COMP2121: Principles of Distributed Systems and Networks

Lecture title

Unit coordinator Dr. Vincent Gramoli
School of Information Technologies

THE UNIVERSITY OF SYDNEY

Motivations

- Entities are distributed so they may act concurrently to update some data
- Data may also be distributed (i.e., replicated) and not all replicas can be accessed simultaneously

- What does it mean to be consistent?

Outline

- Definitions
- Sequential consistency
- Causal consistency
- Replication
- Multiple-access operations

Definitions

THE UNIVERSITY OF
SYDNEY

Definitions

Problem of inconsistency

- Data is distributed (replicated) to
  - Improve the reliability of a system
  - Scale in numbers: preserve performance while multiplying entities
  - Scale in geographical area: preserve performance while increasing space
- Drawback: cost of keeping the replicas consistent
  - All replicas must have to maintain information
  - Intuitively, a client update must be propagated to all replicas
  - but a client can obtain an “up-to-date” information from any replica
- Not so simple!
  - Propagation takes time and one replica may be up-to-date while another is not
  - *Inconsistencies*: two clients may observe a different data depending on the replica they contact

Definitions

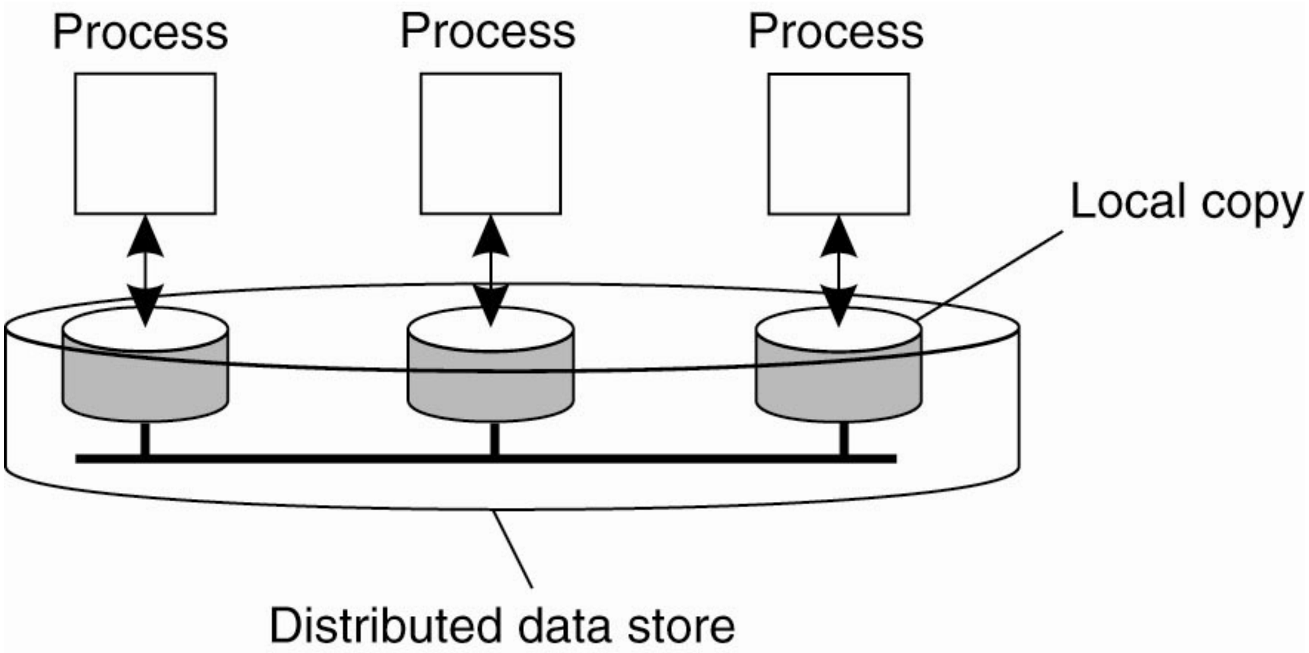
Conflict

- Two (groups of) operations are *concurrent* in the traditional sense here:
  - Each of the two (groups of) operations starts before the other ends
- An *update operation* is an operation that modifies the data
  - E.g., a write operation or a transaction (i.e., a group of operations) with a write
- A *conflict* is a relation between two (groups of) operations:
  - that are concurrent,
  - access the same data, and
  - (at least) one of which is an update

Definitions

Data store

- A *data store* is a distributed collection of storages



- From the standpoint of a given process, only one storage copy represents its *local copy*

Definitions

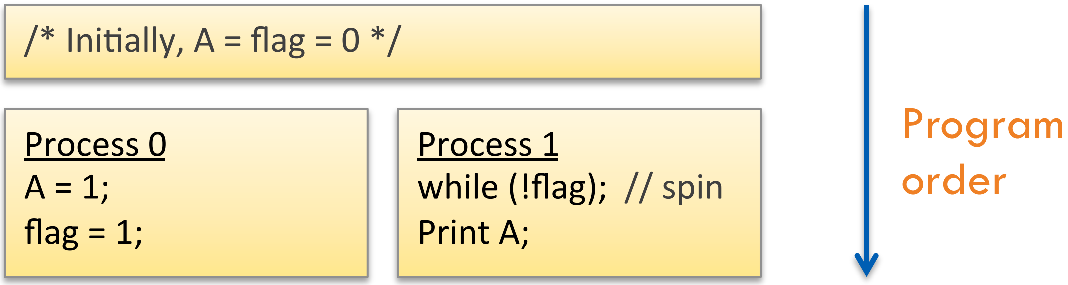
Consistency

- A *consistency model* is a contract between the processes and the data store such that if processes obey certain rules, the store promises to work correctly
- Data-centric vs. client-centric consistency
  - Data-centric consistency: consistency in the general sense (e.g., causal consistency, sequential consistency)
  - Client-centric consistency: consistency for a single client (independently from other clients), assuming that a single client does not trigger multiple concurrent updates (e.g., eventual consistency)

Definitions

Coherence vs. consistency

- Coherence concerns only one memory location
- Consistency concerns apparent ordering for all locations



```
/* Initially, A = flag = 0 */
```

```
Process 0
A = 1;
flag = 1;
```

```
Process 1
while (!flag);  // spin
Print A;
```

- Example: output “0” is coherent but non (sequentially) consistent
- A memory system is coherent if for each location, we can serialize all operations to that location such that, operations performed by any processor appear in program order (i.e., the value returned by a read is the value written by the last write to that location)

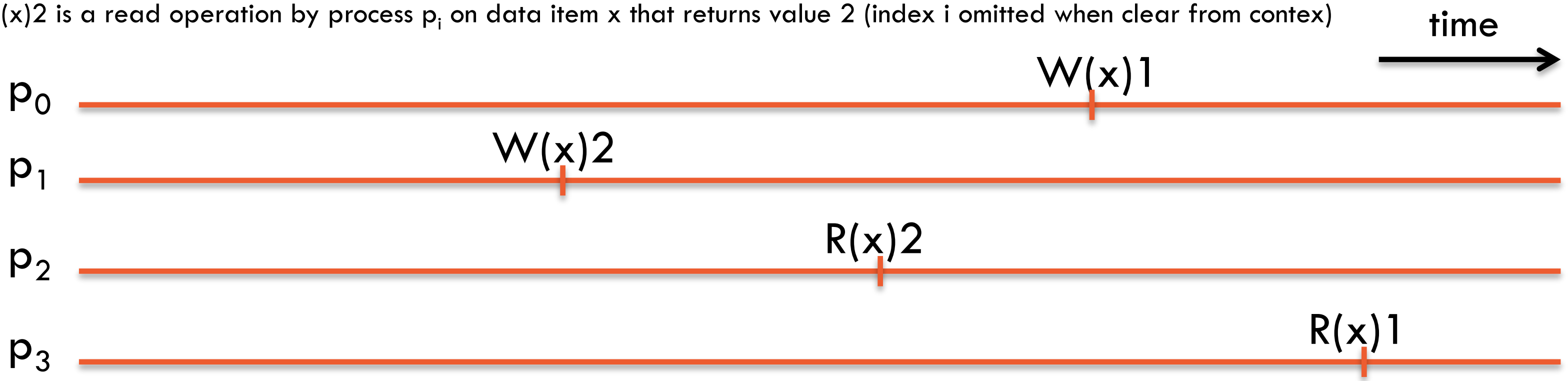
Sequential Consistency

THE UNIVERSITY OF SYDNEY

Sequential consistency

Distributed execution

- *Program order:* The order in which events appear to be executed locally (in the program)
- *Execution:* A sequence of reads (R) and writes (W) on some data items x… executed by distributed processes p_0…
- *Serial execution:* An execution where each process executes (all its operations) one after another
- *Equivalent executions:* Two executions are equivalent if they contain exactly the same reads and writes:
  - executed from the same process,
  - all writes have the same input values and
  - all reads have the same output values
- Notation:
  - $W_i(x)1$ is a write operation by process p_i on data item x with value 1 (index i omitted when clear from context)
  - $R_i(x)2$ is a read operation by process p_i on data item x that returns value 2 (index i omitted when clear from contex)

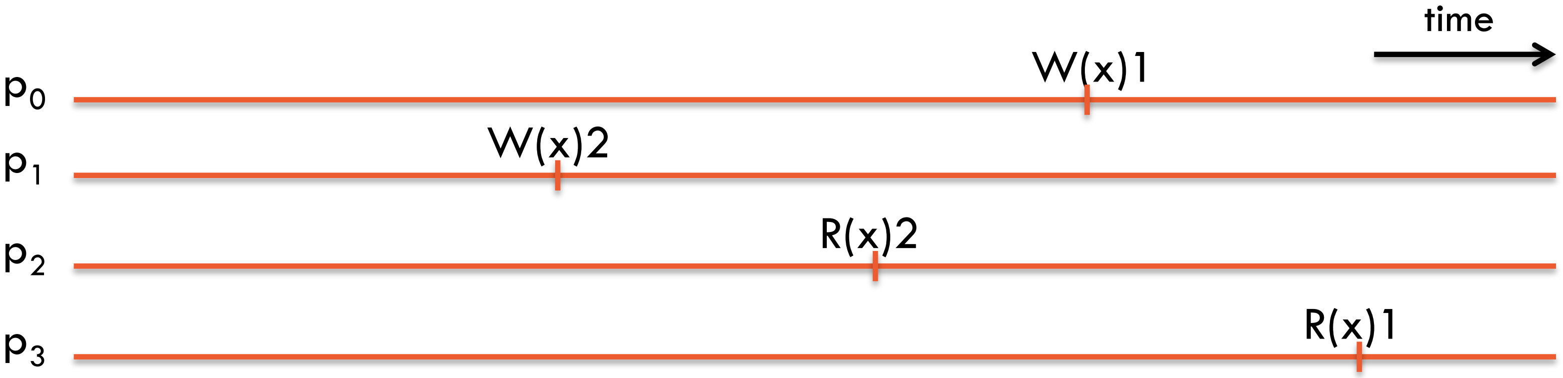


  - Example 1: the execution $W_1(x)2;R_2(x)2;W_0(x)1;R_3(x)1$ is serial in that each process executes one after another in the order p_1; p_2; p_0; p_3

Sequential consistency

Sequentially consistent executions

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

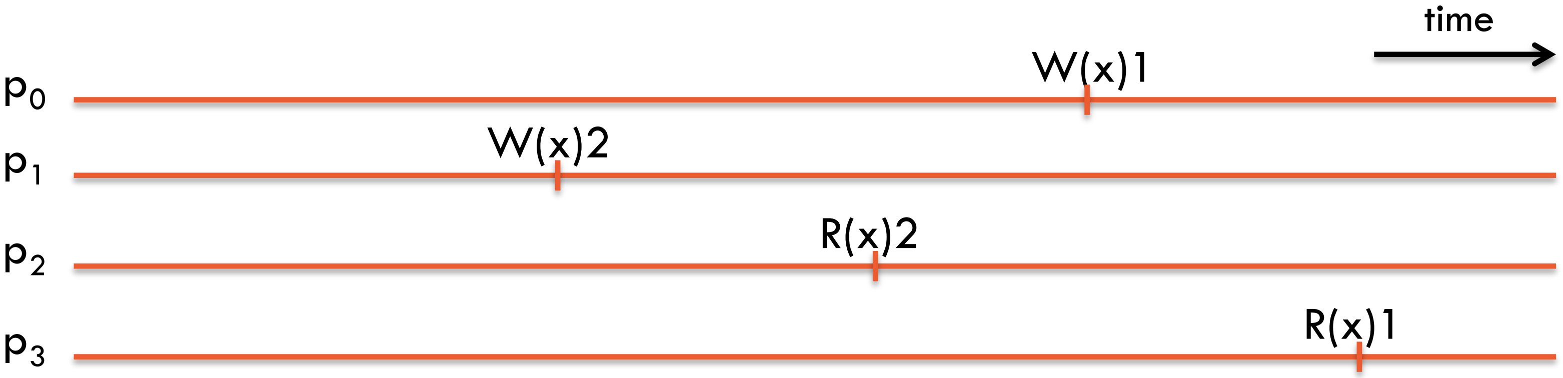


- Example 1(bis): is this execution sequentially consistent as well?

