

Bule Hora University
Faculty of Engineering and Technology
Department of Information technology

Year: III

Semester: II

Subject: Introduction to Distributed System
Code: ITec3102

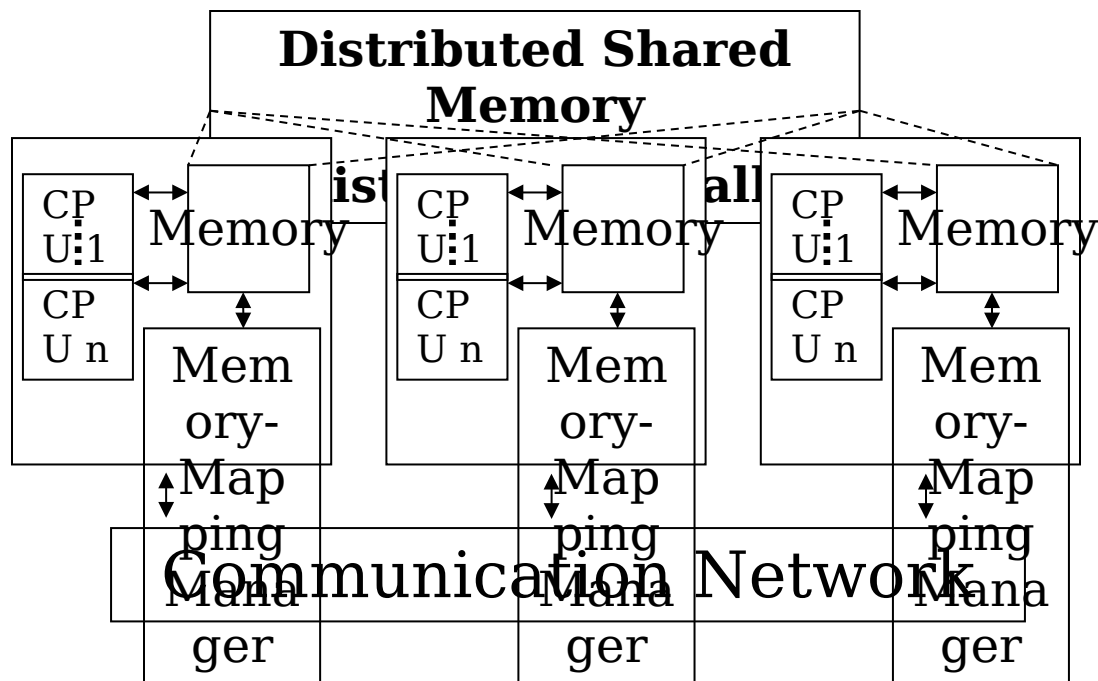
Course

Lecture Notes Unit IV - Models of Distributed Computing

- 1.1 Distributed Shared Memory**
- 1.2 Mutual Exclusion**
- 1.3 Election Algorithms (Leader Selection)**
- 1.4 Data centric Consistency models**
- 1.5 Multi-Core Architecture**

1.1. Distributed Shared Memory

- Distributed Shared Memory (DSM) allows programs running on separate computers to share data without the programmer having to deal with sending messages.
- Programs access what appears to them to be normal memory
- Processes can cause error to one another by altering data



1.2. Mutual Exclusion

1.2.1. Definition

- ✓ A condition in which there is a set of processes, only one of which is able to access a given resource or perform a given function at any time.

1.2.2. Mutual Exclusion in Centralized Systems

Mutual exclusion via:

