MODULE 3: Synchronisation - Logical, Physical, election algorithms, mutual exclusion

Synchronisation (8 Marks)

Understanding Synchronization (2 Marks)

- In distributed systems, multiple processes can access and modify shared resources concurrently.
- Synchronization ensures that these accesses happen in a controlled manner, preventing race conditions and data corruption.
- It establishes a coordinated order for accessing shared resources, guaranteeing consistency and integrity of data.

Physical vs. Logical Clocks (2 Marks)

- **Physical Clocks:** Real-time devices on individual machines that keep track of time.
    - Limitations:
        - **Drift:** Physical clocks may drift due to hardware variations or software inconsistencies, causing inaccurate timekeeping and unreliable coordination in distributed systems.
- **Logical Clocks:** Maintain a logical ordering of events across processes, independent of physical clock discrepancies.
    - Key features:
        - **Abstraction:** Provide a consistent view of time across the system, regardless of physical clock variations.
        - **Partial Ordering:** Ensure events from a single process occur in the order they were sent, and enable partial ordering of events from different processes based on timestamp comparisons.
    - Common examples:

Lamport Timestamps

Concept: Lamport timestamps assign monotonically increasing timestamps to events based on a process's local clock and received messages. This helps

establish a partial order of events across processes in a distributed system, even if physical clocks are not perfectly synchronized.

Key Points:

- **Monotonic Increment:** Each event at a process receives a timestamp greater than all its previous timestamps.
- **Message Inclusion:** When a message is sent, its timestamp is piggybacked on the message. When a message is received, the receiving process updates its local clock to the maximum of its current local clock and the received timestamp, then increments it by 1. This ensures events from a single process are ordered correctly and events related to received messages are ordered after the message reception.

Explanation with Example:

Consider processes P1, P2, and P3 with Lamport timestamps:

- P1: T1 = 1
- P2: T2 = 1
- P3: T3 = 1

1. P1 sends a message (M1) to P2. P1 updates its timestamp to T1' = max(T1, T1) + 1 = 2 and adds T1' to M1.
2. P2 receives M1. P2 updates its local clock to T2' = max(T2, T1') + 1 = 3.

Now, the timestamps reflect the order of events:

- P1: T1 (event 1), T2' (sending M1)
- P2: T2 (event 1), T3' (receiving M1)

Partial Ordering: While Lamport timestamps can establish a partial order (e.g., P1's event 1 happened before P2's receiving M1), they cannot determine the order of independent events sent by different processes at the same time.

Vector Clocks

Concept: Vector clocks assign a vector of timestamps to each process, where each element represents the last event time observed from a specific process. This provides richer information about causal relationships between events across processes.

Key Points:

- Each process maintains a vector clock `V(i)`, where `i` is the process ID.
- Initially, V(i) has all elements set to 0.
- When an event occurs at process `i`, `V(i)` is incremented by 1 at the `i`th position.
- When a message is sent, the sending process's vector clock is included in the message.
- Upon receiving a message, the receiving process updates its vector clock by taking the maximum of its current vector clock and the received vector clock for each element. This ensures that the receiving process accounts for all events from other processes it may have missed.

Explanation with Example:

Consider processes P1, P2, and P3 with vector clocks:

- P1: V(1) = [1, 0, 0]
- P2: V(2) = [0, 1, 0]
- P3: V(3) = [0, 0, 1]

1. P1 sends a message (M1) to P2. P1 updates its vector clock to V(1)' = [2, 0, 0] and includes V(1)' in M1.
2. P2 receives M1. P2 updates its vector clock to V(2)' = max(V(2), V(1)'), resulting in V(2)' = [2, 1, 0].

Understanding Causal Relationships:

Vector clocks allow us to determine the causal relationships between events. In this example:

- P1's event 1 happened before P2's receiving M1 (because V(1)'[1] > V(2)'[1]).

- P2's event 1 and P3's event 1 cannot be ordered with vector clocks because they are independent (V(2)'[2] = V(3)'[2] = 0).

Comparison:

- Lamport timestamps are simpler but provide a partial order.
- Vector clocks are more complex but offer richer information about causal relationships.

Choosing the Right Clock:

The choice depends on the application's needs. If determining causal relationships is crucial, vector clocks are preferred. If a simpler approach for partial ordering suffices, Lamport timestamps may be adequate.