Sequential consistency

Sequentially consistent executions

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



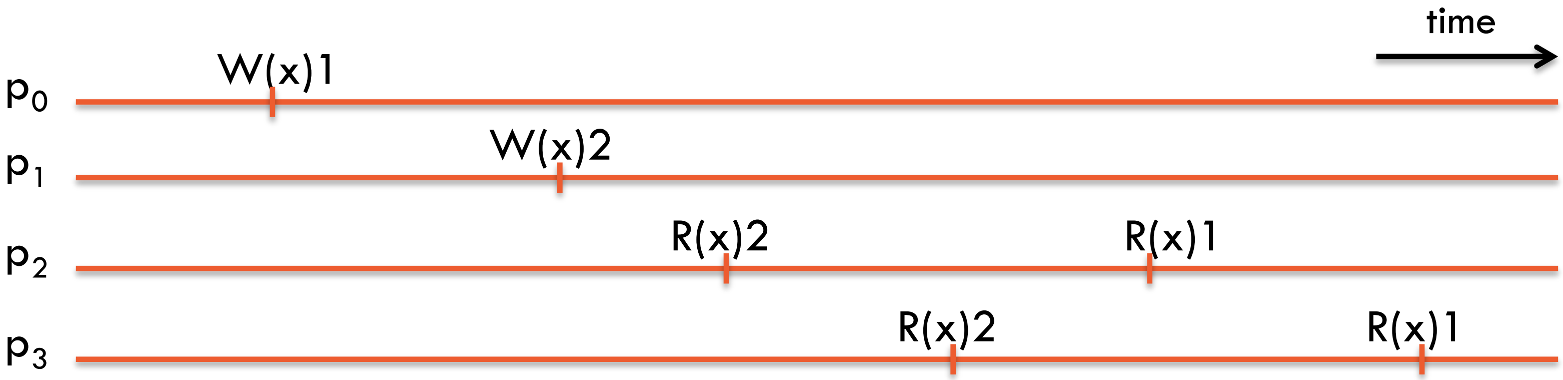
- Example 1(bis): the result of this serial execution is sequentially consistent as well

If an execution is serial, then its result is always sequentially consistent

Sequential consistency

A non-sequentially consistent execution

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

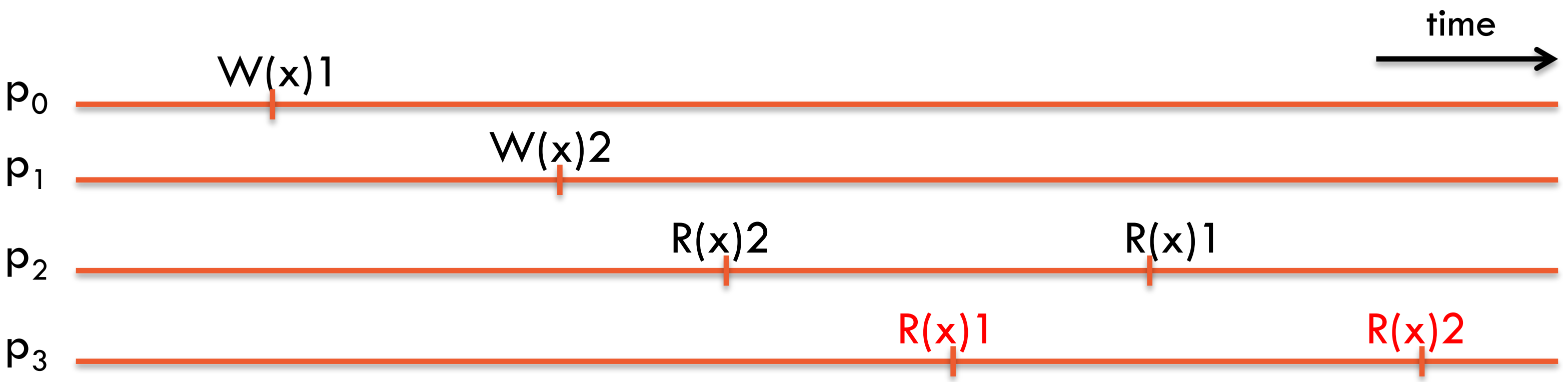


- <u>Example 2</u>: the result of this execution is sequentially consistent
  - The result of the execution is $W_0(x)1$; $W_1(x)2$; $R_2(x)2$; $R_3(x)2$; $R_2(x)1$; $R_3(x)1$
  - An equivalent sequential order with the same result is $W_1(x)2$; $R_2(x)2$; $R_3(x)2$; $W_0(x)1$; $R_2(x)1$; $R_3(x)1$
  - The program order of each process is satisfied by this sequence:
    - $R_3(x)2$ before $R_3(x)1$ and
    - $R_2(x)2$ before $R_2(x)1$

Sequential consistency

A non-sequentially consistent execution

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



- Example 3: the result of this execution is not sequentially consistent
  - The result of the execution is $W_0(x)1$; $W_1(x)2$; $R_2(x)2$; $R_3(x)1$; $R_2(x)1$; $R_3(x)2$
  - There is no sequential execution in which:
    - By program order of p_3, $R_3(x)1$ is before $R_3(x)2$ (implying that $W_0(x)1$ would be before $W_1(x)2$ and
    - By program order of p_2, $R_2(x)2$ before $R_2(x)1$ (implying that $W_1(x)2$ would be before $W_0(x)1$)

⇒ contradiction

Sequential consistency

Application to shared memory multiprocessors

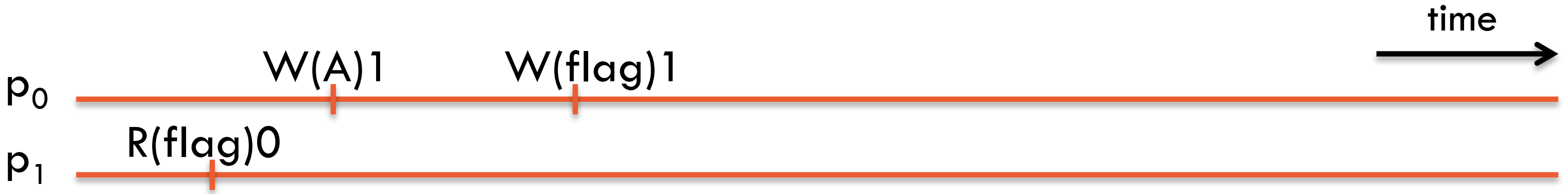
- How can a coherent execution may not be sequentially consistent

```
/* Initially, A = flag = 0 */

Process 0          Process 1
A = 1;             while (!flag);  // spin
flag = 1;          Print A;
```

Program order

- Example 4: What are the possible outputs of this concurrent program?



Sequential consistency

Application to shared memory multiprocessors

- How can a coherent execution may not be sequentially consistent

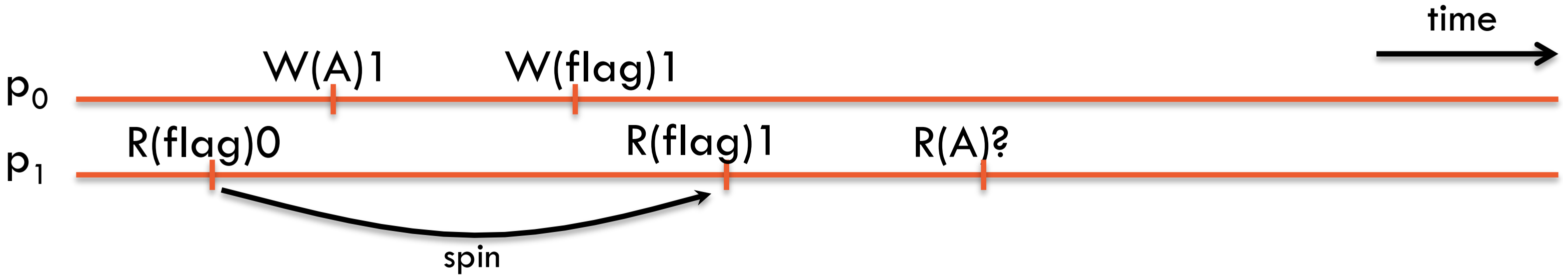
```
/* Initially, A = flag = 0 */
```

```
Process 0
A = 1;
flag = 1;
```

```
Process 1
while (!flag);  // spin
Print A;
```

Program order

- Example 4: What are the possible outputs of this concurrent program?



Sequential consistency

Application to shared memory multiprocessors

- How can a coherent execution may not be sequentially consistent

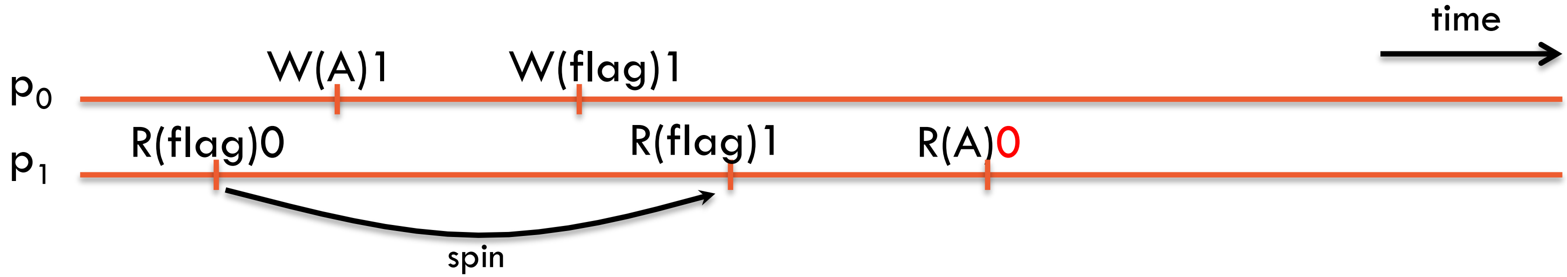
```
/* Initially, A = flag = 0 */
```

```
Process 0
A = 1;
flag = 1;
```

```
Process 1
while (!flag);  // spin
Print A;
```

Program order

- Example 4: output "0" is not (sequentially) consistent as we cannot find a sequential order that respects program order and prints 0.

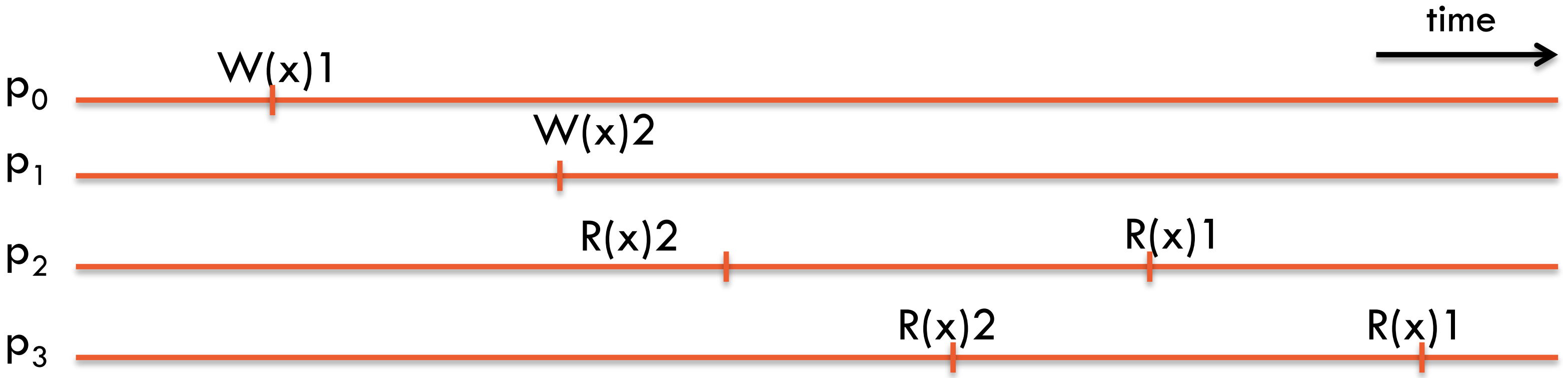


Causal Consistency

Causal consistency

Causally consistent executions

- *Causal consistency*. Writes that are *causally related* (i.e., one happens before the other) must be seen by all processes in the same order. Concurrent writes may be seen in a different order on different machines

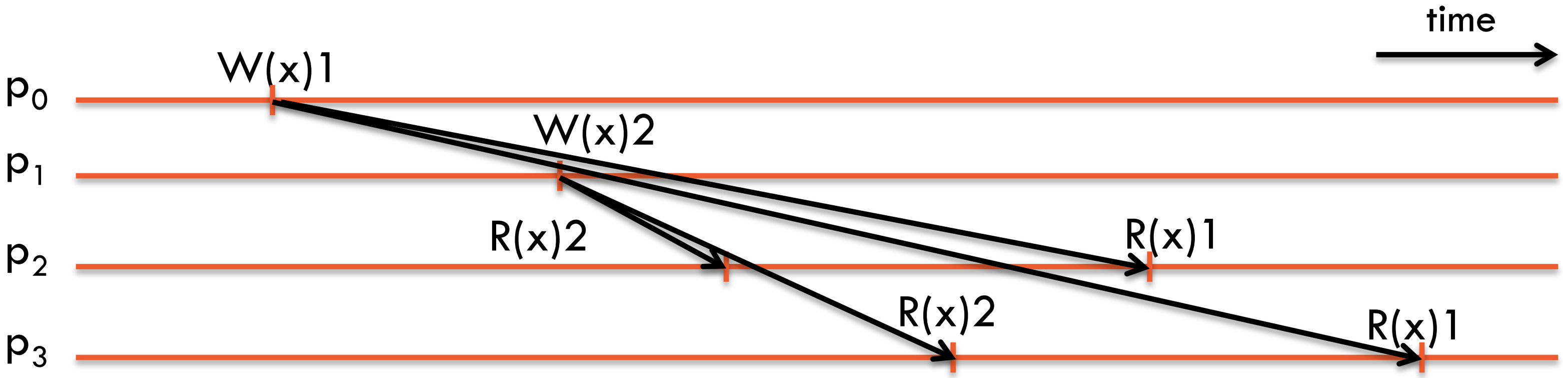


- Example 2(bis): is this execution causally consistent?