- Test & set
- Semaphores
- Messages
- Monitors

1.2.3. Distributed Mutual Exclusion

Create an algorithm to allow a process to obtain exclusive access to a resource

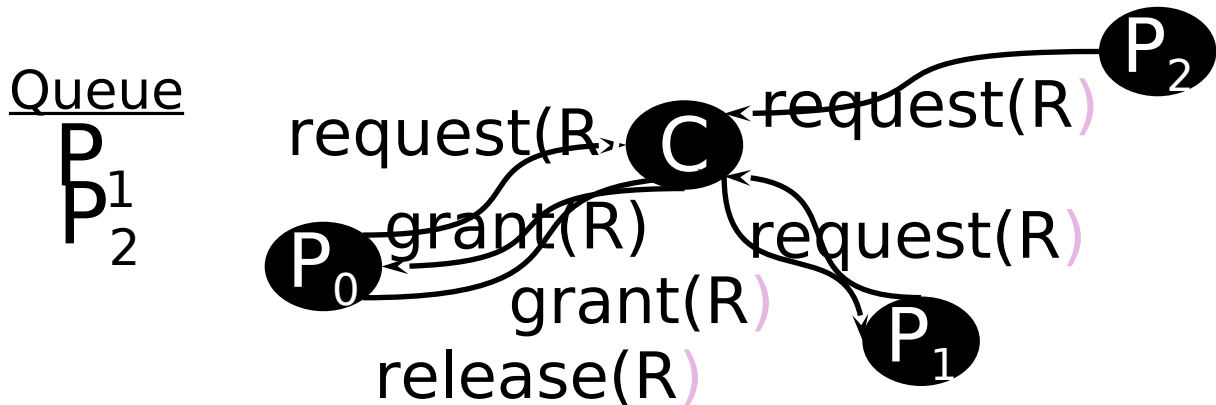
Algorithms

- Centralized Algorithm
- Token Ring Algorithm
- Distributed Algorithm
- Decentralized Algorithm

1.2.3.1. Centralized Algorithm

In this, one process is elected as coordinator. The process that needs resource can send a request to coordinator. If the resource is not accessed by any other processes, the coordinator allows the requested process to use the resource. If another process claimed resource:

- Coordinator does not reply until release
- Maintain queue
 - Service requests in FIFO order



Benefits

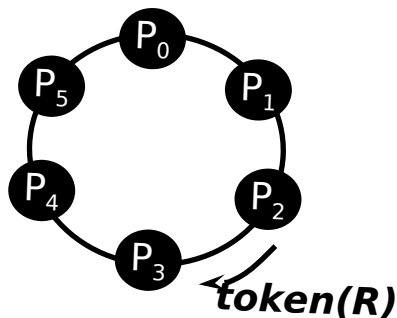
- Fair
 - All requests processed in order
- Easy to implement, understand, verify

Problems

- Process cannot distinguish being blocked from a dead coordinator

1.2.3.2. Token Ring Algorithm

- ✓ Some ordering can be imposed on group
- ✓ Construct logical ring in software
- ✓ Process communicates with neighbor
- ✓ Process 0 gets token for resource R. Token circulates around ring. When process acquires token, Checks to see if it needs to enter critical section. If no, send token to neighbor. If yes, access resource and holds token until done



- Only one process at a time has token
 - Mutual exclusion guaranteed
- Order well-defined

- If token is lost (e.g. process died)
 - It will have to be regenerated
- Does not guarantee FIFO order

1.2.3.3. Distributed Algorithm

Ricart & Agrawala algorithm

- Distributed algorithm using reliable multicast and logical clocks
- Process wants to enter critical section:
 - Compose message containing:
 - Identifier (machine ID, process ID)
 - Name of resource
 - Timestamp (totally-ordered Lamport)
 - Send request to all processes in group
 - Wait until everyone gives permission
 - Enter critical section / use resource
- When process receives request:
 - If receiver not interested:
 - Send OK to sender
 - If receiver is in critical section
 - Do not reply; add request to queue
 - If receiver just sent a request as well:
 - Compare timestamps: received & sent messages
 - Earliest wins
 - If receiver is loser, send **OK**
 - If receiver is winner, do not reply, queue
- When done with critical section
 - Send OK to **all queued requests**

Lamport's Mutual Exclusion

Each process maintains request queue

- Contains mutual exclusion requests

Requesting critical section:

- Process P_i sends request(i, T_i) to all nodes
- Places request on its own queue
- When a process P_j receives a request, it returns a timestamped ack

Entering critical section (accessing resource):

