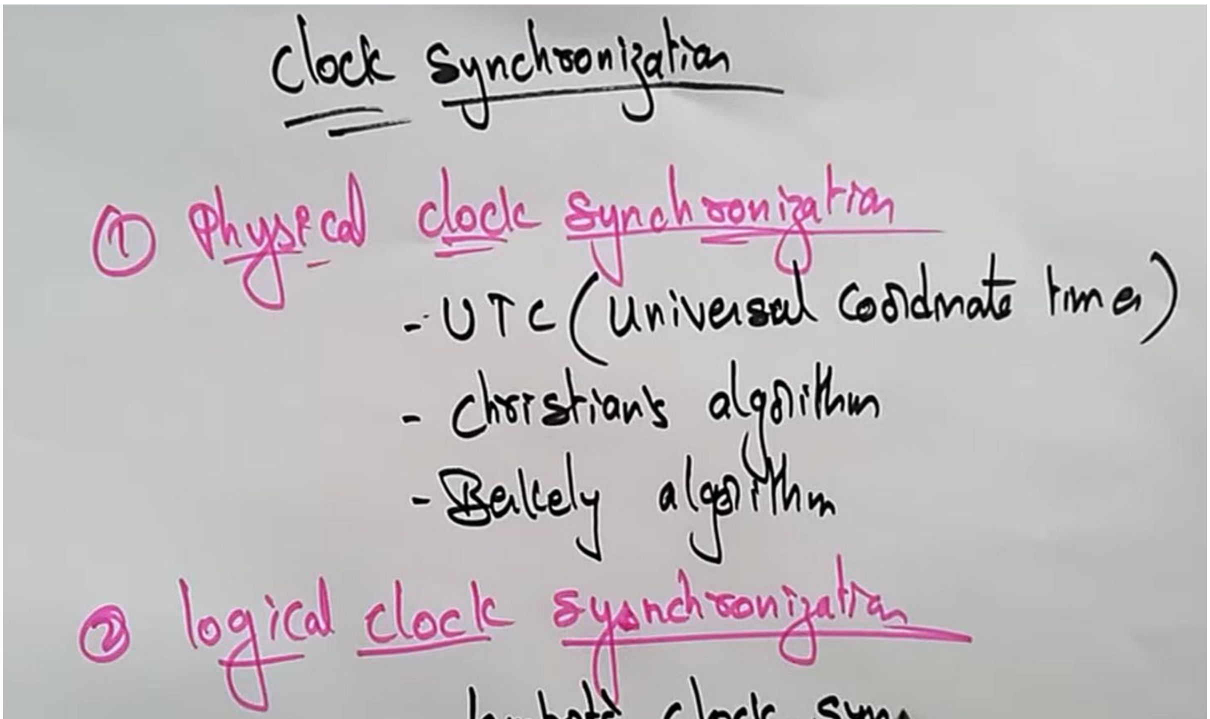
SYNCHRONIZATION







Types of clock synchronization



③ Mutual exclusion clock syn :-

- Lamport's algorithm
- Centralized algorithm
- Distributed algorithm
- token based algorithm.

Physical $\boxed{\text{clock}}$ ⇒ is an electronic device that $\boxed{\text{counts}}$ oscillations in a crystal at a particular freq.

can be used to time stamp an event on that comp

Stored in Counter reg

$E_1 - t_1$

$E_2 - t_2$

✱ Many applications are interested only in the order of the events not the exact time of day at which they occured.

✱ The time diff b/w 2 computers is known as ~~Drift~~

Drift

✱ The clock drift over the time is known as Skew

Several methods are used to attempt the synchronization of physical clocks in ds

- UTC
- Christian's alg
- ~~Ber~~ Berkely alg

① Coordinated Universal Time (UTC)

⇒ All computers are generally synchronized to a standard time called Coordinated Universal Time (UTC).