Causal consistency

Causally consistent executions (con’t)

- *Causal consistency.* Writes that are *causally related* (i.e., one happens before the other) must be seen by all processes in the same order. Concurrent writes may be seen in a different order on different machines

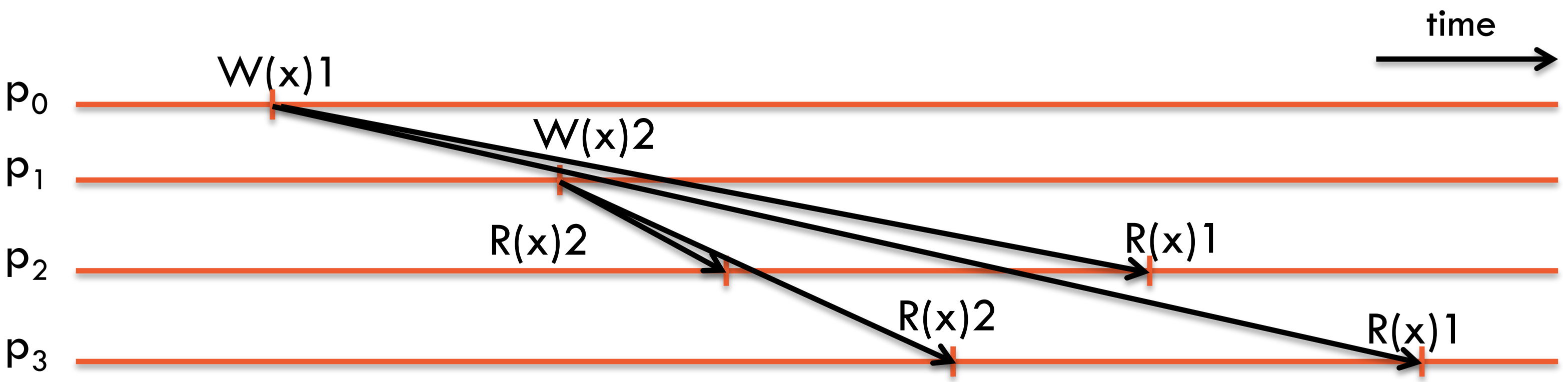


- Example 2(bis): is this execution causally consistent?
  - Write to the same data items are not necessarily causally related
  - A read returning the value written are causally related

Causal consistency

Causally consistent executions (con't)

- *Causal consistency.* Writes that are *causally related* (i.e., one happens before the other) must be seen by all processes in the same order. Concurrent writes may be seen in a different order on different machines



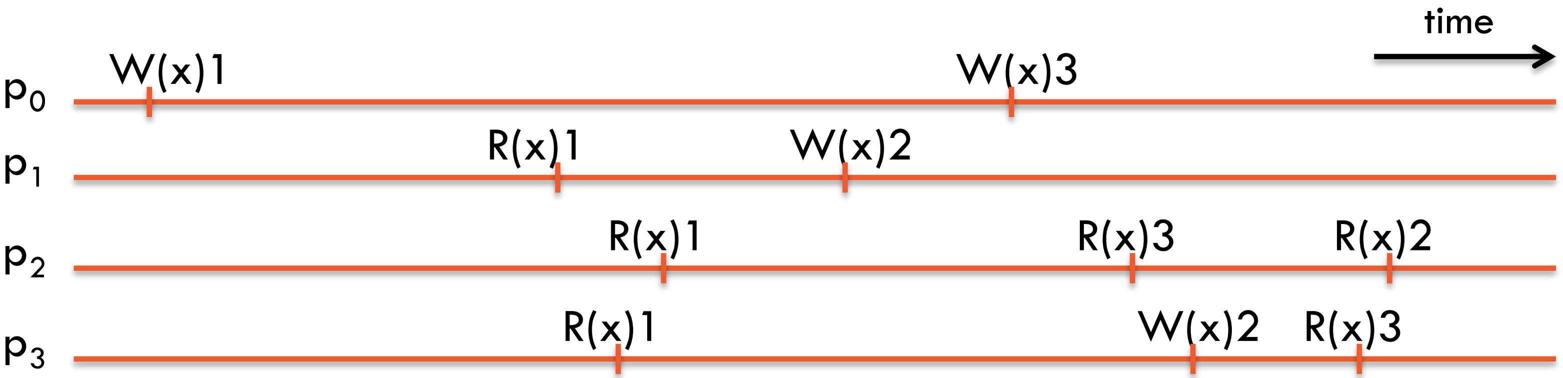
- Example 2(bis): the result of this serial execution is causally consistent

If the result of an execution if sequentially consistent, then it is also causally consistent

Causal consistency

Causally consistent executions (con't)

- *Causal consistency.* Writes that are *causally related* (i.e., one happens before the other) must be seen by all processes in the same order. Concurrent writes may be seen in a different order on different machines

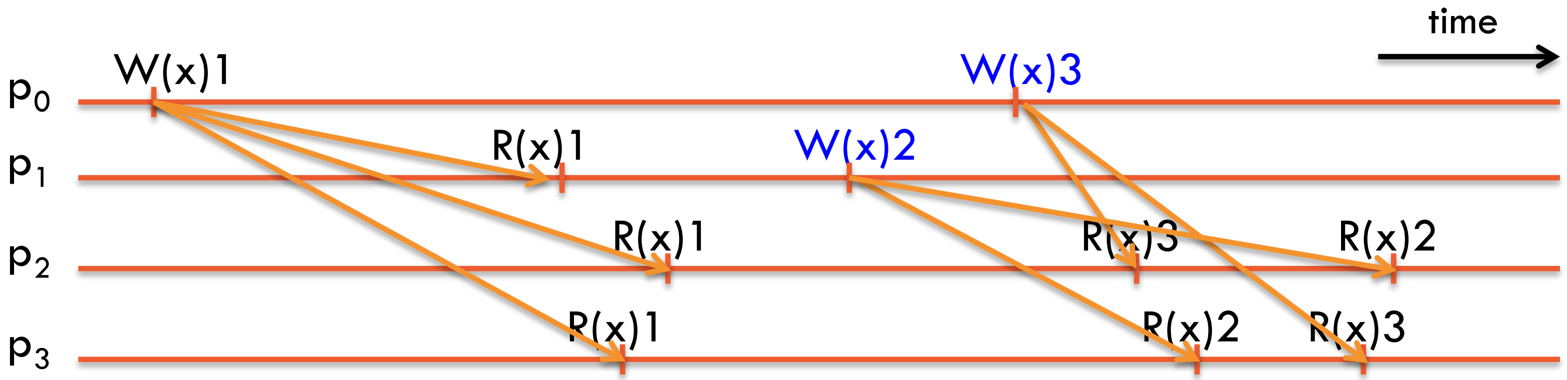


- Example 5: is the result of this execution is causally consistent?

Causal consistency

Causally consistent executions (con't)

- *Causal consistency.* Writes that are *causally related* (i.e., one happens before the other) must be seen by all processes in the same order. Concurrent writes may be seen in a different order on different machines



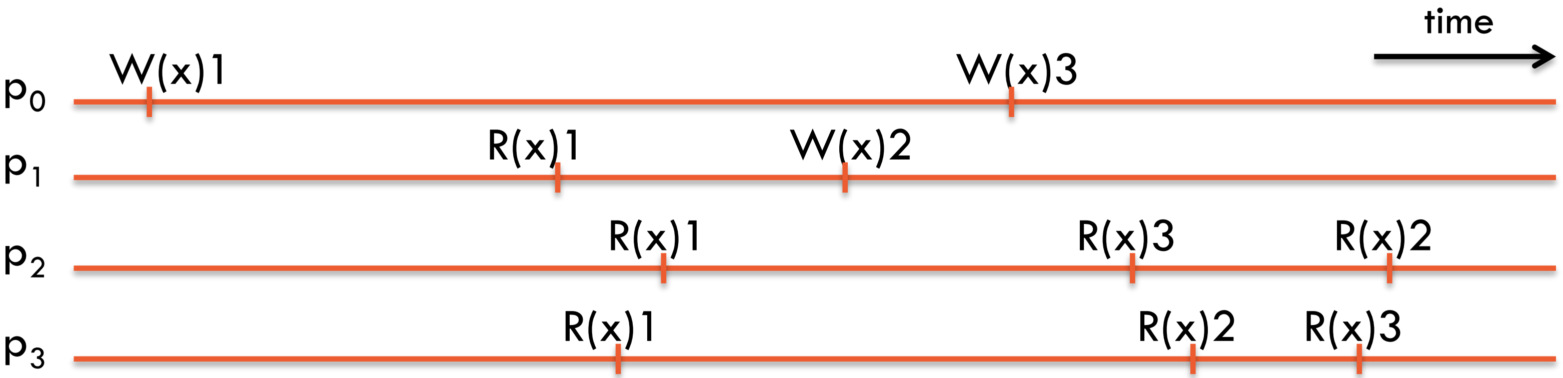
- Example 5: the result of this execution is causally consistent
  - $W_0(x)1$ and $W_0(x)3$ that are causally related are seen in the same order by p_0, p_1, p_2 and p_3
  - $W_1(x)2$ and $W_0(x)3$ are not causally related as none happen before the other

Vector clocks can be used to implement a causally consistent data store

Causal consistency

Causally consistent executions (con't)

- *Causal consistency.* Writes that are *causally related* (i.e., one happens before the other) must be seen by all processes in the same order. Concurrent writes may be seen in a different order on different machines

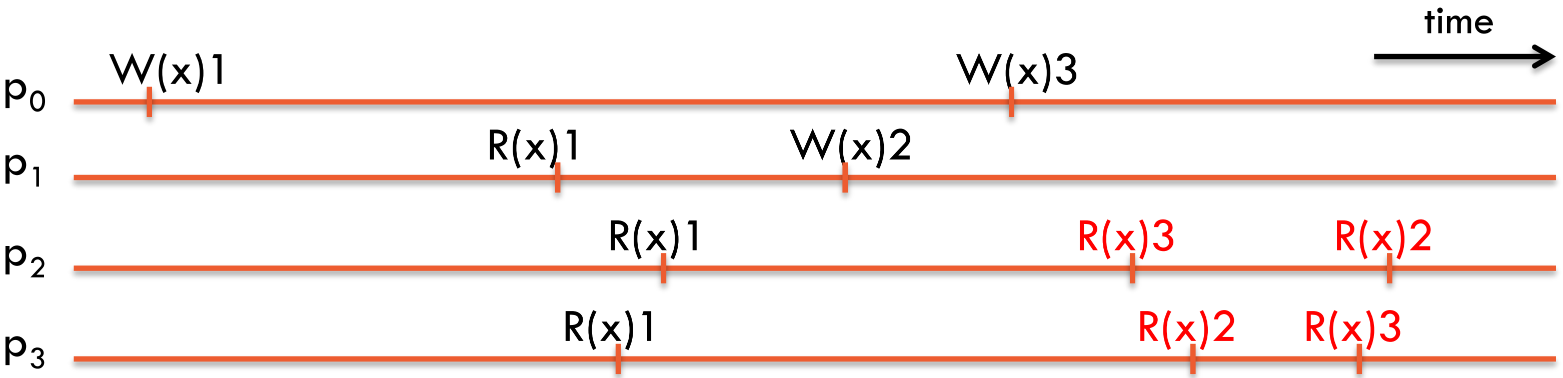


- Example 5: but is it is sequentially consistent?

Causal consistency

Causally consistent executions (con't)

- *Causal consistency.* Writes that are *causally related* (i.e., one happens before the other) must be seen by all processes in the same order. Concurrent writes may be seen in a different order on different machines



- Example 5: the result of this execution is not sequentially consistent

Causal consistency

Application to shared memory multiprocessors

- All commercialized memory models guarantee causal consistency

```
/* Initially, A = flag1 = flag2 = r3 = 0 */
```

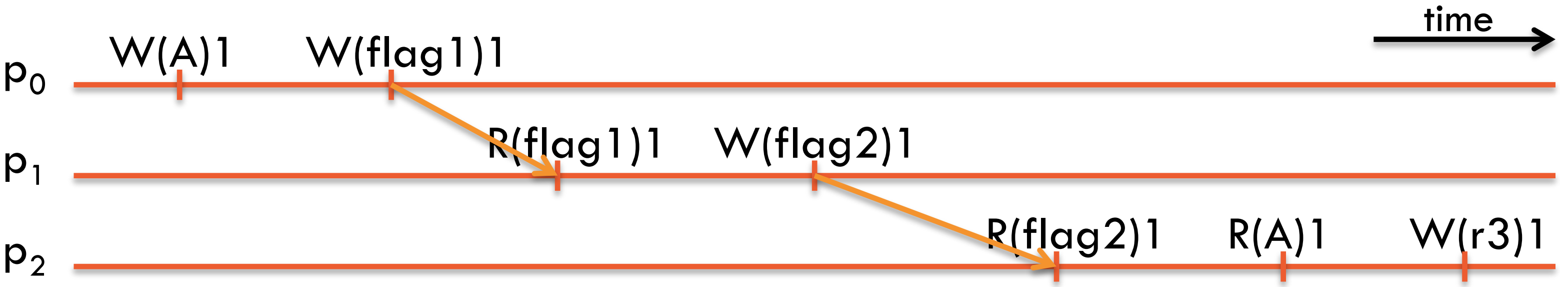
```
Process 0
A = 1;
flag1 = 1;
```

```
Process 1
while (!flag1); // spin
flag2 = 1;
```

```
Process 2
while (!flag2);  // spin
r3 = A;
```

Program order

- Causal consistency implies that at the end r3 = 1, because all the writes are causally related (we can order them with the *happen-before* relation)



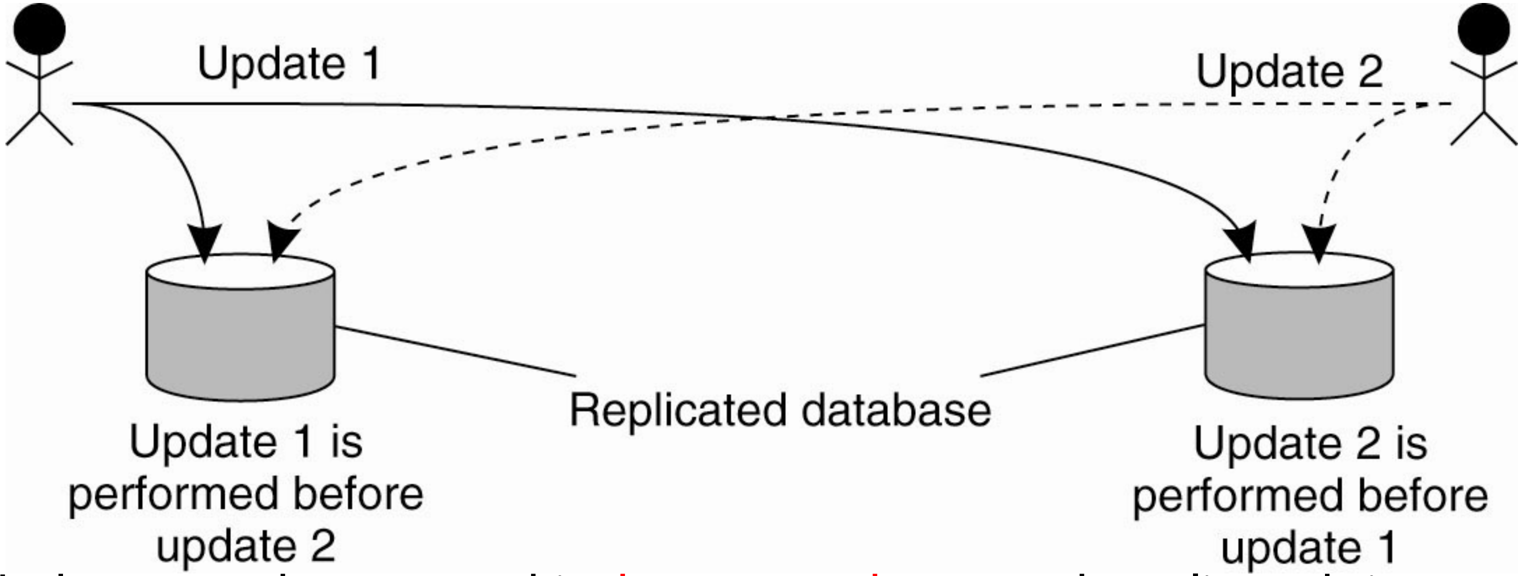
Replication

THE UNIVERSITY OF SYDNEY

Replication

Updating a replicated system

- Writes operations can be carried out to several sites (like in the database example)


Update 1
Update 2
Replicated database
Update 1 is performed before update 2
Update 2 is performed before update 1

- Updates must be executed in the same order at each replicated site
- The use of totally-ordered broadcast (cf., previous lecture) can solve this
- However, our previous Lamport's clock solution does not scale well