Election Algorithms (8 Marks)

Purpose and Benefits of Election Algorithms (2 Marks)

- Used to choose a single coordinator process among multiple contenders in a distributed system.
- The coordinator can perform various tasks:
    - **Breaking Ties:** Resolving conflicts or making decisions when multiple processes have equal priority.
    - **Managing Shared Resources:** Acting as a central control point for accessing or updating shared resources to prevent race conditions.

Popular Election Algorithms (4 Marks)

Bully Algorithm (2 Marks)

Concept: The Bully Algorithm is a simple and fast election algorithm for choosing a coordinator process in a distributed system. It leverages process IDs (assumed unique) and relies on a "bullying" approach.

Advantages:

- Easy to understand and implement.
- Fast election time, especially for small networks.

Disadvantages:

- Less fault-tolerant. Failures can disrupt the election if the elected coordinator fails.
- Not scalable for large networks as message overhead can increase significantly.

Algorithm Explanation:

1. **Process Failure Detection:** A process (P) notices the coordinator has failed (e.g., by missing heartbeats or timeouts).
2. **Election Initiation:** P initiates the election by sending an ELECTION message to all processes with higher IDs.
3. **Responding to ELECTION:** When a process with a higher ID (Q) receives an ELECTION message, it understands it cannot be the coordinator as P (with a lower ID) is still alive. Q sends a COORDINATOR message back to P, indicating it won't participate in the election.
4. **Bully and Become Coordinator:** Upon receiving COORDINATOR messages from all higher-ID processes or no responses after a timeout, P declares itself the coordinator and sends a COORDINATOR DONE message to all processes.
5. **Update Others:** All processes update their state to reflect the new coordinator (P) upon receiving the COORDINATOR DONE message.

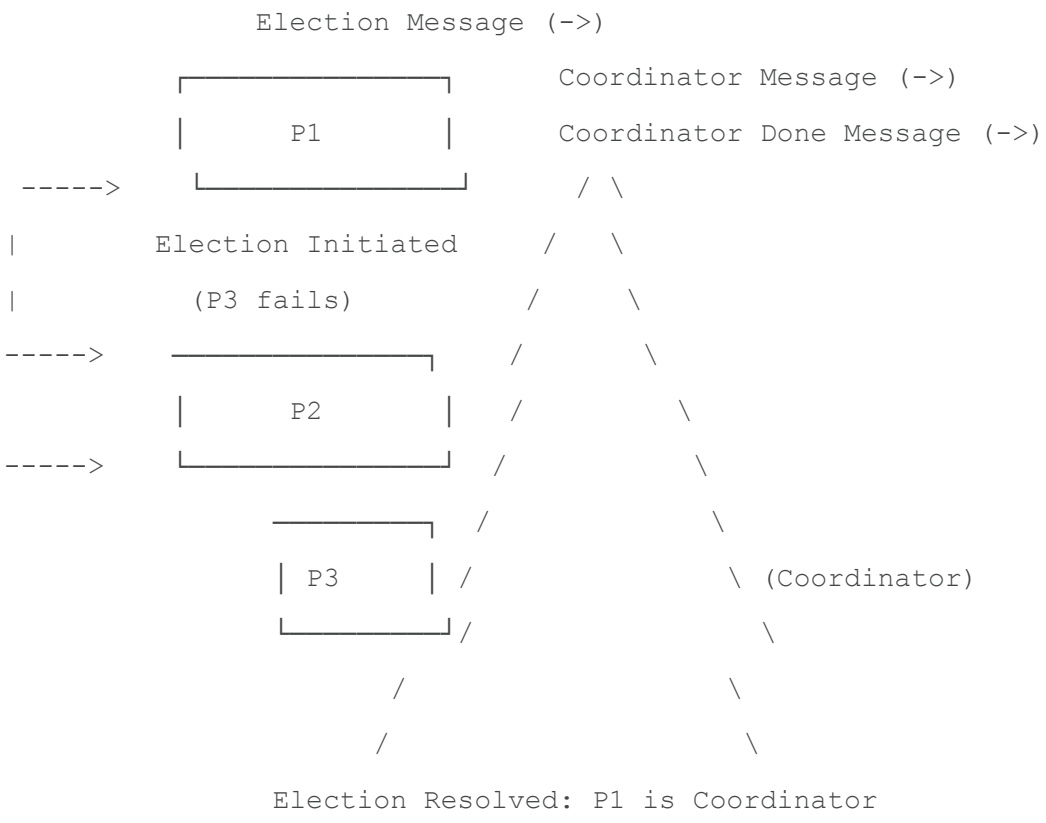
Example:

Consider processes P1, P2, and P3 with P3 as the coordinator.