✲ UTC is the primary time standard by which the world regulates clocks and time. It is available via radio signal, telephone line, satellite (GPS)

⇒ UTC is broadcasted via the satellites

✲ UTC broadcasting service provides an accuracy of 0.5 msec

⇒ Computer servers and online services with UTC receivers can be synchronized by satellite broadcasts.

the (Exact time)

Problem :- Sometimes we simply need the exact time, not just an ordering.

Solution :- Universal Coordinated Time (UTC) :

✲ Based on the no. of transitions per second of the cesium 133 atom (pretty accurate).

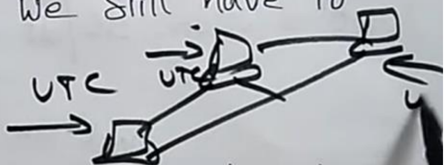
✲ At present, the real time is taken as the average of some 50 cesium-clocks around the world.

✲ Introduces a ~~lead~~ leap second from time to time to

⇒ UTC is broadcast through short wave

✲ Satellites can given an accuracy of about ±0.5ms

Problem :- Suppose we have a distributed system with a UTC-receiver somewhere in it ⇒ we still have to distribute its time to each m/c.

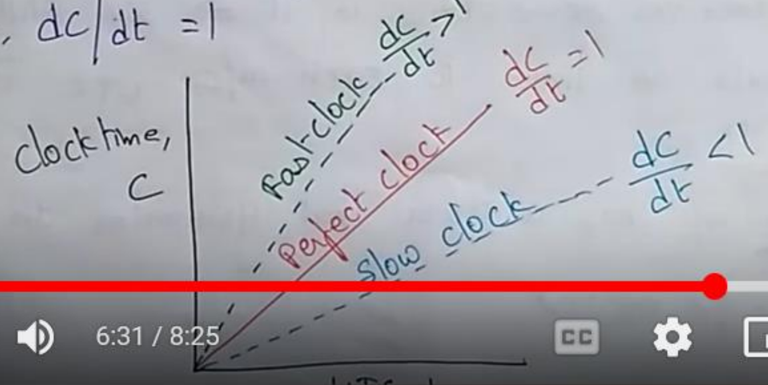


Basic principle :-

- Every m/c has a timer that generates an interrupt H times per second.

- There is a clock in m/c p that ticks on each timer interrupt. Denote the value of that clock by $C_p(t)$ where t is UTC time.

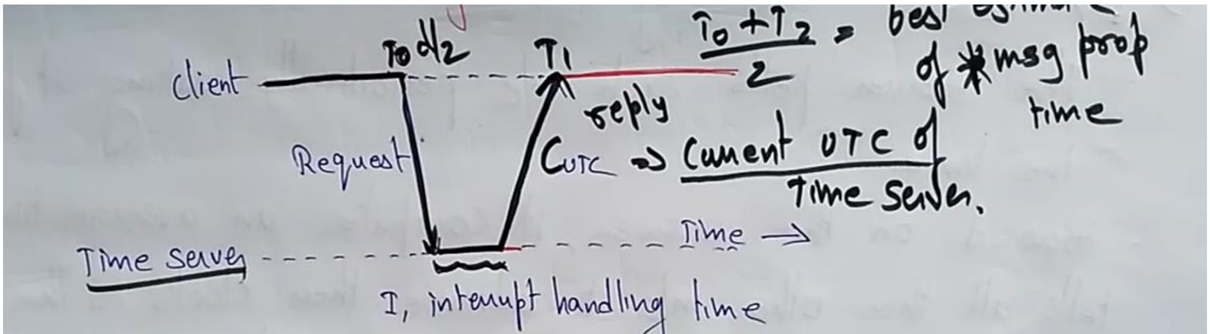
Ideally, we have that for each m/c p, $C_p(t) = t$, or in other words, $dC/dt = 1$



(2) Physical Clocks - Christian's Algorithm

- The simplest algorithm for setting time, it issues a Remote Procedure Call to time server and obtain the time.