Replication

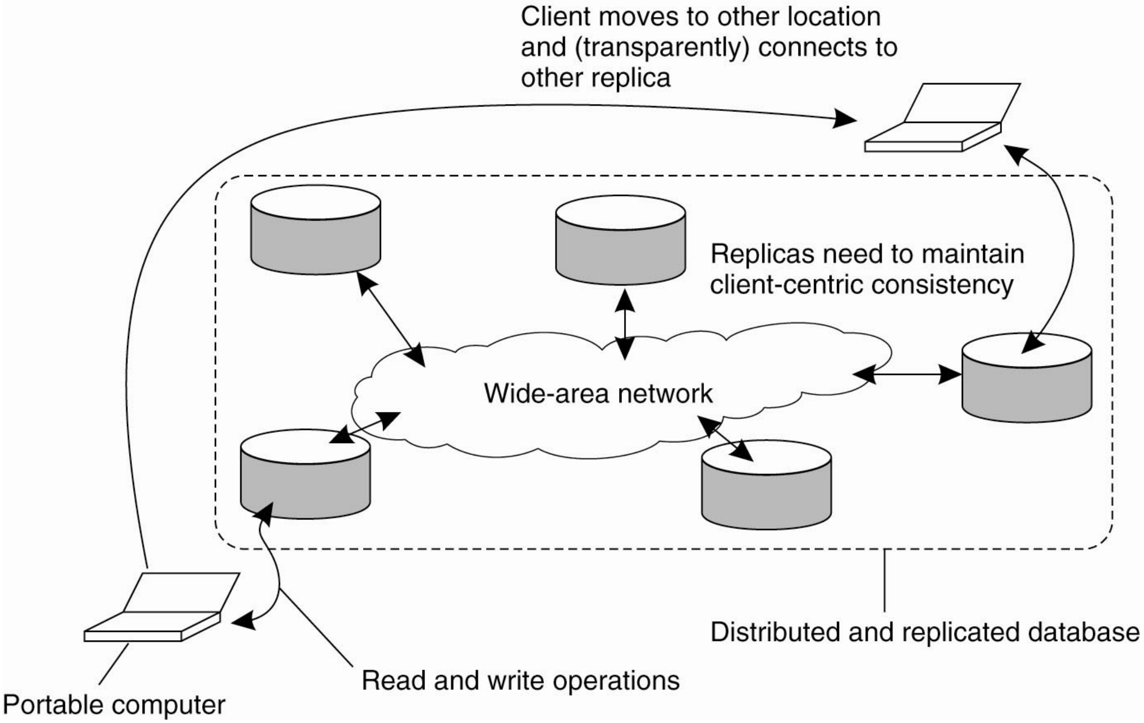
Eventual consistency: eventually all replicas are consistent

- Assumption: no concurrent updates
  - The only type of conflicts occur between a reading operation and an updating operation (read-write conflict)
  - Never between two updating operations (no write-write conflicts)
- Example:
  - A web server is the only one to update the webpage content
  - Clients typically access the webpage in read mode only (without modifying it)

Replication

Eventual consistency (con’t)

- A mobile user accesses a database by connecting to one of its replicas in a transparent way (the user’s application is unaware of which replica)
- The user executes several update requests and disconnects
- The user then reconnects from a different location or through a different device.
- If the changes have not been propagated by the system to all replicas, the user may observe a client-centric inconsistency


Client moves to other location and (transparently) connects to other replica
Replicas need to maintain client-centric consistency
Wide-area network
Distributed and replicated database
Read and write operations
Portable computer

Replication

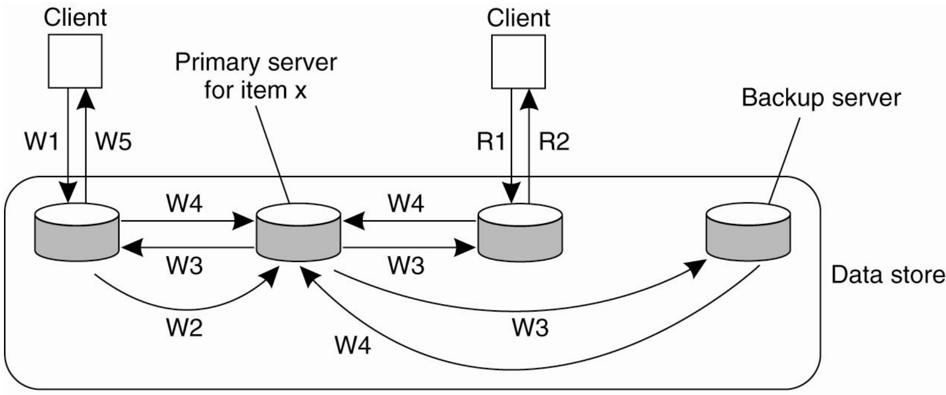
Coordinator

- Active replication
  - Forward all messages to a central coordinator (leader)
  - The coordinator is a *sequencer* that chooses a unique sequence number for each message
  - It sends this sequence number along with the message to all replicas
  - Operations are carried out in the order of their sequence number

Replication

Primary-backup

- Primary-backup protocol [Budhiuraja et al., 1993]
  - All write operations on x are forwarded to a primary server for x
  - The primary does the update and forwards the request to backup servers
  - Each backup executes the updates and acknowledges the primary
  - The primary sends a response to the client
  - Read operations can be carried out locally



W1. Write request
W2. Forward request to primary
W3. Tell backups to update
W4. Acknowledge update
W5. Acknowledge write completed

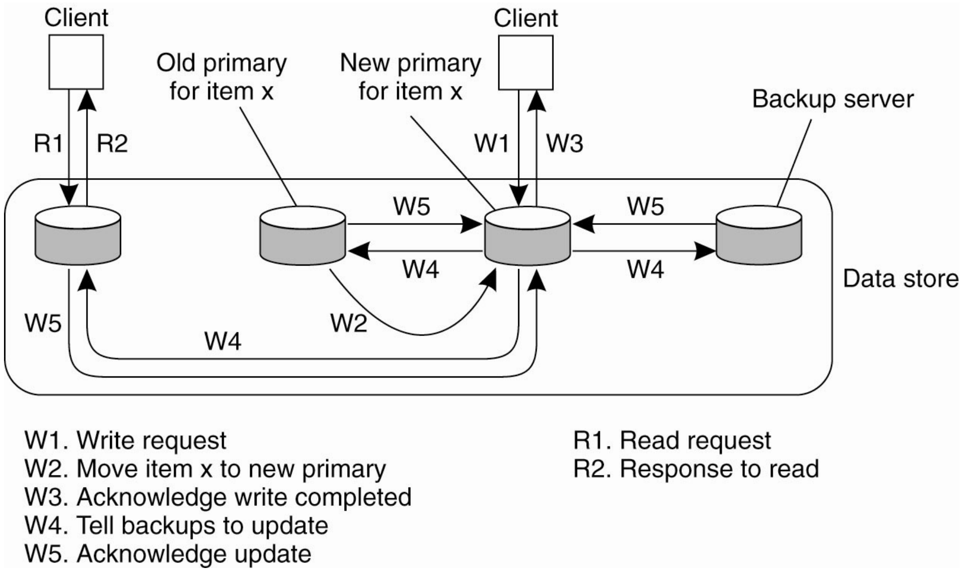
R1. Read request
R2. Response to read

- Pro: guarantees sequential consistency
- Cons: delayed answer to client due to blocking execution

Replication

Primary-backup

- Local-write primary-backup
  - When a process wants to update a data item x it locates it
  - It moves the item to its location, becoming the new primary and updates the item locally
  - It acknowledges the client
  - It tells the backup to execute the update locally
  - The backups acknowledge the new primary
- Pro: does not block



Replication

Quorum system [Gifford 1979, Thomas 1979]

- *Quorum system*: a set S of quorums that are mutually intersecting sets
- Property: for any pair of quorums Q1, Q2 in the quorum system S we have: Q1 ∩ Q2 ≠ ∅.

- Key idea:
  - Consider a single quorum system
  - Write:
    - forward the update request along with a version that is a pair of timestamp and node identifier, to (all replicas of) a quorum
    - Once acknowledged by a quorum of replicas, respond to client
  - Read:
    - When a read on x is requested, forward the request to all replicas belonging to a quorum and picks the response with the maximum timestamp
    - Return the chosen copy to the client

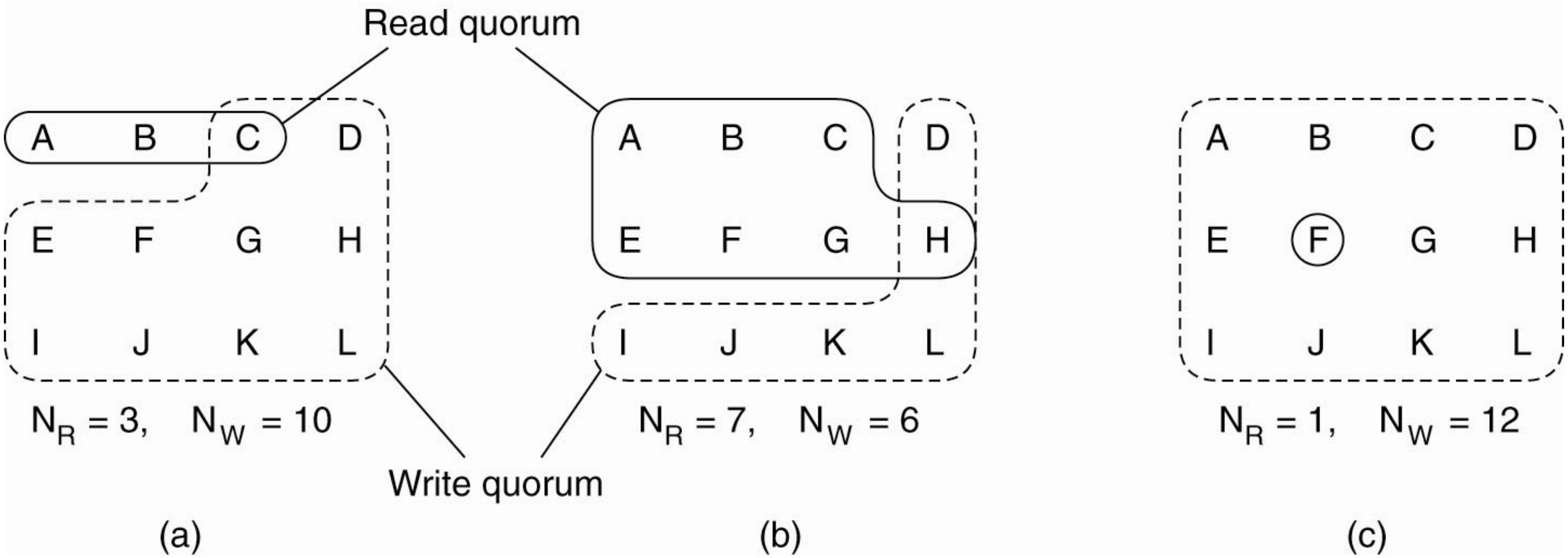
- Nodes have to know about the quorum system

Replication

Quorum system [Gifford 1979, Thomas 1979]

- Gifford's approach: Consider N replicas.
  - Upon read request, forward the request to at least N_R servers (a *read quorum*)
  - Upon writing, forward the request to at least N_W servers (a *write quorum*) such that:

$N_R + N_W > N$ (each read quorum intersects with each write quorum) and
$N_W > N/2$ (each write quorum intersects with each write quorum)



(b) Cannot work as $N_W = N/2$

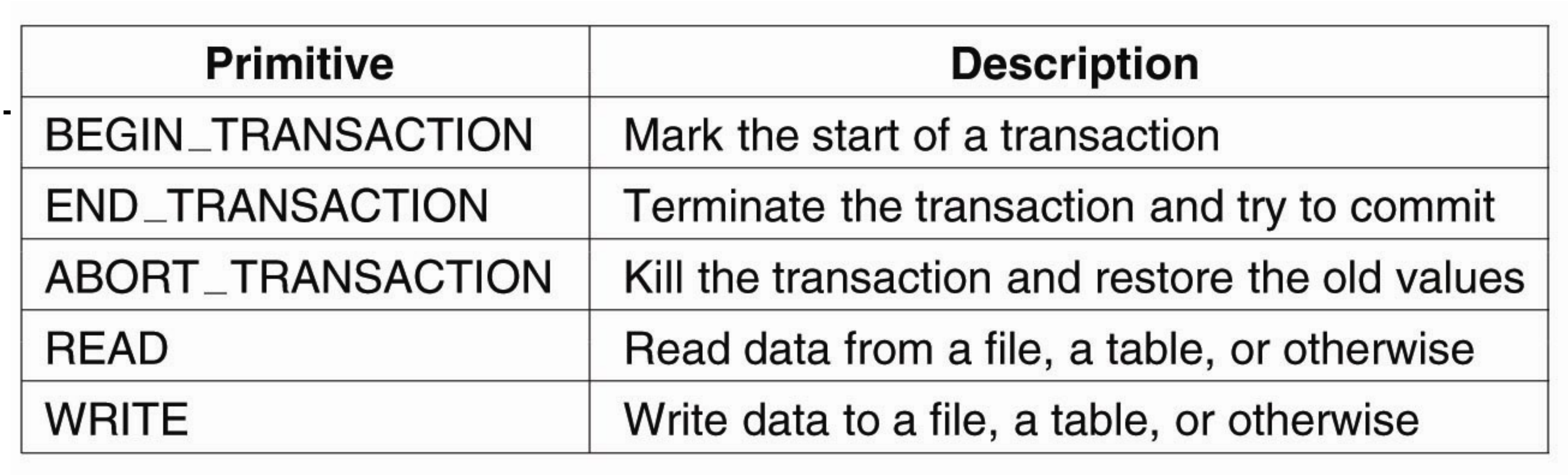
Multiple-Access Operations

Multi-access operations

Definition

- Any single operation can now access multiple objects (e.g., *transaction*)
- We have to consider new consistency criteria (previous ones do not apply)