1. P3 fails.

2. P2 detects the failure and initiates the election by sending ELECTION messages to P1.
3. P1 receives the ELECTION message and realizes it has the highest remaining ID. It sends a COORDINATOR message back to P2.
4. P2 receives the COORDINATOR message and updates its state.
5. P1 broadcasts a COORDINATOR DONE message, informing everyone it's the new coordinator.

Diagram:



Ring Election Algorithm (2 Marks)

Concept: The Ring Election Algorithm uses a virtual ring structure for election. A special token is passed around the ring, and the process holding the token at the end becomes the coordinator.

Advantages:

- Fault-tolerant. The election can restart from any surviving process with the token if the coordinator fails.
- Scalable to larger networks compared to the Bully Algorithm.

Disadvantages:

- May have higher latency depending on the size of the ring. The time it takes for the token to circulate can be significant, impacting election completion time.
- Requires a mechanism to handle a broken ring (e.g., process failures) to ensure election completion.

Algorithm Explanation:

1. **Initial State:** Processes are arranged in a logical ring structure. A special token circulates around the ring, and initially, any process can hold it.
2. **Election Trigger:** When a process detects the coordinator failure (e.g., missing token after a timeout), it starts the election by holding onto the token (if it doesn't already have it).
3. **Token Passing:** The process holding the token passes it to its logical successor in the ring.
4. **Coordinator Selection:** The process that receives the token for the first time after the election is initiated becomes the new coordinator.
5. **Notification:** The new coordinator sends a COORDINATOR message to all processes informing them of its role.

Example:

Consider a ring of processes P1, P2, P3, and P4, with P4 currently holding the token (not necessarily the coordinator).

1. P3 detects the coordinator failure (previously held by P1).
2. P3 keeps the token (since it has it) and starts the election.
3. P3 passes the token to P4.
4. P4 receives the token and realizes it's the first process to have it after the election initiation. It becomes the coordinator.
5. P4 broadcasts a COORDINATOR message to all processes.

Diagram:

```
Token Flow (->
```

Considerations When Choosing an Election Algorithm (2 Marks)

- **Simplicity and Efficiency:** Easier to understand and implement algorithms may be preferred for less critical tasks.
- **Fault Tolerance:** For crucial tasks, algorithms that can recover from process failures are essential.
- **Performance:** Consider the communication overhead and latency involved in different algorithms.

MODULE 4: Resource and Process Management... Six steps / design issues for Load Sharing /Balancing etc.

Resource and Process Management (8 Marks)

Understanding Resource and Process Management (2 Marks)

Distributed systems consist of multiple interconnected machines collaborating to achieve a common goal. Efficient resource and process management is crucial for optimal performance and resource utilisation. This involves:

- **Resource Allocation:** Distributing resources (processors, memory, network bandwidth) among processes to complete tasks efficiently.

- **Process Scheduling:** Determining the order in which processes are executed on available resources.
- **Load Balancing:** Distributing workload evenly across resources to prevent overloading specific nodes and improve overall system performance.
- **Process Migration:** Moving processes between machines to balance load or utilise resources more effectively.
- **Deadlock Detection and Recovery:** Identifying situations where processes are permanently blocked waiting for resources held by each other, and implementing strategies to regain control.

Benefits of Effective Resource and Process Management (2 Marks)

- **Improved Performance:** Efficient resource allocation and load balancing ensure tasks are completed quickly without overloading resources.
- **Scalability:** The system can accommodate increased processing demands by adding new machines and redistributing resources.
- **Availability:** By preventing deadlocks and recovering from failures, the system remains operational for users.
- **Cost Optimization:** Resources are utilized efficiently, avoiding bottlenecks and underutilized machines.