- A m/c sends a request to time server in $\boxed{\text{"}d/2\text{" Seconds}}$ where d = max differ b/w a clock and UTC.
- The time server sends a ~~reple~~ reply with current UTC when receives the request
- The m/c measures the time delay b/w time server sending the message and m/c receiving it. Then it uses



Both T_0 & T_1 are measured with same clock

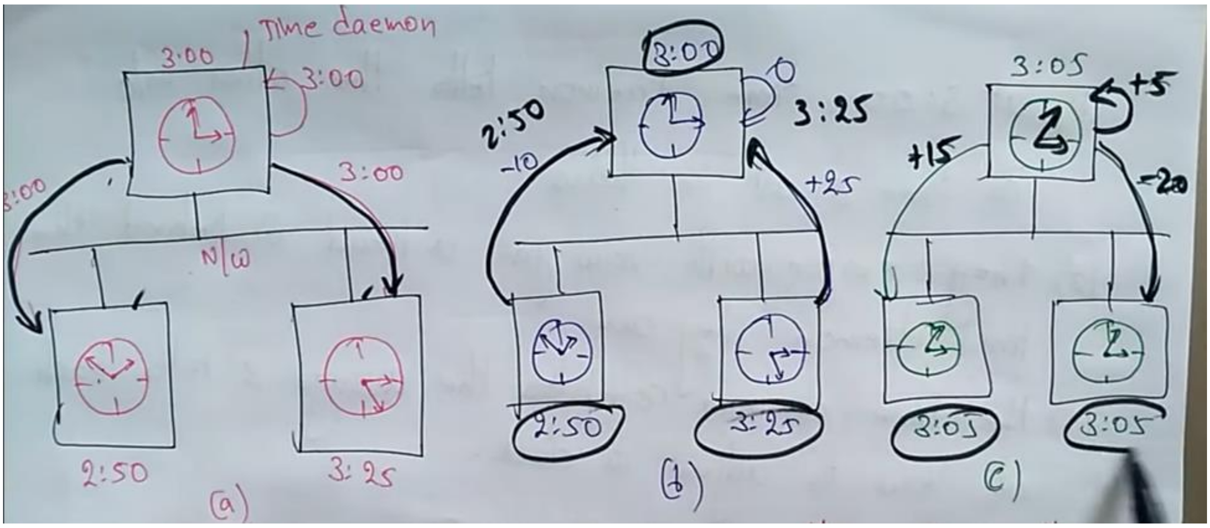
The best estimation of message propogation time = $(T_0+T_1)/2$

The new time can be set to the time returned by server plus time that elapsed since server generated the time stamp:

$$T_{new} = T_{server} + (T_0+T_1)/2$$

physical clock – Berkeley Algorithm:

- The server polls each m/c periodically, asking it for the time
- Based on the answers, it computes an average time & tells all the other m/c to advance their clocks to the new time or slow their clocks down until some specific reduction has been achieved



logical clocks

- If two m/c do not interact, there is no need to synchronize them

Synchronize

- What usually matters is that processes agree on the order in which events occur rather than the time at which they occurred

★ Absolute time is not important

★ use logical clocks

★ No concept of happened - when

Lamport's algorithm :-

Each message carries a timestamp of the sender's cloc[k]

when a message arrives :

$a \rightarrow b$ (1, 2)