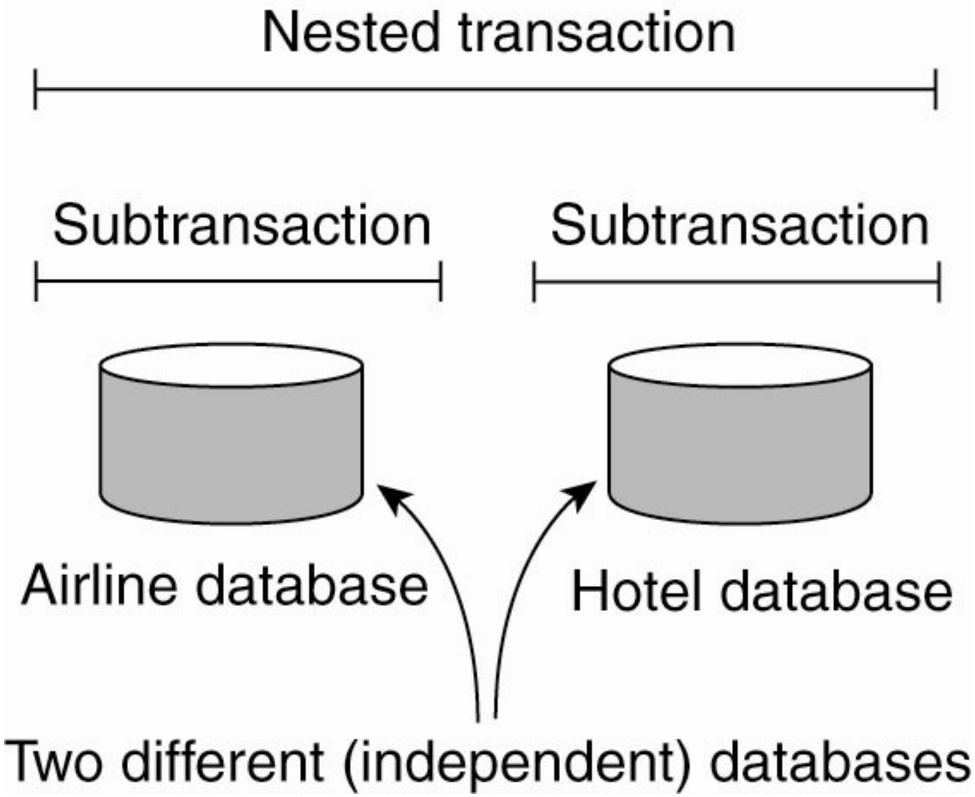
-



Primitive	Description
BEGIN_TRANSACTION	Mark the start of a transaction
END_TRANSACTION	Terminate the transaction and try to commit
ABORT_TRANSACTION	Kill the transaction and restore the old values
READ	Read data from a file, a table, or otherwise
WRITE	Write data to a file, a table, or otherwise

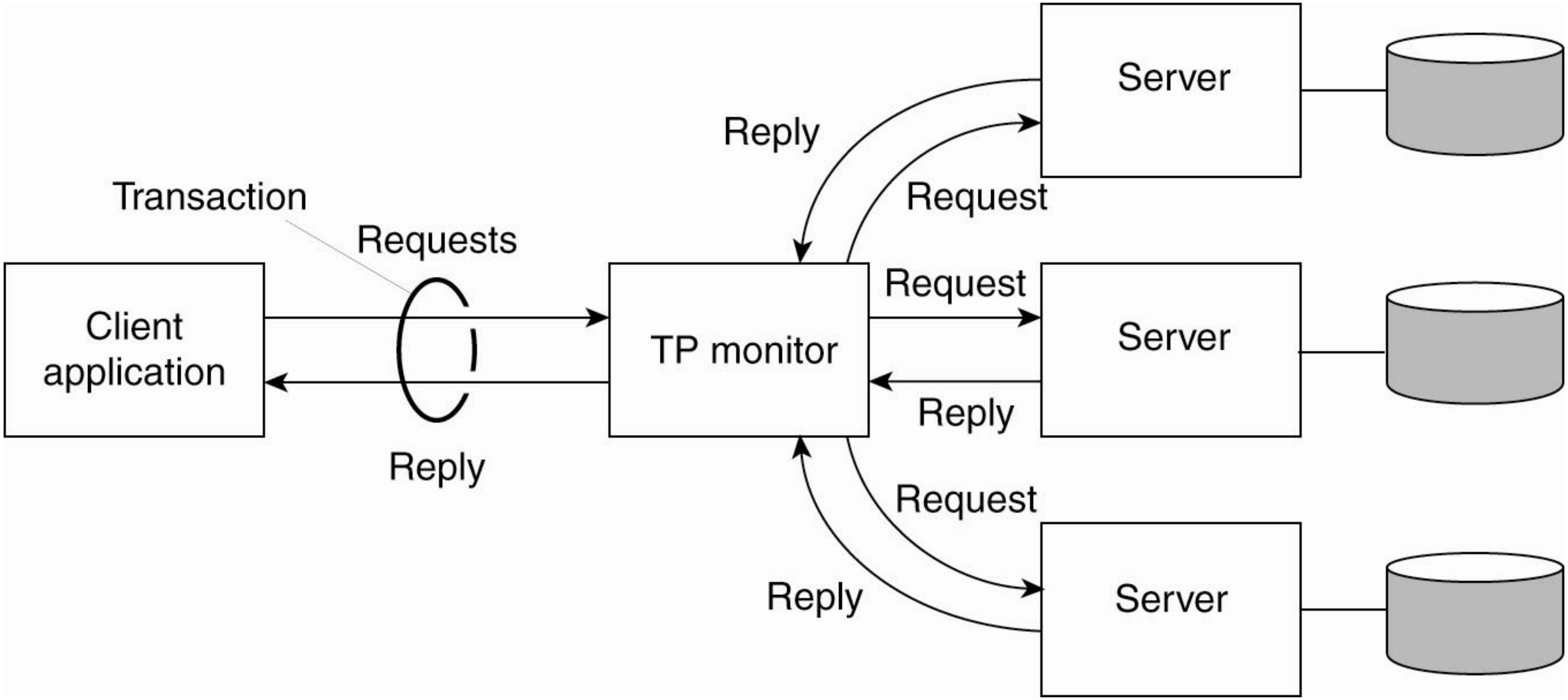
Database transactions

- Nested transactions



Database transactions

- The role of a transaction processing monitor in distributed systems



Database transactions

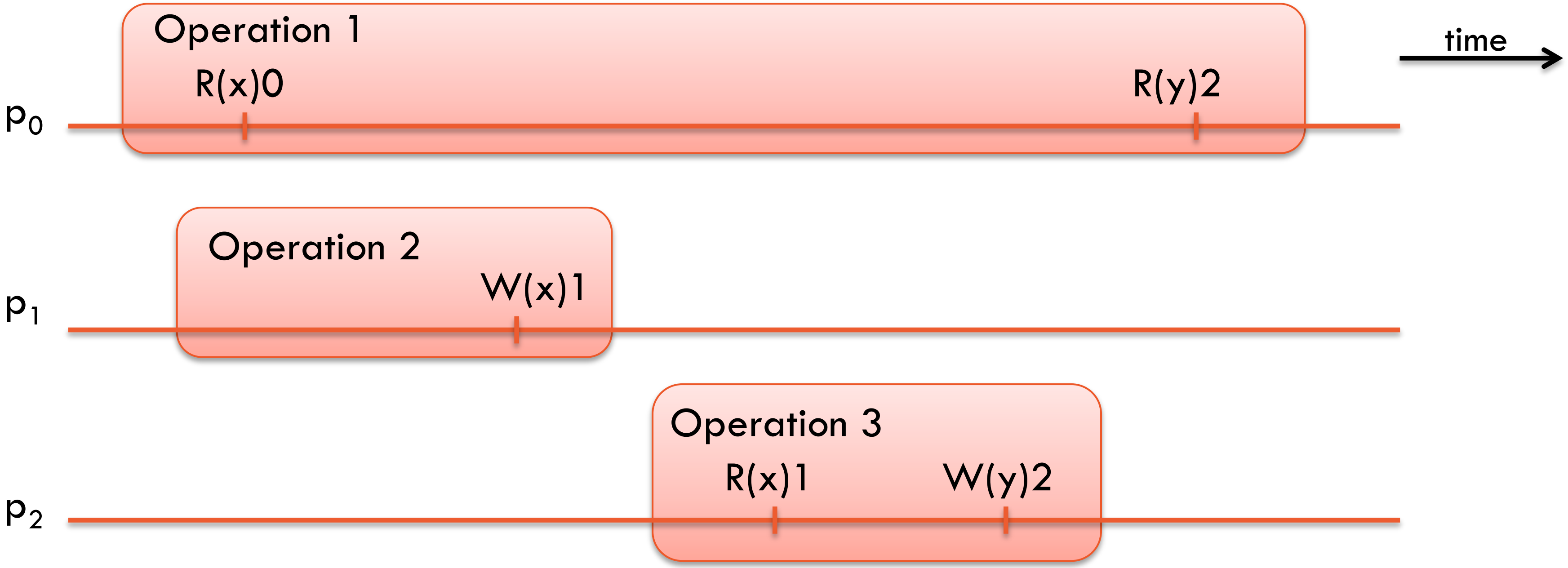
Properties

- Characteristic properties of transactions:
  - Atomic: To the outside world, the transaction happens indivisibly
  - Consistent: The transaction does not violate system invariants
  - Isolated: Concurrent transactions do not interfere with each other
  - Durable: Once a transaction commits, the changes are permanent
- These are named the *ACID* properties

Serializability

Serializability [Papadimitriou, JACM 1979]

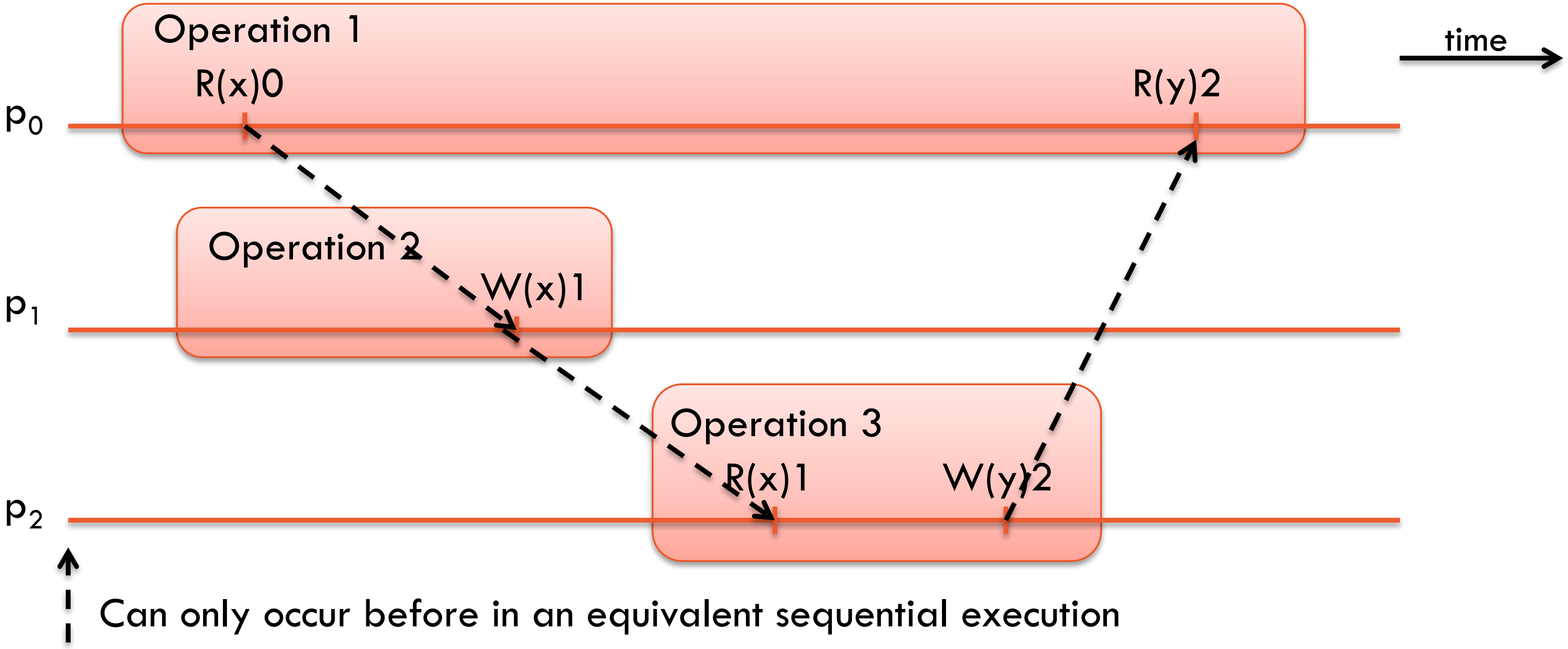
- *Serializability*: the result of an execution (of multi-access operations) is serializable if there exists an equivalent sequential execution.



Serializability

Serializability

- *Serializability:* the result of an execution (of multi-access operations) is serializable if there exists an equivalent sequential execution.



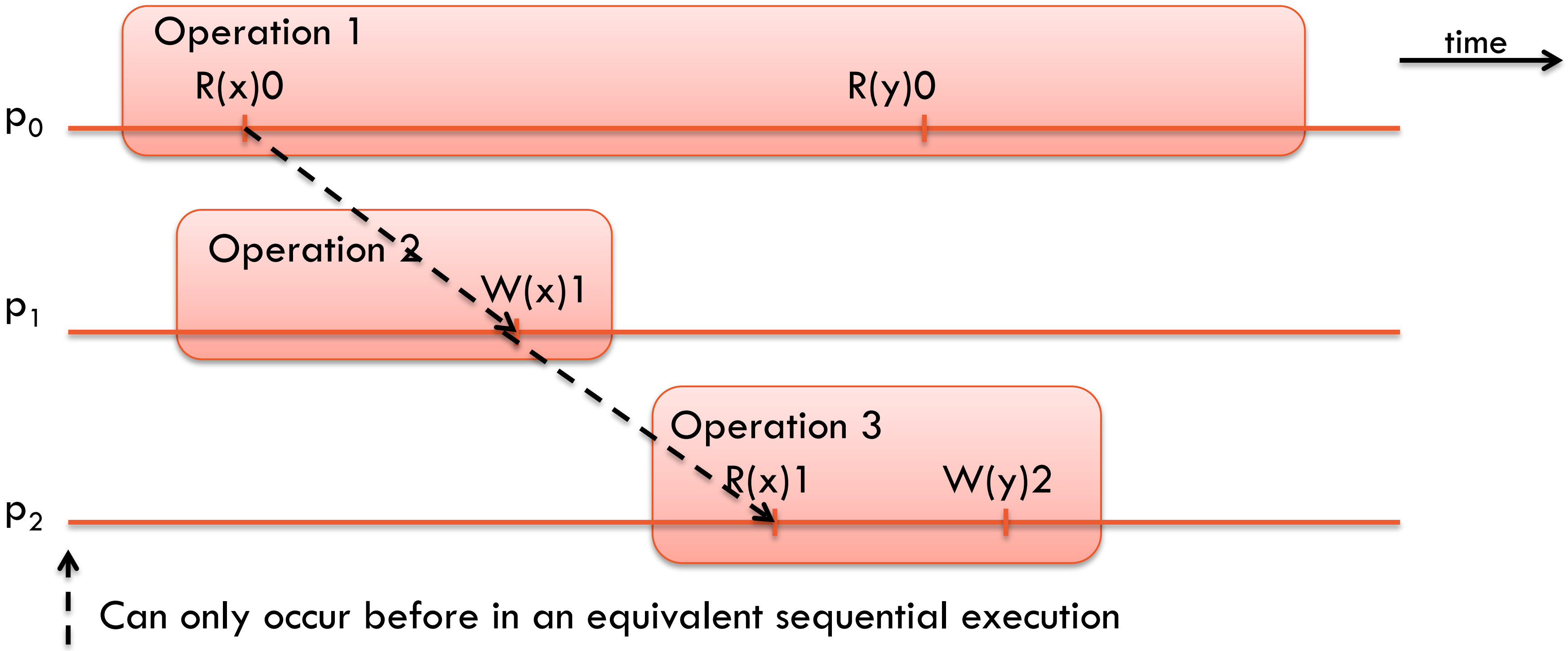
Can only occur before in an equivalent sequential execution

There is a cycle op1- ->op2- ->op3- ->op1 in the precedence graph ⇒ non-serializable

Serializability

Serializability

- *Serializability*: the result of an execution (of multi-access operations) is serializable if there exists an equivalent sequential execution.