- P_i received a message (*ack* or *release*) from every other process with a timestamp larger than T_i
- P_i 's request has the earliest timestamp in its queue

Difference from Ricart-Agrawala:

- Everyone responds ... always - no hold-back
- Process decides to go based on whether its request is the earliest in its queue

Releasing critical section:

- Remove request from its own queue
- Send a timestamped *release* message
- When a process receives a *release* message
 - Removes request for that process from its queue
 - This may cause its own entry have the earliest timestamp in the queue, enabling it to access the critical section

1.2.3.4. Decentralized Algorithm

- ✓ Every replica has a coordinator that controls access to it (the coordinator is the node that stores it)
- ✓ For a process to use the resource it must receive permission from coordinators
- ✓ This guarantees exclusive access as long as a coordinator only grants access to one process at a time

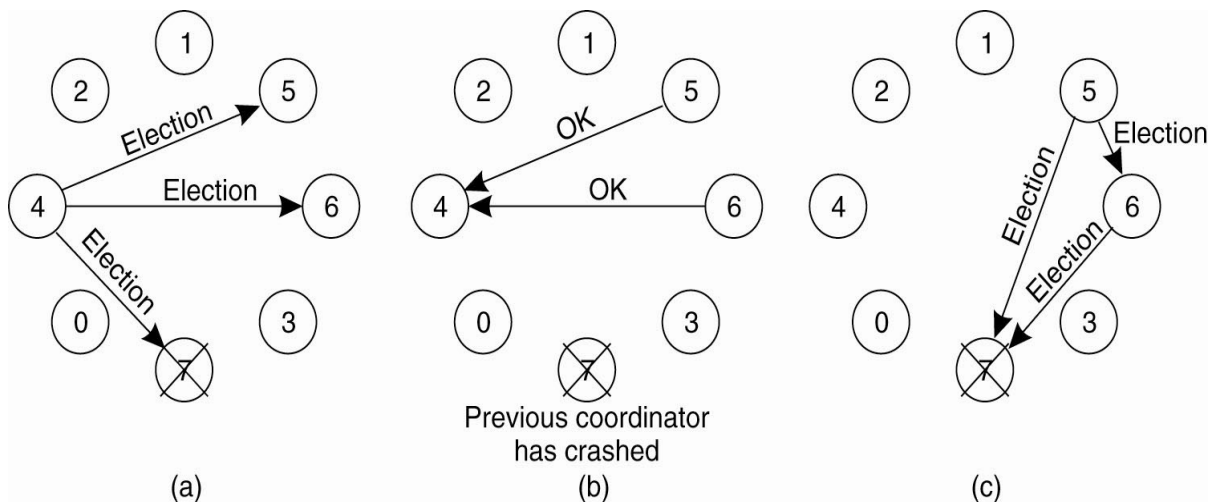
1.3. Election Algorithms (Leader Selection)

- ✓ Many Distributed Systems require a process to act as *coordinator* (for various reasons). The selection of this process can be performed automatically by an "election algorithm".
- ✓ For simplicity, we assume the following:
 - o Processes each have a unique, positive identifier.
 - o All processes know all other process identifiers.
 - o The process with the highest valued identifier is duly elected coordinator.

- ✓ When an election “concludes”, a coordinator has been chosen and is known to all processes.
- The overriding goal of all election algorithms is to have all the processes in a group *agree* on a coordinator.
- There are two types of election algorithm:
 1. **Bully**: “the biggest guy in town wins”.
 2. **Ring**: a logical, cyclic grouping.