Challenges in Resource and Process Management (2 Marks)

- **Heterogeneity:** Machines in a distributed system may have varying capabilities, requiring dynamic resource allocation.
- **Distributed Nature:** Processes and resources are geographically dispersed, necessitating coordination across the network.
- **Deadlock Potential:** Processes competing for shared resources can create deadlock situations.
- **Dynamic Workload:** Load may vary over time, requiring algorithms to adjust resource allocation accordingly.

Resource and Process Management Techniques (2 Marks)

- Centralized vs. Distributed Scheduling: Centralized control provides better overall visibility but introduces a single point of failure. Distributed scheduling offers flexibility but may lead to suboptimal resource allocation.
- Static vs. Dynamic Allocation: Static allocation assigns resources at compile time, while dynamic allocation assigns resources at runtime based on current needs.
- Load Balancing Algorithms: Techniques like round-robin, least-loaded, and dynamic thresholds can distribute workload efficiently.

Design Issues for Load Sharing/Balancing (8 Marks)

Understanding Load Sharing/Balancing (2 Marks)

Load sharing/balancing aims to distribute workload (tasks, processes) evenly across available resources in a distributed system. This prevents overloading specific nodes, improves overall system responsiveness, and avoids bottlenecks.

Six Design Issues for Load Sharing/Balancing (6 Marks)

Information Collection (2 Marks)

- How do processes gather information about system load and resource availability? (e.g., CPU usage, memory usage, network bandwidth utilization)
    - **Monitoring Tools:** Agents or daemons can constantly monitor resource utilization on each machine.
    - **Periodic Updates:** Processes can periodically report their load to a central server or a gossip protocol can be used for more dynamic updates.

Work Unit Definition (2 Marks)

- How are tasks or processes divided into manageable units for distribution? (e.g., data processing jobs, independent subtasks)
    - **Granularity:** Work units should be large enough to minimize overhead but small enough for parallel execution.

    - **Task Decomposition:** Complex tasks can be broken down into smaller, independent subtasks suitable for load balancing.

Work Assignment (2 Marks)

- How are work units assigned to specific resources based on load information? (e.g., central server, dynamic allocation algorithms)
    - **Centralized Server:** A central server can collect load information and assign work units to underloaded resources.
    - **Distributed Algorithms:** Processes can exchange load information and use algorithms like least-loaded or round-robin to choose a suitable resource.

Migration (2 Marks)

- Can processes or work units be migrated between resources to balance load? (e.g., container migration in cloud environments)
- **Static vs. Dynamic Migration:** Processes might be statically assigned to resources initially, but dynamically migrated later based on load changes.
- **Migration Considerations:** Overhead of migration, data consistency requirements, and network bandwidth availability need to be evaluated.

Information Dissemination (2 Marks)