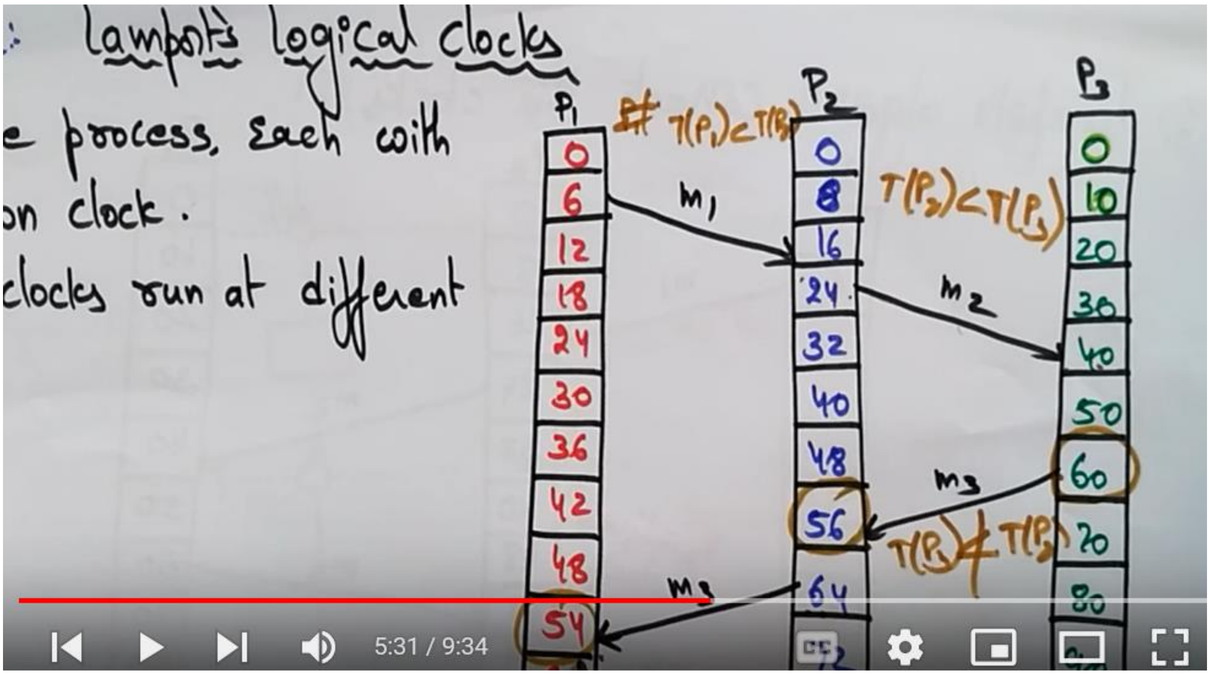
- if receiver's clock $\leq$ message timestamp $m(1)$