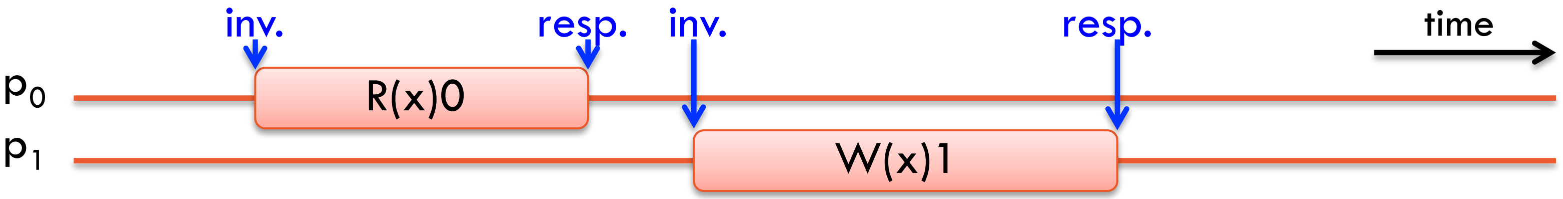
Can only occur before in an equivalent sequential execution

No cycle op1- ->op2- ->op3 in the precedence graph ⇒ serializable

Linearizability

Linearizability (of reads and writes) [Herlihy and Wing, TOPLAS 1990]

- *Real-time precedence:* if an operation o_1 returns before another operation o_2 is invoked (at a potentially different process) then o_1 *precedes* o_2 *with respect (w.r.t) to real-time.*



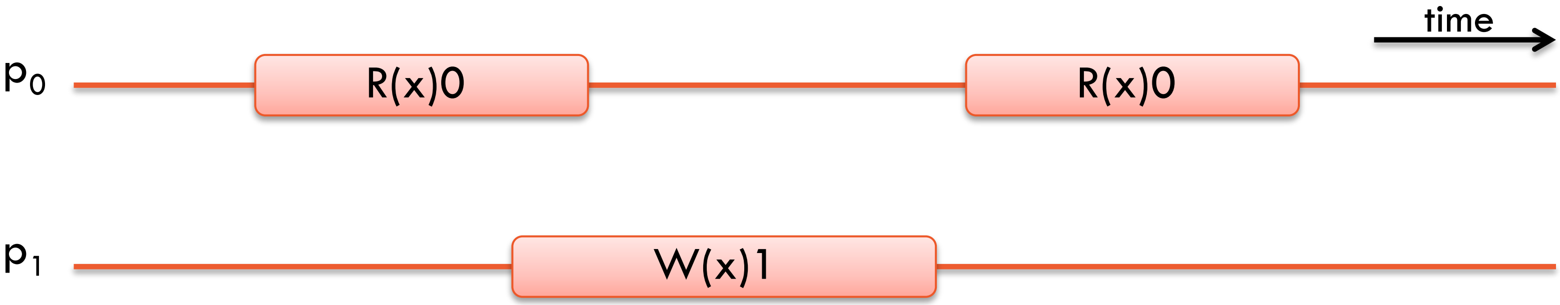
$R_0(x)0$ precedes $W_1(x)1$ w.r.t real time

- *Linearizability*. The result of each execution is
  - equivalent to a sequential execution that respects the real-time precedence (i.e., in which an operation returning before another is invoked is always ordered before)
  - is the same as if the (read and write) operations by all processes on the same data store were executed in some sequential order and the operations of each individual process appear in this sequence in its program order

Linearizability

Linearizability (of reads and writes)

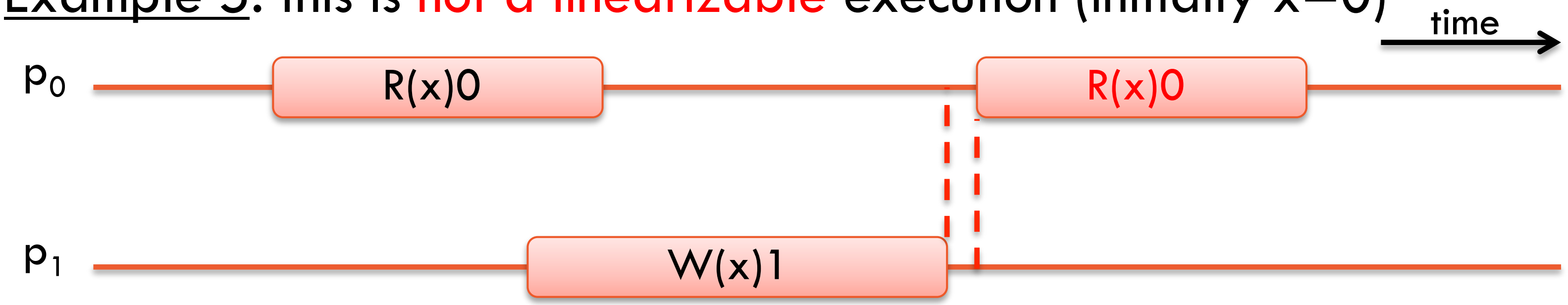
- *Linearizability.* The result of each execution is
  - equivalent to a sequential execution that respects the real-time precedence (i.e., in which an operation returning before another is invoked is always ordered before)
  - is the same as if the (read and write) operations by all processes on the same data store were executed in some sequential order and the operations of each individual process appear in this sequence in its program order
- Example 5: is this a linearizable execution? (initially x=0)



Linearizability

Linearizability (of reads and writes)

- *Linearizability*. The result of each execution is
  - equivalent to a sequential execution that respects the real-time precedence (i.e., in which an operation returning before another is invoked is always ordered before)
  - is the same as if the (read and write) operations by all processes on the same data store were executed in some sequential order and the operations of each individual process appear in this sequence in its program order
- Example 5: this is not a linearizable execution (initially x=0)

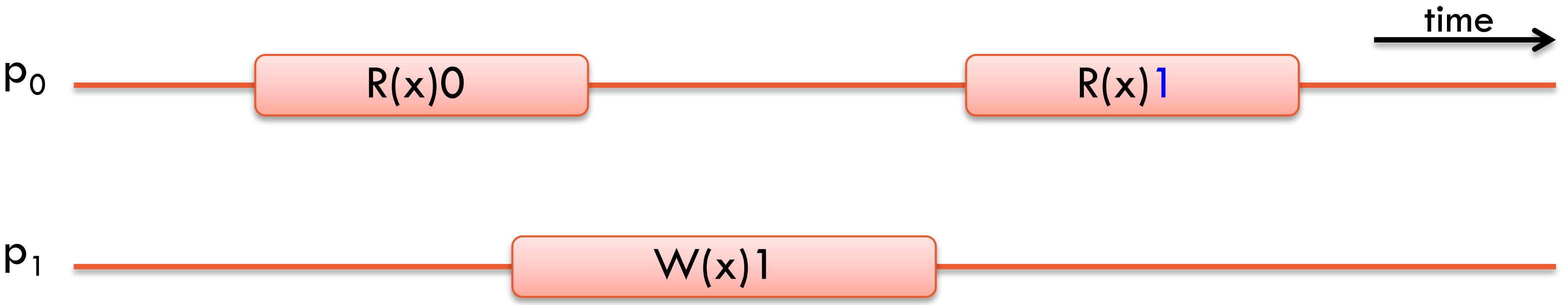


- $W_1(x)1$ precedes $R_0(x)0$ w.r.t real-time, thus $R_0(x)0$ cannot return the initial value of x.

Linearizability

Linearizability (of reads and writes)

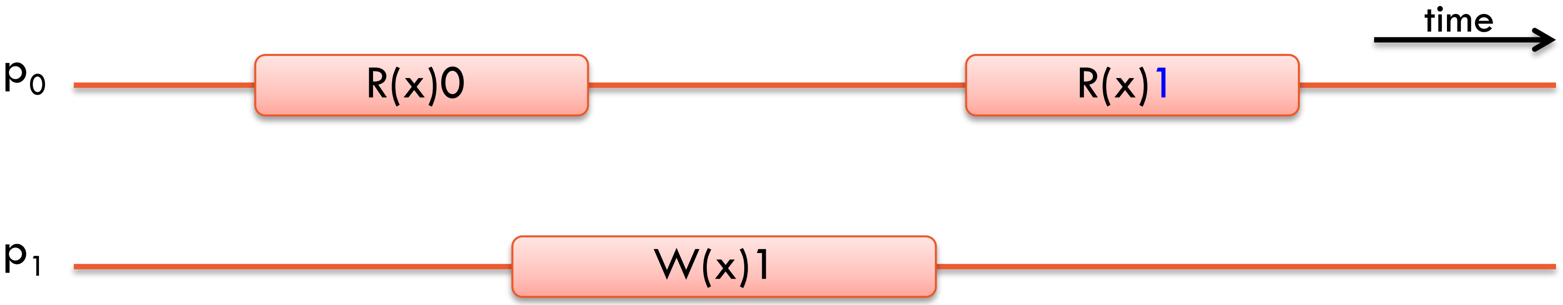
- *Linearizability.* The result of each execution is
  - equivalent to a sequential execution that respects the real-time precedence (i.e., in which an operation returning before another is invoked is always ordered before)
  - is the same as if the (read and write) operations by all processes on the same data store were executed in some sequential order and the operations of each individual process appear in this sequence in its program order
- Example 6: is this a linearizable execution? (initially x=0)



Linearizability

Linearizability (of reads and writes)

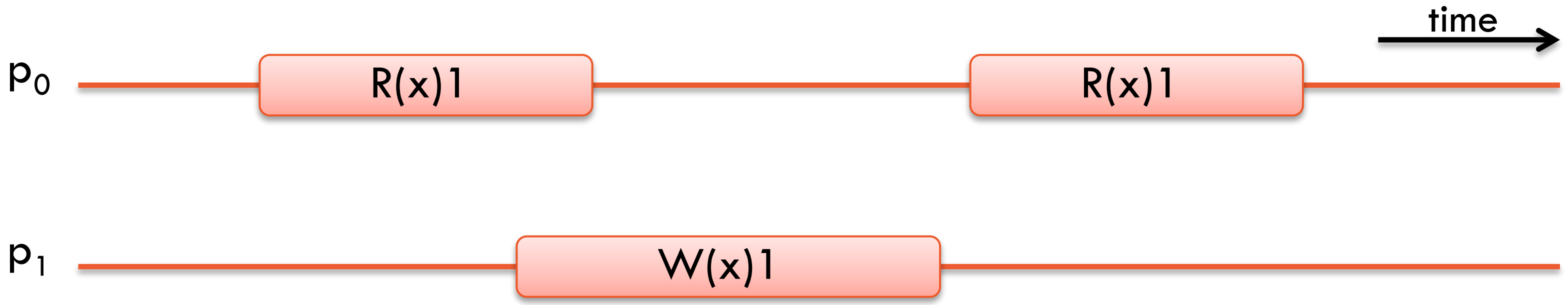
- *Linearizability.* The result of each execution is
  - equivalent to a sequential execution that respects the real-time precedence (i.e., in which an operation returning before another is invoked is always ordered before)
  - is the same as if the (read and write) operations by all processes on the same data store were executed in some sequential order and the operations of each individual process appear in this sequence in its program order
- Example 6: this is a linearizable execution (initially x=0)



Linearizability

Linearizability (of reads and writes)

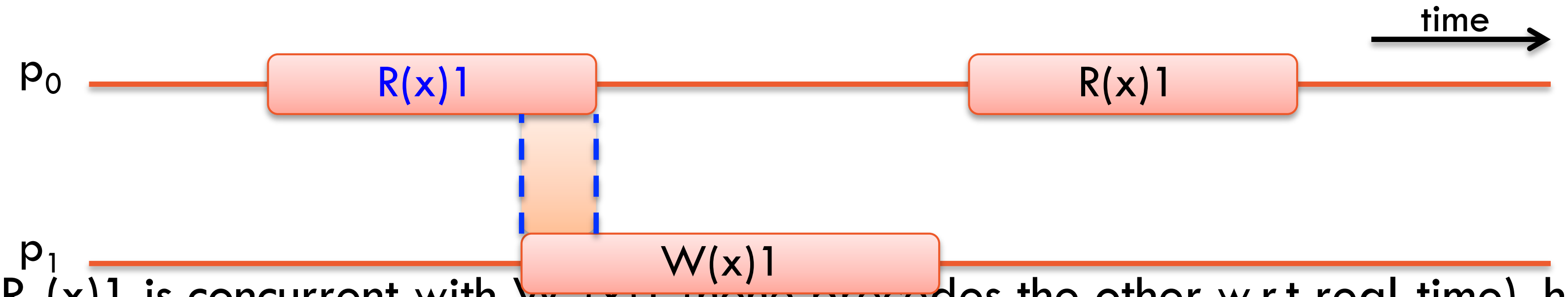
- *Linearizability.* The result of each execution is
  - equivalent to a sequential execution that respects the real-time precedence (i.e., in which an operation returning before another is invoked is always ordered before)
  - is the same as if the (read and write) operations by all processes on the same data store were executed in some sequential order and the operations of each individual process appear in this sequence in its program order
- Example 7: is this a linearizable execution? (initially x=0)



Linearizability

Linearizability (of reads and writes)

- *Linearizability.* The result of each execution is
  - equivalent to a sequential execution that respects the real-time precedence (i.e., in which an operation returning before another is invoked is always ordered before)
  - is the same as if the (read and write) operations by all processes on the same data store were executed in some sequential order and the operations of each individual process appear in this sequence in its program order
- Example 7: this is a linearizable execution (initially x=0)



- $R_0(x)1$ is concurrent with $W_1(x)1$ (none precedes the other w.r.t real-time), hence $R_0(x)$ can return 1 (or 0) without violating linearizability

Linearizability

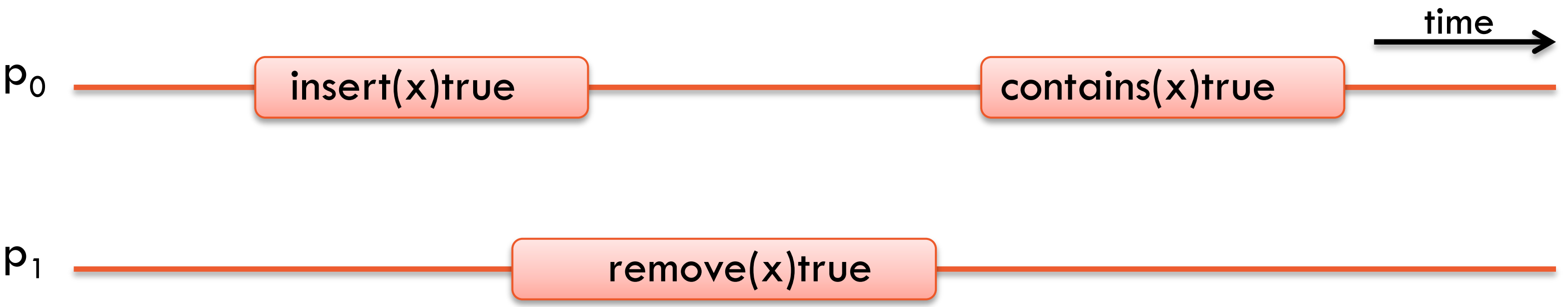
Linearizability (of higher level operations)

- Linearizability can also apply to other kind of operations.