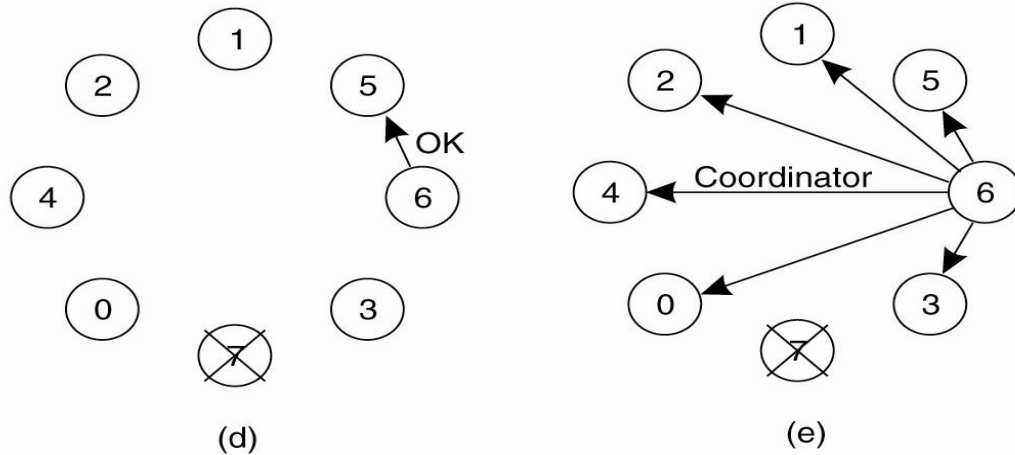
1.3.1. Bully Algorithm

2. P sends an *ELECTION* message to all processes with higher numbers.
3. If no one responds, P wins the election and becomes coordinator.
4. If one of the higher-ups answers, it takes over. P 's job is done.



The bully election algorithm:

- (a) Process 4 holds an election.
- (b) 5 and 6 respond, telling 4 to stop.
- (c) Now 5 and 6 each hold an election.

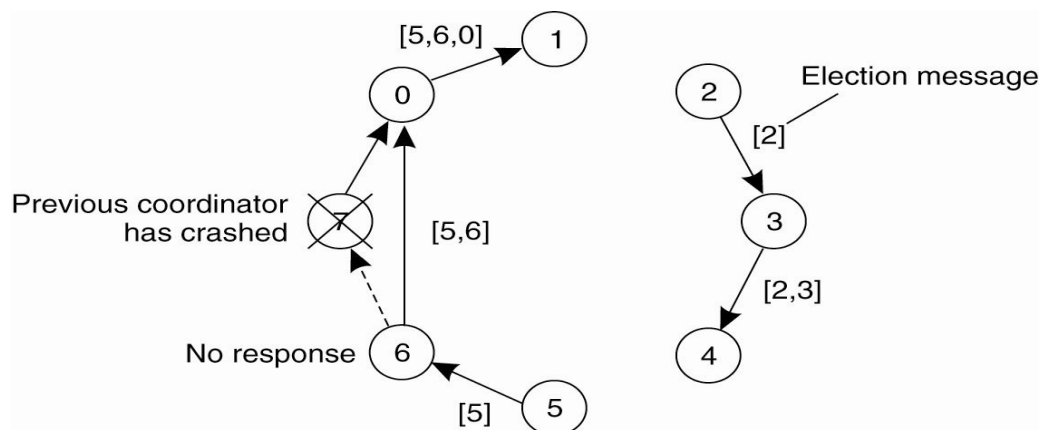


d. Process 6 tells 5 to stop.

(e) Process 6 wins and tells everyone.

1.3.2. The “Ring” Election Algorithm

- ✓ The processes are ordered in a “logical ring”, with each process knowing the identifier of its successor (and the identifiers of all the other processes in the ring).
- ✓ When a process “notices” that a coordinator is down, it creates an ELECTION message (which contains its own number) and starts to circulate the message around the ring.
- ✓ Each process puts itself forward as a candidate for election by adding its number to this message (assuming it has a higher numbered identifier).
- ✓ Eventually, the original process receives its original message back (having circled the ring), determines who the new coordinator is, then circulates a COORDINATOR message with the result to every process in the ring.
- ✓ With the election over, all processes can get back to work.