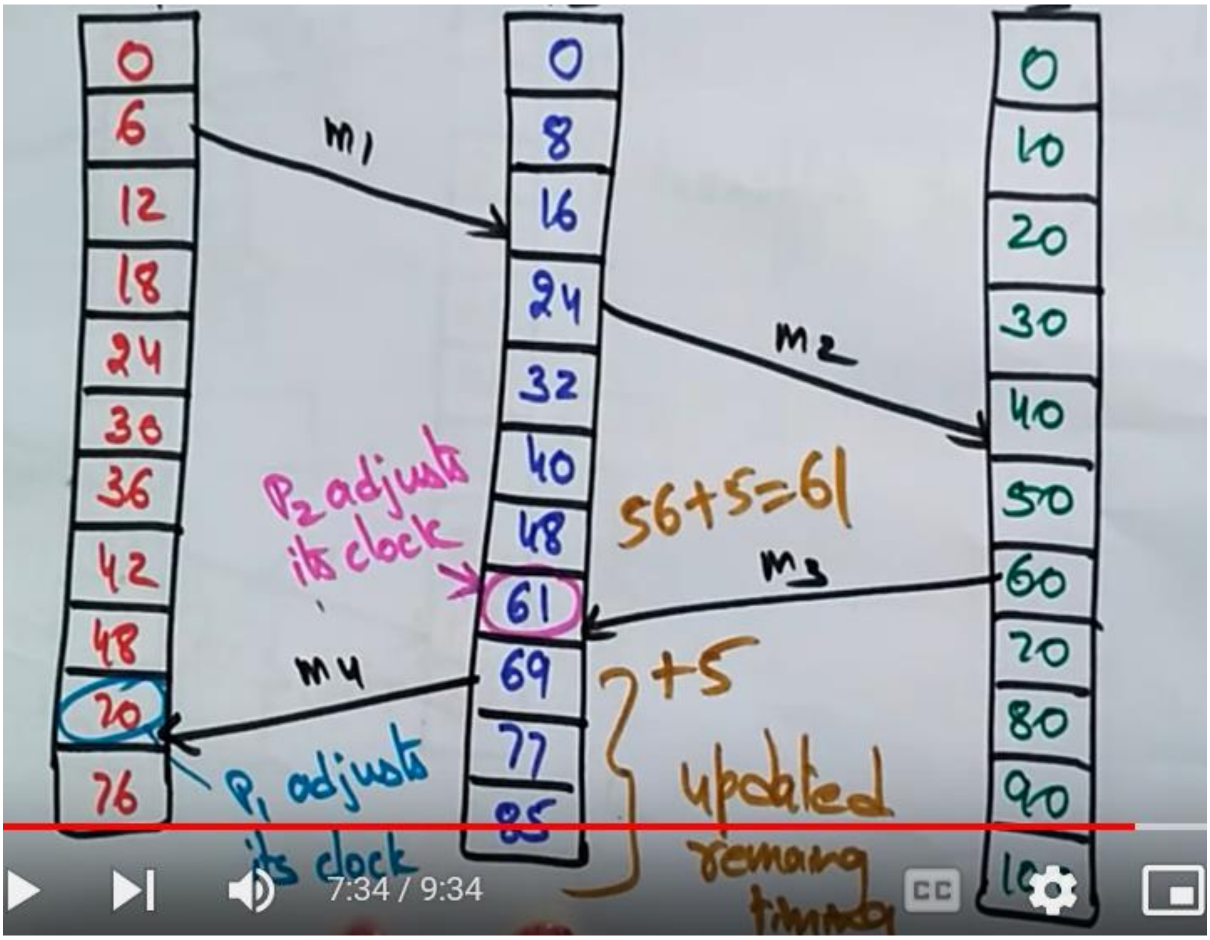
  set system clock to (message timestamp + 1)

- else do nothing

clock must be advanced between any two events in [the] same process

1:14 / 9:34


Lamport's logical clocks
e process, Each with
on clock.
clocks run at different
P_1
P_2
P_3
$T(P_1) < T(P_2)$
$T(P_2) < T(P_3)$
$T(P_3) \nless T(P_2)$
m_1
m_2
m_3
0 6 12 18 24 30 36 42 48 54
0 8 16 24 32 40 48 56 64
0 10 20 30 40 50 60 70 80
5:31 / 9:34


m_1
m_2
m_3
m_4
$56+5=61$
P_2 adjusts its clock
P_1 adjusts its clock
+5
updated
0 6 12 18 24 30 36 42 48 70 76
0 8 16 24 32 40 48 61 69 77 85
0 10 20 30 40 50 60 70 80 90
7:34 / 9:34

e different algorithms based on message passing to imple
utual exclusion in distributed systems are

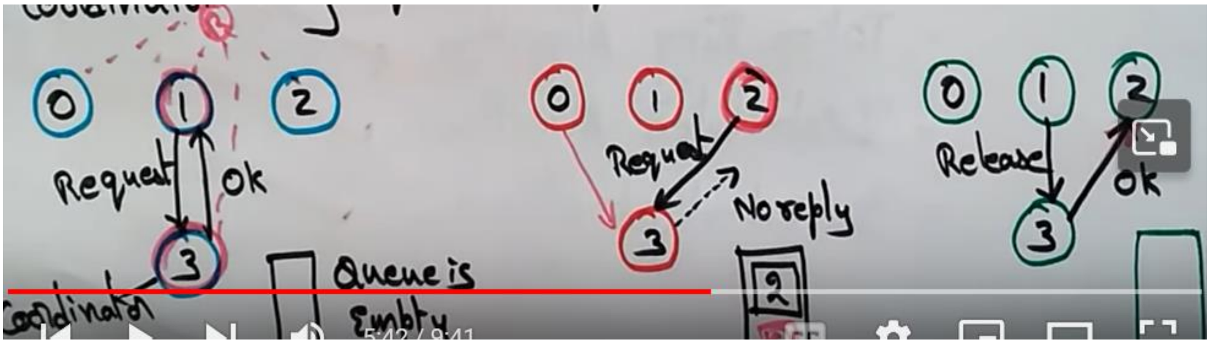
- Centralized Algorithm
- Token Ring Algorithm