Consider an integer set object that exports (as opposed to the previous register exporting read/write):

- insert(int a)boolean b that adds an element to the set if not already present and return true, returns false if already present
- delete(int a)boolean b that removes an integer from the set if it is currently present and returns true, false if not present
- contains(int a)boolean b that checks whether an integer is present in the set

- Example 8: is this a linearizable execution of set operations? (initially x ∉ set)



Linearizability

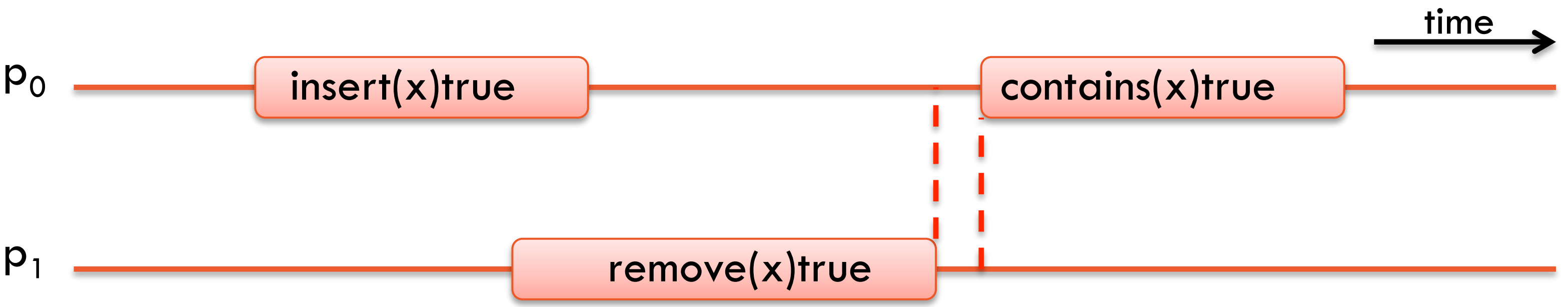
Linearizability (of higher level operations)

- Linearizability can also apply to other kind of operations.

Consider an integer set object that exports (as opposed to the previous register exporting read/write):

- insert(int a)boolean b that adds an element to the set if not already present and return true, returns false if already present
- delete(int a)boolean b that removes an integer from the set if it is currently present and returns true, false if not present
- contains(int a)boolean b that checks whether an integer is present in the set

- Example 8: this is not a linearizable execution of set operations (initially x ∉ set)



Linearizability

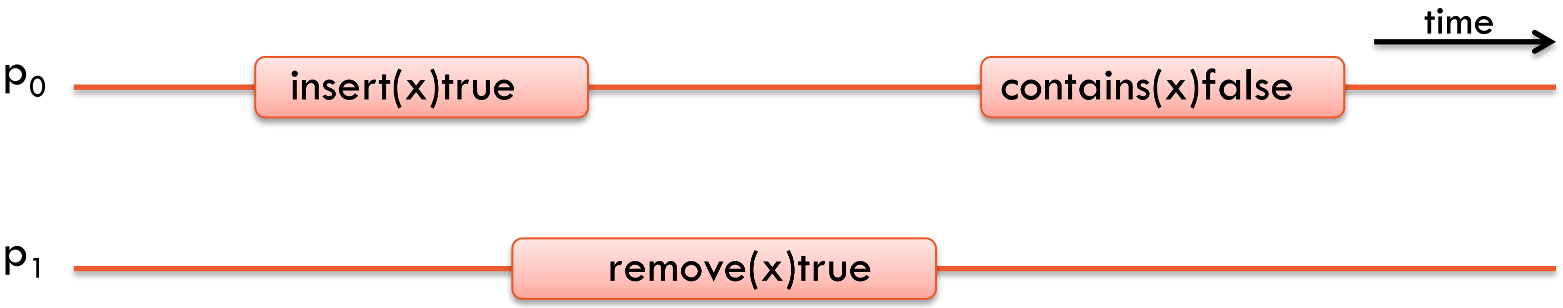
Linearizability (of higher level operations)

- Linearizability can also apply to other kind of operations.

Consider an integer set object that exports (as opposed to the previous register exporting read/write):

  - insert(int a)boolean b that adds an element to the set if not already present and return true, returns false if already present
  - delete(int a)boolean b that removes an integer from the set if it is currently present and returns true, false if not present
  - contains(int a)boolean b that checks whether an integer is present in the set

- Example 9: is this a linearizable execution of set operations? (initially $x \notin$ set)



Linearizability

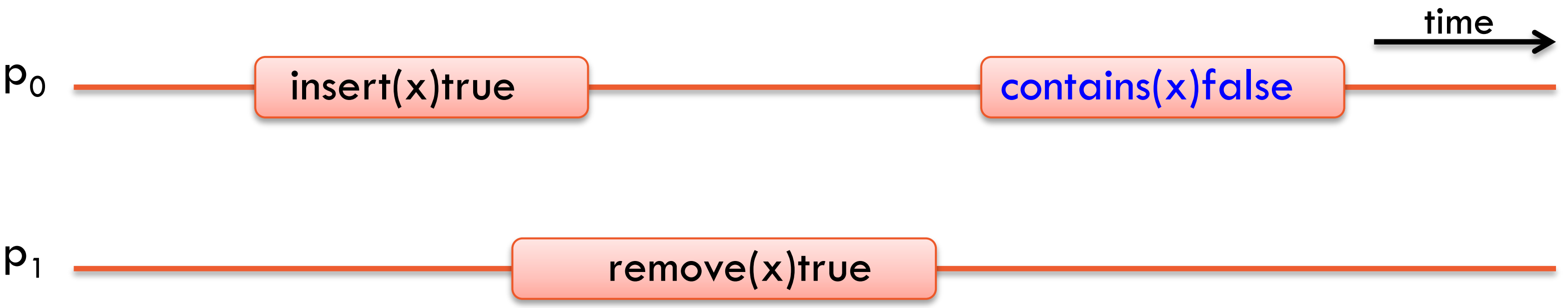
Linearizability (of higher level operations)

- Linearizability can also apply to other kind of operations.

Consider an integer set object that exports (as opposed to the previous register exporting read/write):

  - insert(int a)boolean b that adds an element to the set if not already present and return true, returns false if already present
  - delete(int a)boolean b that removes an integer from the set if it is currently present and returns true, false if not present
  - contains(int a)boolean b that checks whether an integer is present in the set

- Example 9: this is a linearizable execution of set operations (initially x ∉ set)



Linearizability

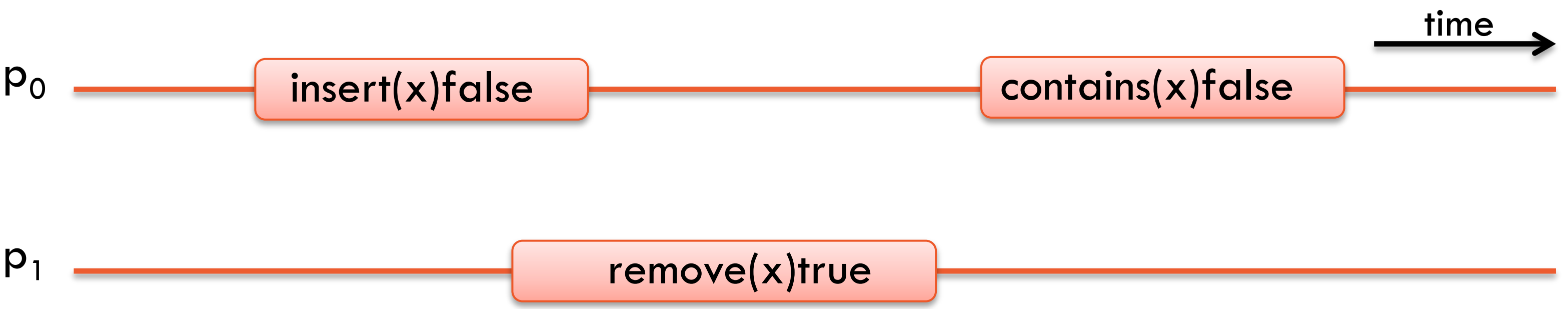
Linearizability (of higher level operations)

- Linearizability can also apply to other kind of operations.

Consider an integer set object that exports (as opposed to the previous register exporting read/write):

  - insert(int a)boolean b that adds an element to the set if not already present and return true, returns false if already present
  - delete(int a)boolean b that removes an integer from the set if it is currently present and returns true, false if not present
  - contains(int a)boolean b that checks whether an integer is present in the set

- <u>Example 10</u>: is this a linearizable execution of set operations? (initially x ∉ set)



Linearizability

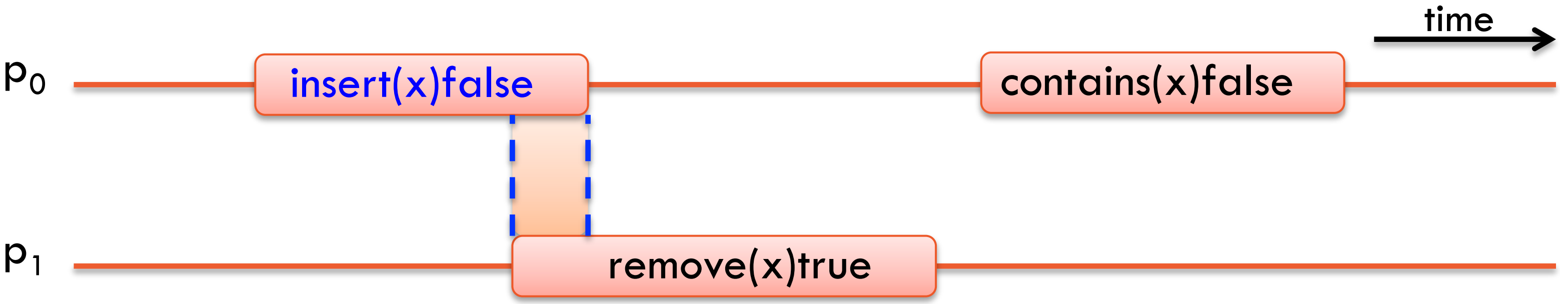
Linearizability (of higher level operations)

- Linearizability can also apply to other kind of operations.

Consider an integer set object that exports (as opposed to the previous register exporting read/write):

- insert(int a)boolean b that adds an element to the set if not already present and return true, returns false if already present
- delete(int a)boolean b that removes an integer from the set if it is currently present and returns true, false if not present
- contains(int a)boolean b that checks whether an integer is present in the set

- Example 10: this is a linearizable execution of set operations (initially x ∈ set)



Linearizability

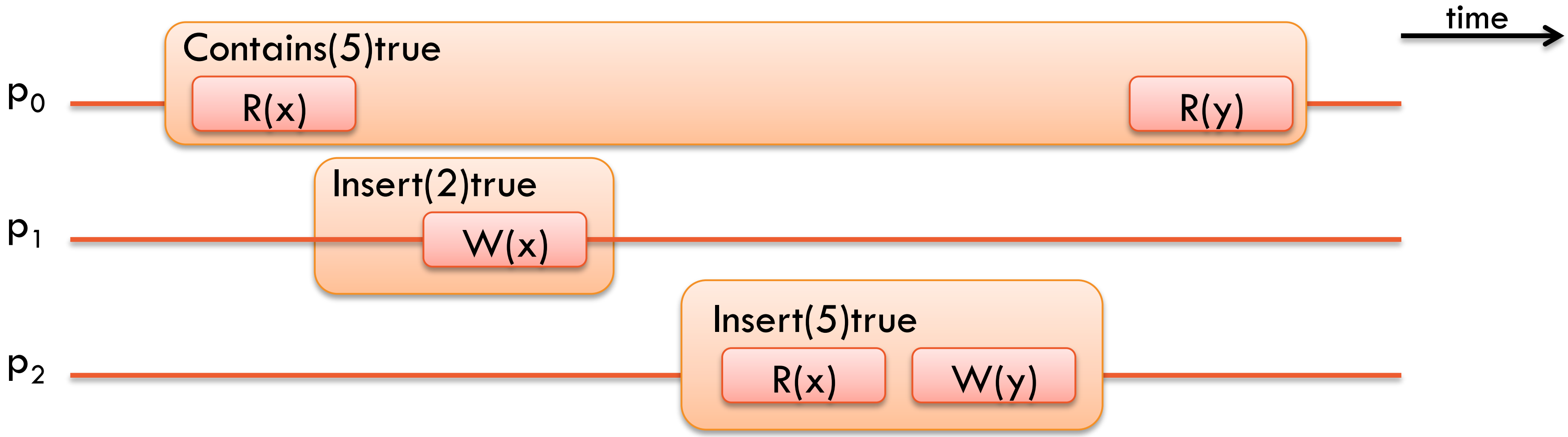
Linearizability (of higher level operations)

- Linearizability can also apply to other kind of operations.