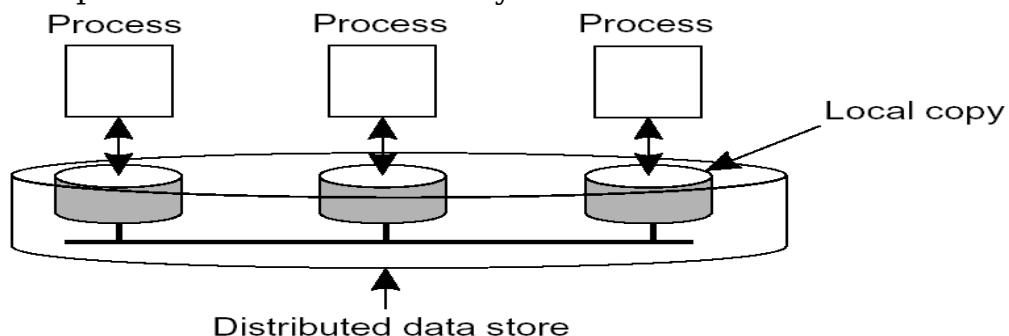


1.4. Introduction to Consistency Model

- ✓ An important issue in distributed systems is the replication of data. Data are generally replicated to enhance **reliability** or improve **performance**. One of the major problems is keeping replicas consistent. Informally, this means that when one copy is updated we need to ensure that the other copies are updated as well; otherwise the replicas will no longer be the same.
- ✓ If objects (or data) are shared, we need to do something about concurrent accesses to guarantee state consistency.
- ✓ To keep replicas consistent, we generally need to ensure that all **conflicting** operations are done in the same order everywhere
- ✓ Conflicting operations: From the world of transactions:
 - Read-write conflict: a read operation and a write operation act concurrently
 - Write-write conflicts: two concurrent write operations

Consistency Model

- ✓ A consistency model is essentially a contract between processes and the data store (distributed collection of storages accessible to clients). It says that if processes agree to obey certain rules, the store promises to work correctly.



Data Centric Consistency Models

Strong consistency models: Operations on shared data are synchronized (models not using synchronization operations):

- Strict consistency (related to absolute global time)
- Linearizability (atomicity)
- Sequential consistency (what we are used to - serializability)
- Causal consistency (maintains only causal relations)
- FIFO consistency (maintains only individual ordering)

Weak consistency models: Synchronization occurs only when shared data is locked and unlocked (models with synchronization operations):

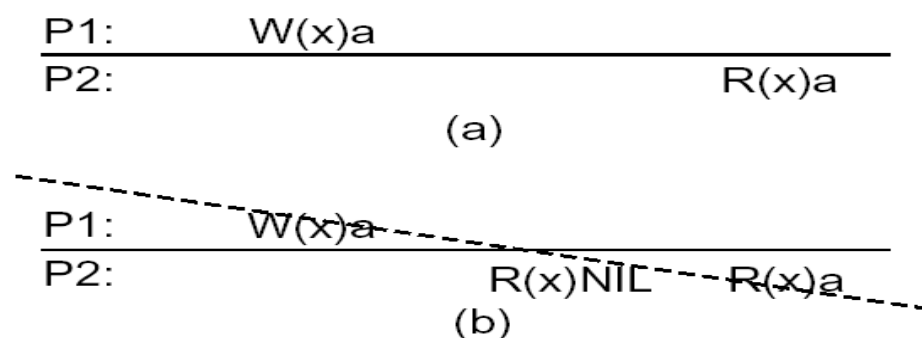
- General weak consistency
- Release consistency
- Entry consistency

Observation: The weaker the consistency model, the easier it is to build a scalable solution.

Strict Consistency

Any read to a shared data item X returns the value stored by the most recent write operation on X .

Observation: It doesn't make sense to talk about "the most recent" in a distributed environment.



- Assume all data items have been initialized to NIL
- $W(x)a$: value a is written to x
- $R(x)a$: reading x returns the value a

The behavior shown in Figure (a) is correct for strict consistency

The behavior shown in Figure (b) is incorrect for strict consistency

Sequential Consistency

Sequential consistency is a slightly weaker consistency model than strict consistency. A data store is said to be sequentially consistent when it satisfies the following condition:

The result of any execution is the same as if the (read and write) operations by all processes on the data store were executed in some sequential order, and the operations of each individual process appear in this sequence in the order specified by its program.