→ Centralized Algorithm :

→ one process is elected as the coordinator

→ When ever a process wants to access a shared resource, it sends request to the coordinator to ask for permission

→ Coordinator may queue requests



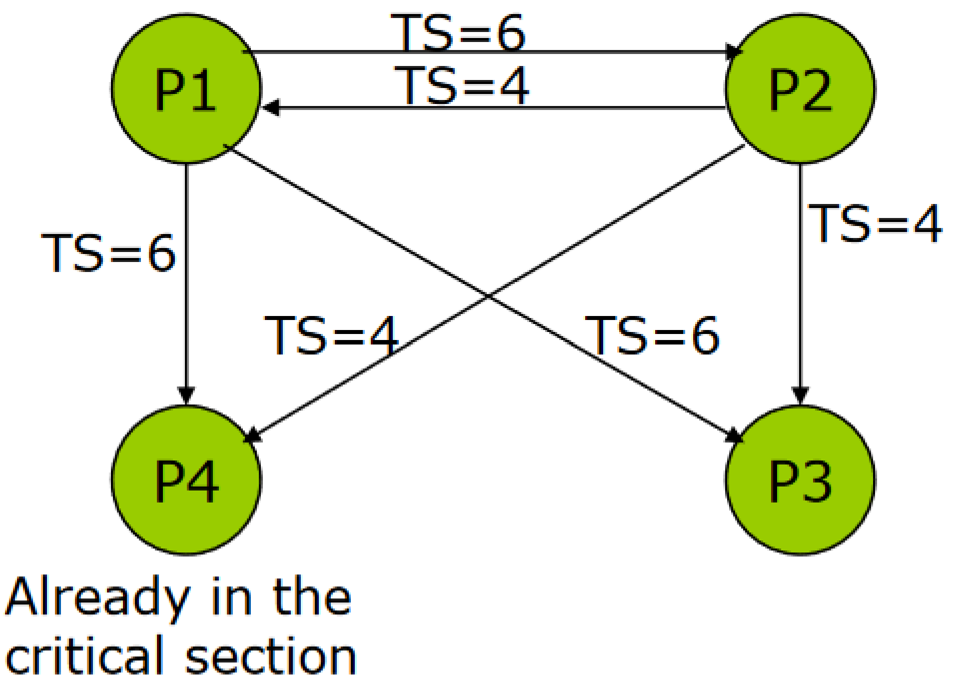
System throughput S (rate at which the system executes requests for the CS)