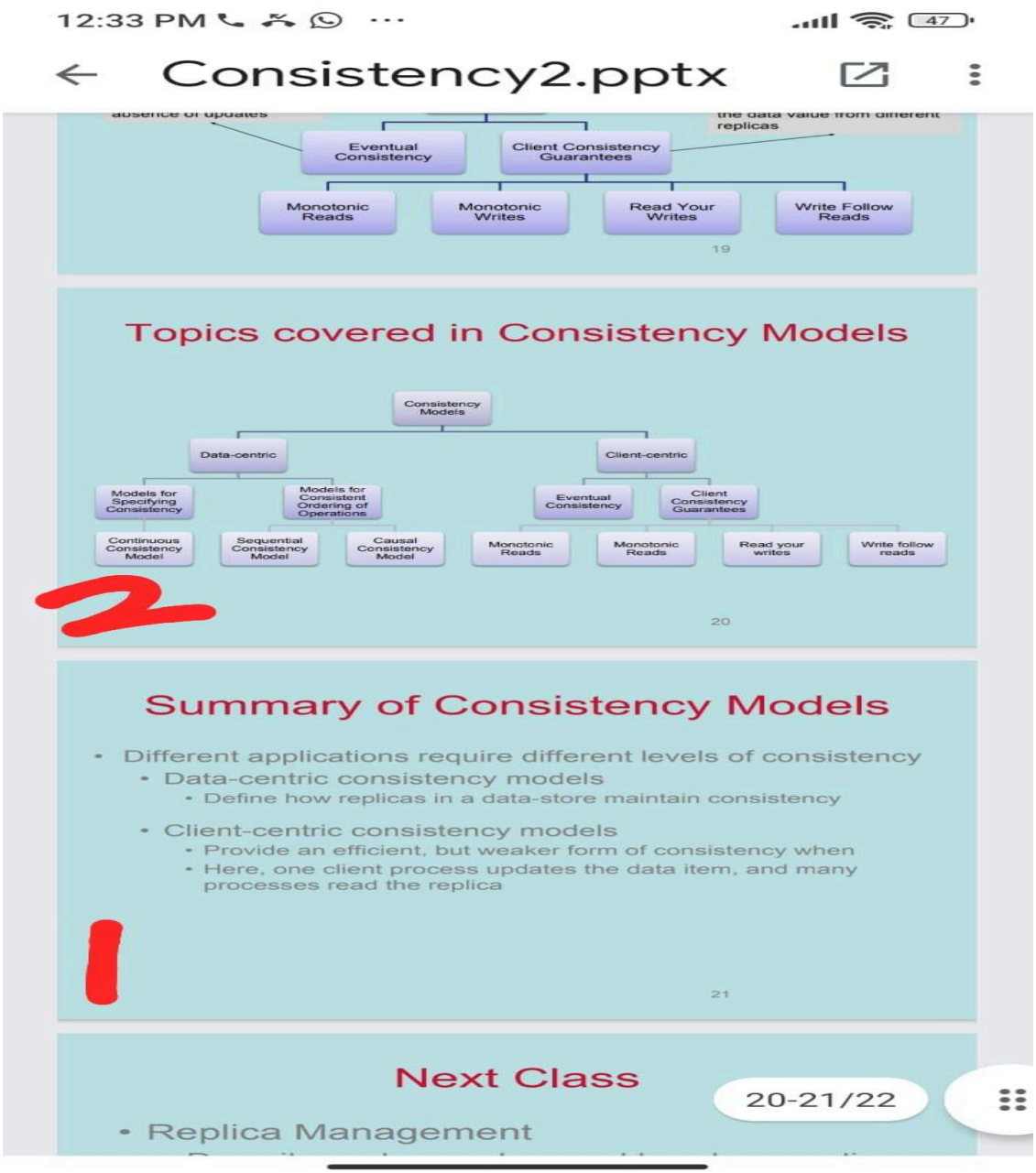
- How is load information kept up-to-date across the system? (e.g., periodic updates, gossip protocols)
- **Maintaining Consistency:** Outdated load information can lead to unbalanced allocation. Periodic updates or gossip protocols can ensure consistent load data.
- **Gossip Protocols:** Processes periodically exchange load information with their neighbors, allowing the information to propagate through the network.

Failure Handling (2 Marks)

- How does the system handle failures of processes or resources during load balancing? (e.g., replicating tasks, retrying assignments)

- **Fault Tolerance:** Mechanisms to handle failures are crucial for a robust system.
- **Retries:** Work units can be retried

MODULE 5 : Replication and Consistency Models... Total 11 models... 7 data centric... 4 client centric....



Summary of Consistency Models

Consistency	Description
Strict	Absolute time ordering of all shared accesses matters.
Linearizability	All processes must see all shared accesses in the same order. Accesses are furthermore ordered according to a (nonunique) global timestamp
Sequential	All processes see all shared accesses in the same order. Accesses are not ordered in time
Causal	All processes see causally-related shared accesses in the same order.
FIFO	All processes see writes from each other in the order they were used. Writes from different processes may not always be seen in that order

(a)

Consistency	Description
Weak	Shared data can be counted on to be consistent only after a synchronization is done
Release	Shared data are made consistent when a critical region is exited
Entry	Shared data pertaining to a critical region are made consistent when a critical region is entered.

(b)

Example: Consistency for Mobile Users

Example: Consider a distributed database to which you have access through your notebook.

Assume your notebook acts as a front end to the database.

- At location A you access the database doing reads and updates.
- At location B you continue your work, but unless you access the same server as the one at location A, you may detect inconsistencies:
  - you may be reading newer entries than the ones available at A
  - your updates at B may eventually conflict with those at A
  - your updates at A may not have yet been propagated to B

Note: The only thing you really want is that the entries you updated and/or read at A, are in B the way you left them in A. In that case, the database will appear to be consistent to you.