Consider an integer set object that exports (as opposed to the previous register exporting read/write)

  - insert(int a)boolean b that adds an element to the set if not already present and return true, returns false if already present
  - delete(int a)boolean b that removes an integer from the set if it is currently present and returns true, false if not present
  - contains(int a)boolean b that checks whether an integer is present in the set

- Example 11: this is linearizable at the set level but not linearizable at the register level



Conclusion

- There are several consistency criteria
  - Strong criteria (restrictive):
    - provides lots of safety guarantees (are enough for most safety requirements)
    - provides low performance
  - Weak criteria (not restrictive):
    - provides less safety guarantees (can be enough for specific applications)
    - gives high performance

- Some criteria strength are comparable (from the weakest to the strongest):
  - causal consistency < sequential consistency
  - serial ⇒ sequential consistent ⇒ causally consistent
  - serializable ≠ linearizability

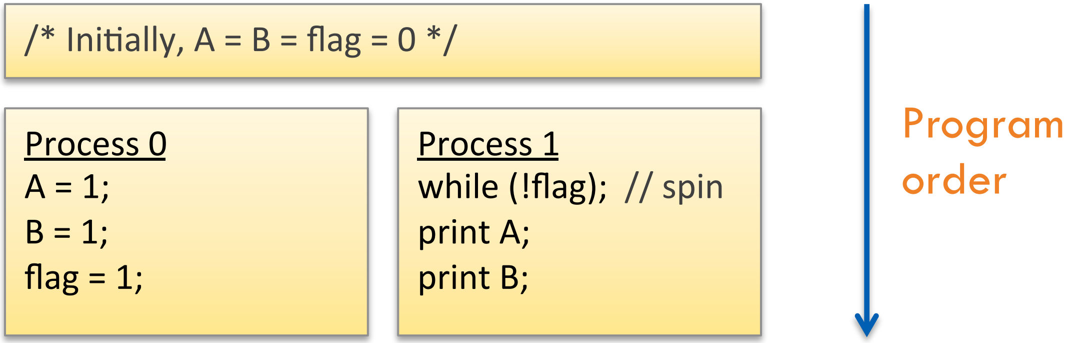
Backup

THE UNIVERSITY OF SYDNEY

Relaxed consistency

Relaxation limitations

- Example: all orders cannot be relaxed



```
/* Initially, A = B = flag = 0 */
```

```
Process 0
A = 1;
B = 1;
flag = 1;
```

```
Process 1
while (!flag);  // spin
print A;
print B;
```

- Reads on A and B can be reordered and writes on A and B can be reordered
- However, writes on A and B must be before write on flag and reads of A and B must come after the read of flag.
- Solution, 2 fences are added on Compaq Alpha, IBM PowerPC, SUN RMO

1. Between writes on A/B and write of flag and
2. Between reads of flag and reads on A/B

Continuous Consistency

Continuous consistency

Continuous consistency

- Observation: We can actually talk about a degree of consistency:
  - Replicas may differ in their numerical values

  E.g., any 2 copies of a stock market prices should not deviate by >$0.02

  - Replicas may differ in their relative staleness

  E.g., any 2 copies should not deviate by >0.5%

  - There may be differences with respect to number and order of performed update operations

  E.g., the copy of a web cache should not be k updates far apart from the server copy

- *Consistency unit (conit)*: specifies the data unit over which consistency is to be measured

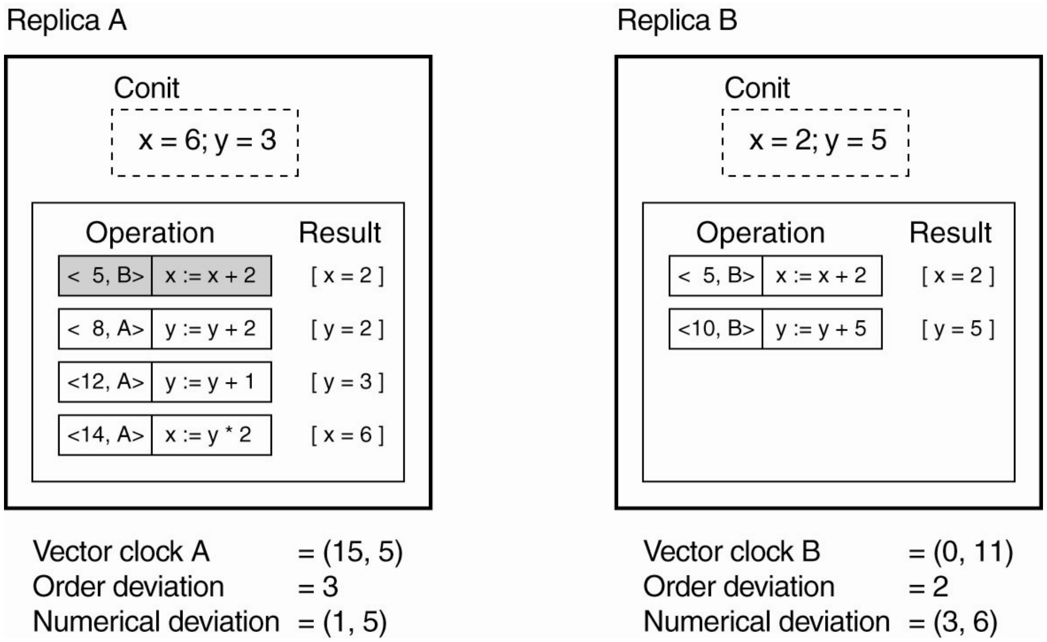
Continuous consistency

Continuous consistency

- Keeping track of consistency deviation (initially, all 0, grey means committed)
  - Initially committed value for x and y on both sites are 0
  - Conit contains the variables x and y
  - Each replica maintains a vector clock
  - B sends A operation [<5,B>: x:=x+2];
    A has made this operation permanent
    (cannot be rolled back)
  - There is only 1 operation from B that A did not see yet:
    - numerical deviation of A is 1,
    - weight is max($|y_B-y_A|$, $|x_B-x_A|$) = 5

(where y_B is the committed value of y

at B, i.e., 0 as there is no commit at B).



Replica A

Conit

x = 6; y = 3

Operation		Result
< 5, B>	x := x + 2	[x = 2]
< 8, A>	y := y + 2	[y = 2]
<12, A>	y := y + 1	[y = 3]
<14, A>	x := y * 2	[x = 6]

Vector clock A	= (15, 5)
Order deviation	= 3
Numerical deviation	= (1, 5)

Replica B

Conit

x = 2; y = 5

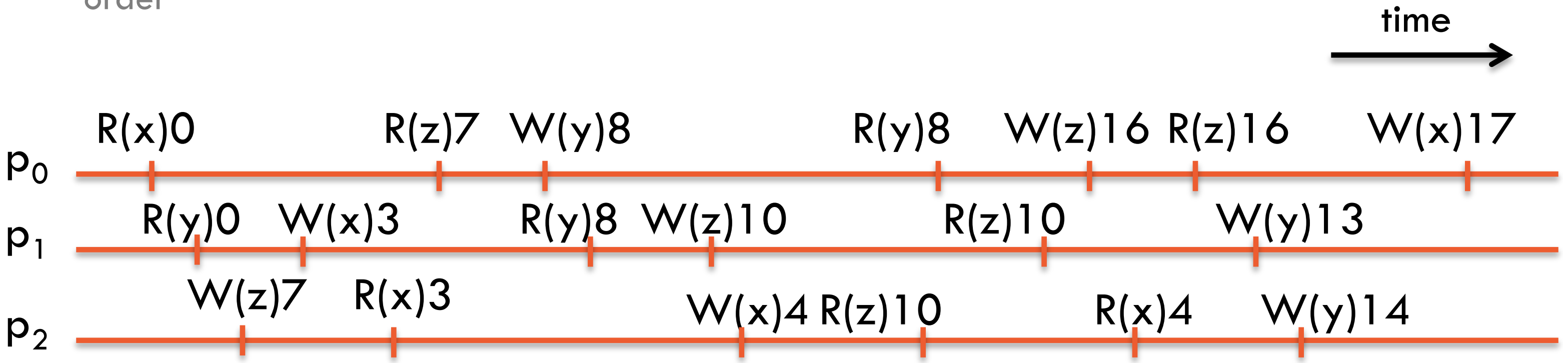
Operation		Result
< 5, B>	x := x + 2	[x = 2]
<10, B>	y := y + 5	[y = 5]

Vector clock B	= (0, 11)
Order deviation	= 2
Numerical deviation	= (3, 6)

Sequential consistency

A non-sequentially consistent execution

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



- Example 4: this is sequentially consistent
  - Is this equivalent to some serial execution? No

p0: R(x)0; R(z)7; W(y)8; R(y)8; W(z)16; R(z)16; W(x)17 cannot be before p1 or p2

p1: R(y)0; W(x)3; R(y)8; W(z)10; R(z)10; W(y)13 cannot be before p0 (x) or p2

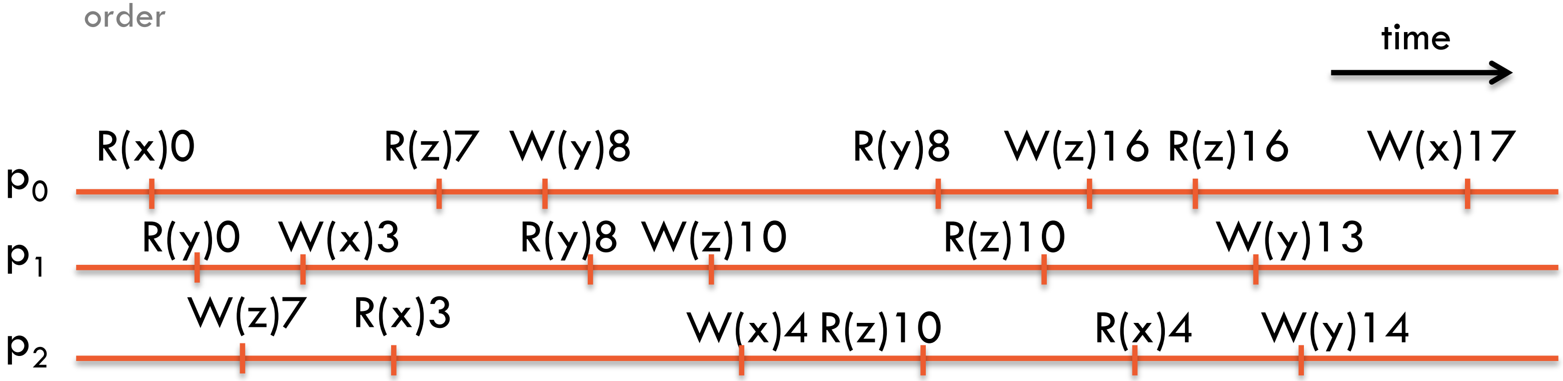
p2: W(z)7; R(x)3; W(x)4; R(z)10; R(x)4; W(y)14

p0 writes 17 on x, but p1 and p2 sees different value for x; p2 writes 14 on y but p0, p1 sees 8.

Sequential consistency

A non-sequentially consistent execution

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

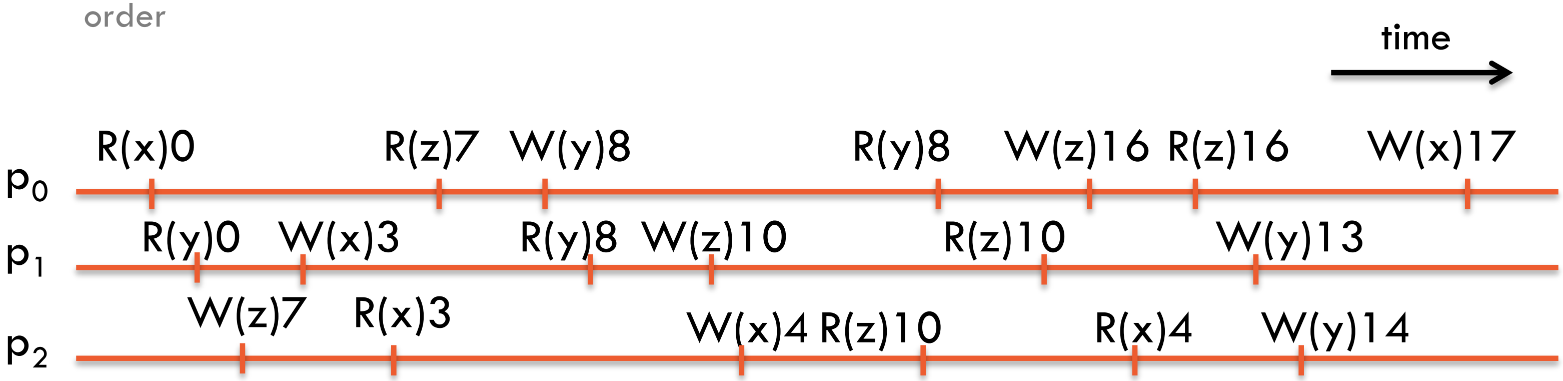


- Example 4: is the result of this execution sequentially consistent?

Sequential consistency

A non-sequentially consistent execution

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

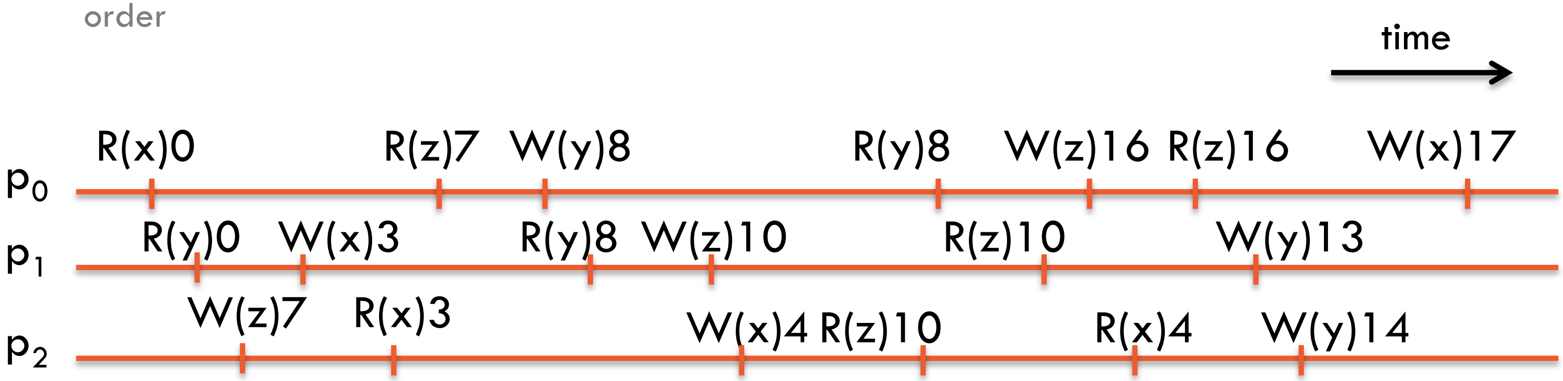


- Example 4: this is sequentially consistent
  - Is this equivalent to some serial execution?

Sequential consistency

A non-sequentially consistent execution

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



- Example 4: this is sequentially consistent
  - Is this equivalent to some serial execution? No