Figure (a) - a sequentially consistent data store

- P1 first performs $W(x)a$ to x . Later in absolute time, P2 also performs $W(x)b$ to x
- Both P3 & P4 first read value b and later value a . The write operation of P2 appears to have taken place before that of P1 to both P3 & P4

P1:	$W(x)a$		
P2:	$W(x)b$		
P3:		$R(x)b$	$R(x)a$
P4:		$R(x)b$	$R(x)a$

(a)

~~P1:~~	~~$W(x)a$~~		
P2:	$W(x)b$		
P3:		$R(x)b$	$R(x)a$
P4:		$R(x)a$	~~$R(x)b$~~

(b)

Figure (b) - a data store that is not sequentially consistent

- Not all processes see the same interleaving of write operations

Linearizability

A consistent model that is weaker than strict consistency, but stronger than sequential consistency is linearizability.

A data store is said to be **linearizable** when each operation is timestamped and the following condition holds:

The result of any execution is the same as if the (read and write) operations by all processes on the data store were executed in some sequential order, and the operations of each individual process appear in this sequence in the order specified by its program.

Casual Consistency

The causal consistency model is a weaker model than sequential consistency.

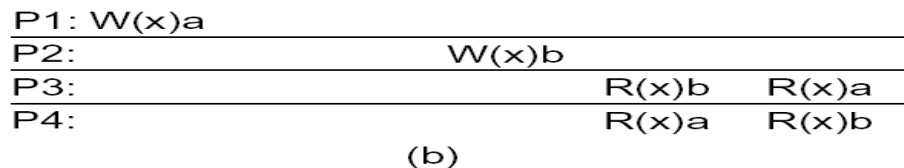
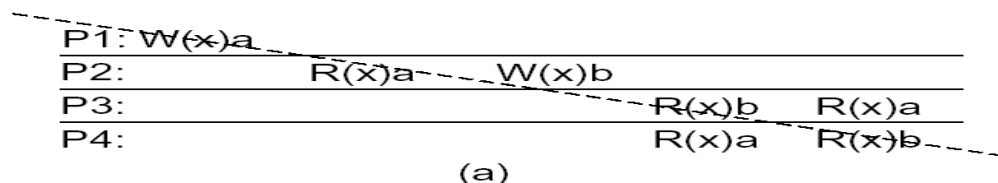
It makes a distinction between events that are potentially causally related and those that are not.

- If event B is caused or influenced by an earlier event, A, causality requires that everyone else first see A, then see B.

Figure (a) - a data store that is not causally consistent

- Two writes, $W(x)a$ and $W(x)b$, are casually related since b may be a result of a computation involving $R(x)a$

Figure (b) - a data store that is causally consistent

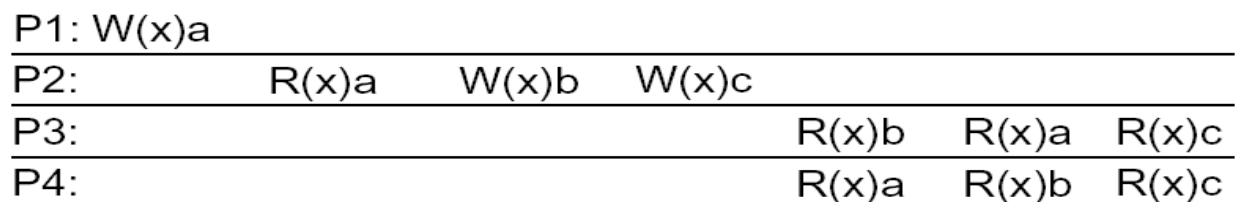


FIFO Consistency

FIFO consistency is weaker than causal consistency

A data store is said to be FIFO consistent when it satisfies the following condition:

Writes done by a single process are received by all other processes in the order in which they were issued, but writes from different processes may be seen in a different order by different processes.