Eventual Consistency

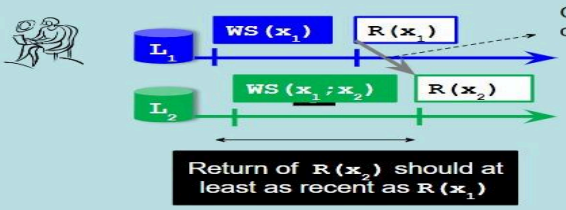
- Many applications can tolerate a inconsistency for a long time
  - Webpage updates, Web Search – Crawling, indexing and ranking, Updates to DNS Server
- In such applications, it is acceptable and efficient if replicas in the data-store rarely exchange updates
- A data-store is termed as *Eventually Consistent* if:
  - All replicas will gradually become consistent in the absence of updates
- Typically, updates are propagated infrequently in eventually consistent data-stores

Designing Eventual Consistency

- In eventually consistent data-stores,
  - *Write-write conflicts* are rare
    - Two processes that write the same value are rare
    - Generally, one client updates the data value
      - e.g., One DNS server updates the name to IP mapping
    - Such rare conflicts can be handled through simple mechanisms, such as mutual exclusion
  - R*ead-write conflict* are more frequent
    - Conflicts where one process is reading a value, while another process is writing a value to the same variable
    - Eventual Consistency Design has to focus on efficiently resolving such conflicts

Monotonic Reads

- The model provides guarantees on successive reads
- If a client process reads the value of data item **x**, then any successive read operation by that process should return the same or a more recent value for **x**

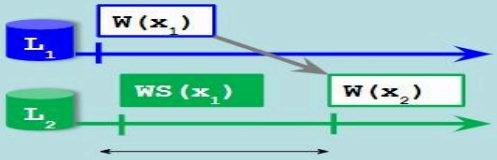


Monotonic Writes

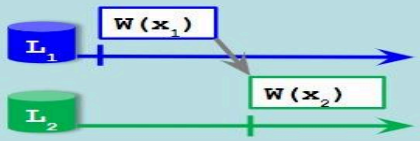
This consistency model assures that writes are monotonic

A write operation by a client process on a data item **x** is completed before any successive write operation on **x** by the same process

- A new write on a replica should wait for all old writes on any replica



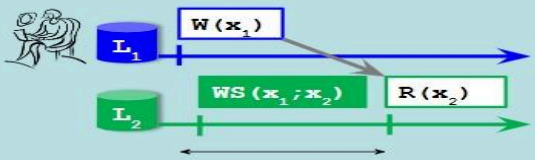
) operation should be performed only after result of W (x_1) has been updated at L_2



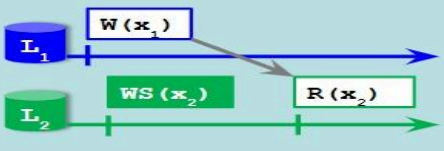
The data-store does not provide monotonic write consistency

Read Your Writes

- The effect of a write operation on a data item **x** by a process will always be seen by a successive read operation on **x** by the same process
- Example scenario:
  - In systems where password is stored in a replicated data-base, the password change should be seen immediately



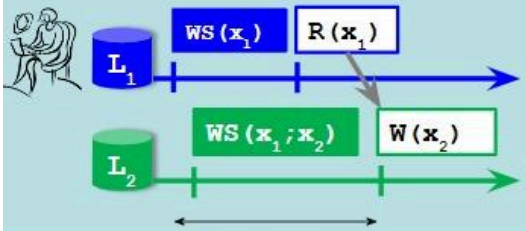
R (x_2) operation should be performed only after the updating the Write Set WS (x_1) at L_2



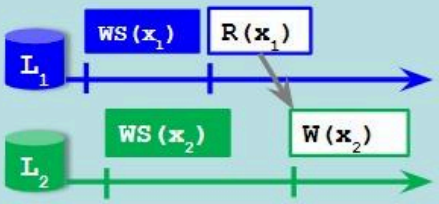
A data-store that does not provide *Read Your Write* consistency

Write Follow Reads

- A write operation by a process on a data item **x** following a previous read operation on **x** by the same process is guaranteed to take place on the same or a more recent value of **x** that was read
- Example scenario:
  - Users of a newsgroup should post their comments only after they have read all previous comments



$W(x_2)$ operation should be performed only after the all previous writes have been seen



A data-store that does not guarantee Write Follow Read Consistency Model