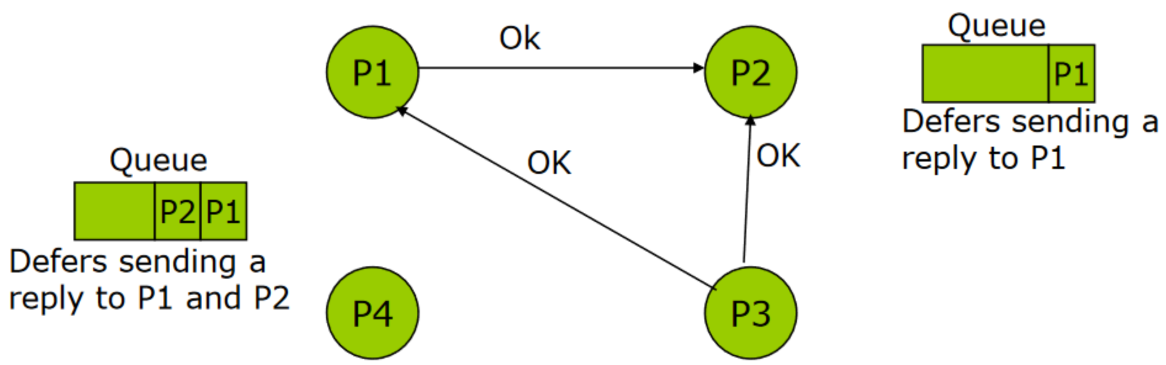
$$\boxed{S = \frac{1}{sd + E}}$$

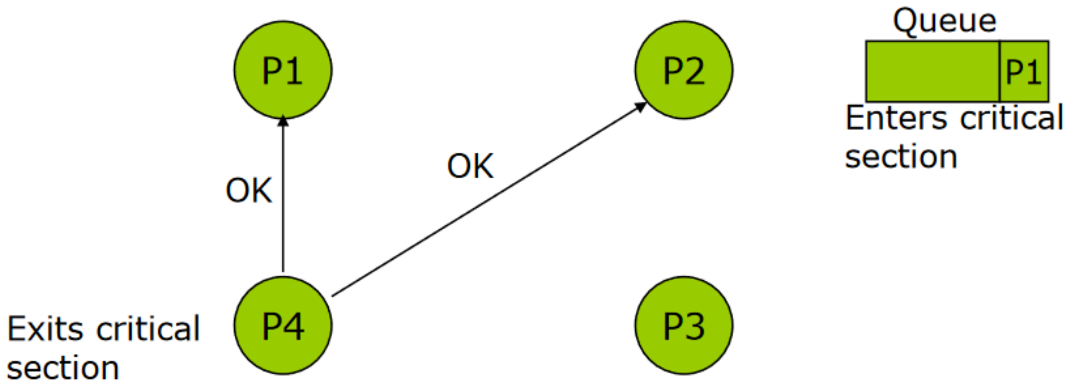
sd = synchronization delay

E = average execution time

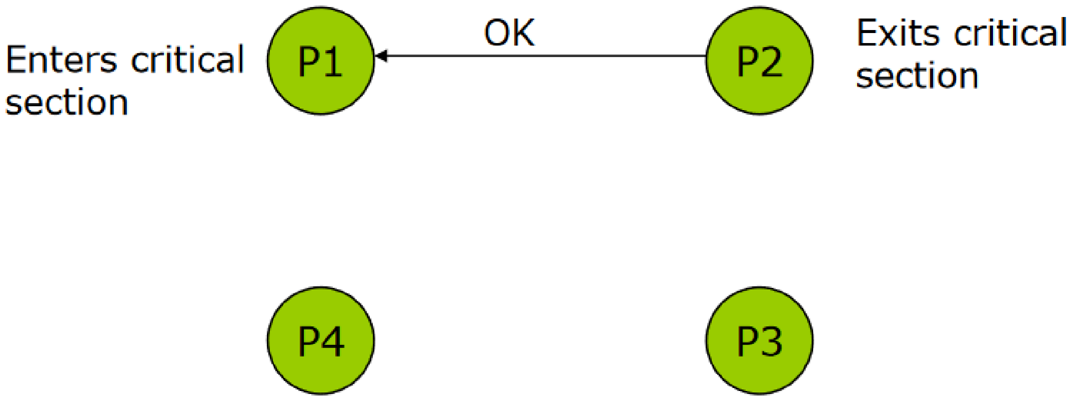
Ricart and Agrawala Algorithm







Queue
P1
Enters critical section
P1
P2
OK
OK
Exits critical section
P4
P3



Enters critical section
P1
OK
P2
Exits critical section
P4
P3