Weak Consistency

Although FIFO consistency can give better performance than the stronger consistency models, it is still unnecessarily restrictive for many applications because they require that writes originating in a single process be seen everywhere in order

Not all applications require seeing all writes or seeing them in order

Solution: Use a synchronization variable. Synchronize(S) synchronizes all local copies of the data store

Using synchronization variables to partly define consistency is called weak consistency - has three properties:

1. Accesses to synchronization variables are sequentially consistent.
2. No access to a synchronization variable is allowed to be performed until all previous writes have completed everywhere.
3. No data access is allowed to be performed until all previous accesses to synchronization variables have been performed.

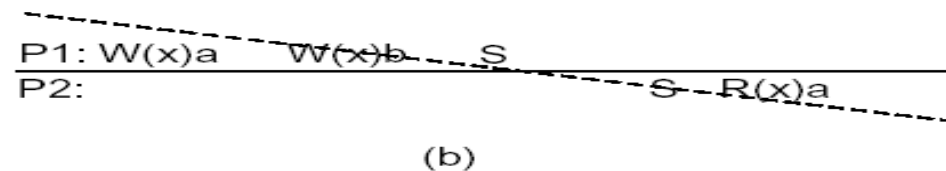
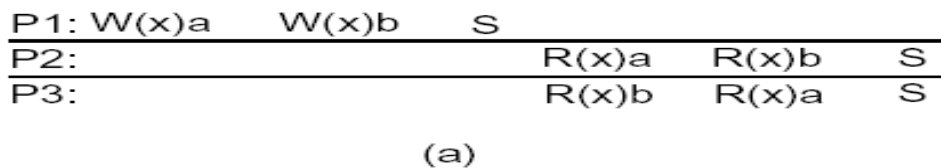


Figure (a) - a data store that is weak consistent (i.e., valid sequence)

- P1 performs $W(x)a$ and $W(x)b$ and then synchronizes. P2 and P3 have not yet been synchronized, thus no guarantees are given about what they see

Figure (b) - a data store that is not weak consistent

Since P2 has synchronized, $R(x)$ in P2 must read **b**

Release Consistency

Weak consistency has the problem that when a synchronization variable is accessed, the data store does not know whether this is being done because the process is either

1. Finished writing the shared data, or
2. About to start reading data

Consequently, the data store must take the actions required in both cases

- Make sure that all locally initiated writes have been completed (i.e., propagated to other copies)
- Gathering in all writes from other copies

If the data store could tell the difference between entering a critical region or leaving one, a more efficient implementation might be possible.

Idea: Divide access to a synchronization variable into two parts: an acquire and a release phase.

1. About to start accessing data - Acquire forces a requester to wait until the shared data can be accessed
2. Finished accessing the shared data - Release sends requester's local value to other servers in data store.

P1:	Acq(L)	W(x)a	W(x)b	Rel(L)
P2:			Acq(L)	R(x)b
P3:				R(x)a

Since P3 does not do an acquire before reading x, the data store has no obligation to give it the current value of x, so returning a is ok.

Entry Consistency

- With release consistency, all local updates are propagated to other copies/servers during release of shared data.
- With entry consistency, each shared data item is associated with a synchronization variable.
- In order to access consistent data, each synchronization variable must be explicitly acquired.

Release consistency affects all shared data but entry consistency affects only those shared data associated with a synchronization variable.

P1:	Acq(Lx)	W(x)a	Acq(Ly)	W(y)b	Rel(Lx)	Rel(Ly)
P2:			Acq(Lx)	R(x)a	R(y)	NIL
P3:			Acq(Ly)	R(y)b		

Since P2 did not do acquire before reading y, P2 may not read the latest.

By having the distributed system use and handle distributed shared objects (i.e., the system does acquire on the object's associated synch variable when a client access a shared distributed object).

1.5. Multi-Core Architecture

- ✓ Multi-core architectures with an especially high number of cores (tens or hundreds or even more) and it is used in distributed systems.
- ✓ Multi-core computer is composed of two or more independent cores

Core (CPU): computing unit that reads/executes program instructions

Ex) dual-core, quad-core, hexa-core, octa-core, ...