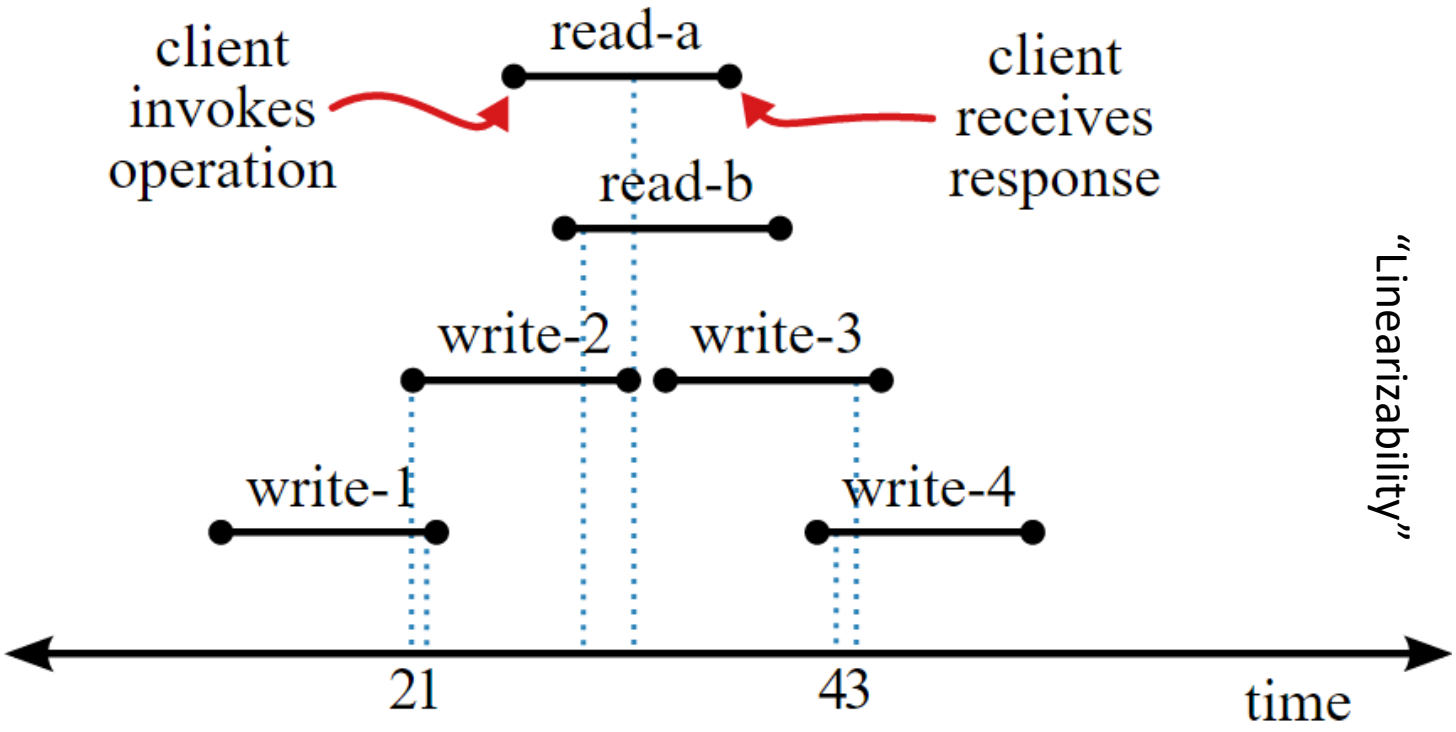
Consistency



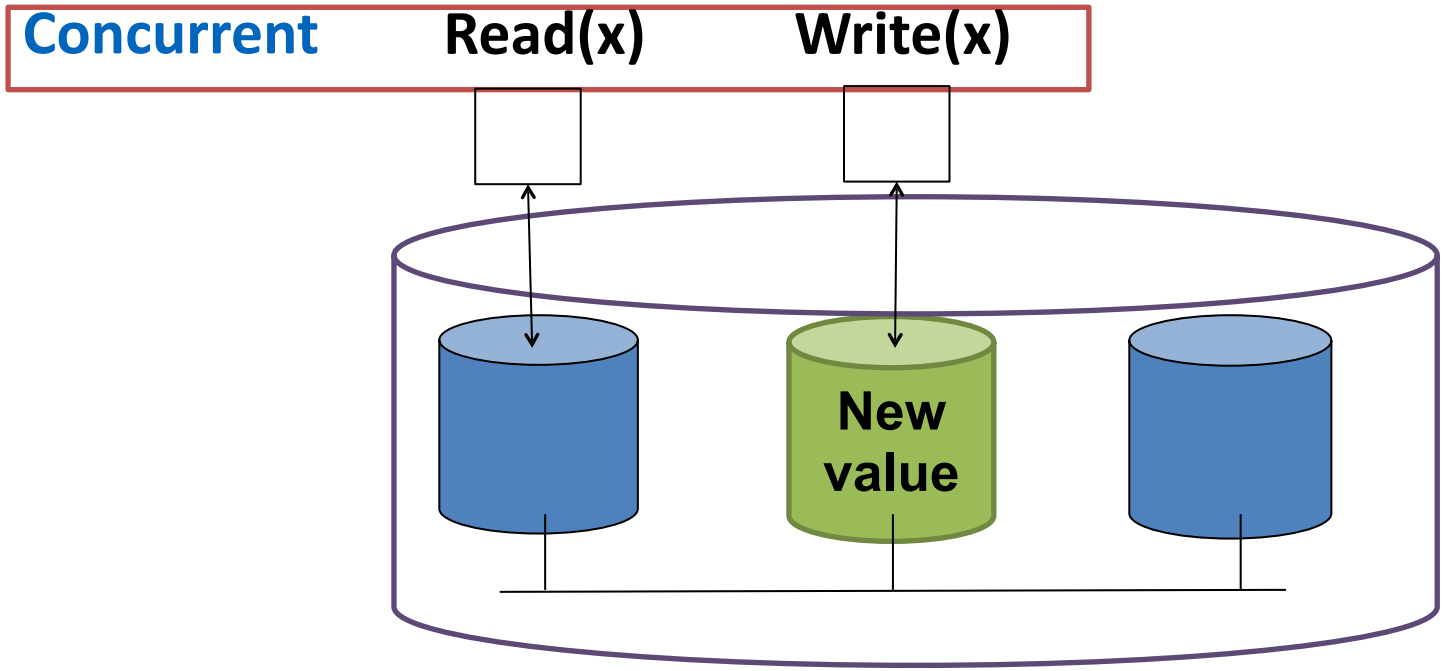
Consistency – according to dictionary.com

- steadfast adherence to the same principles, course, form, etc.

CONSISTENCY MODELS

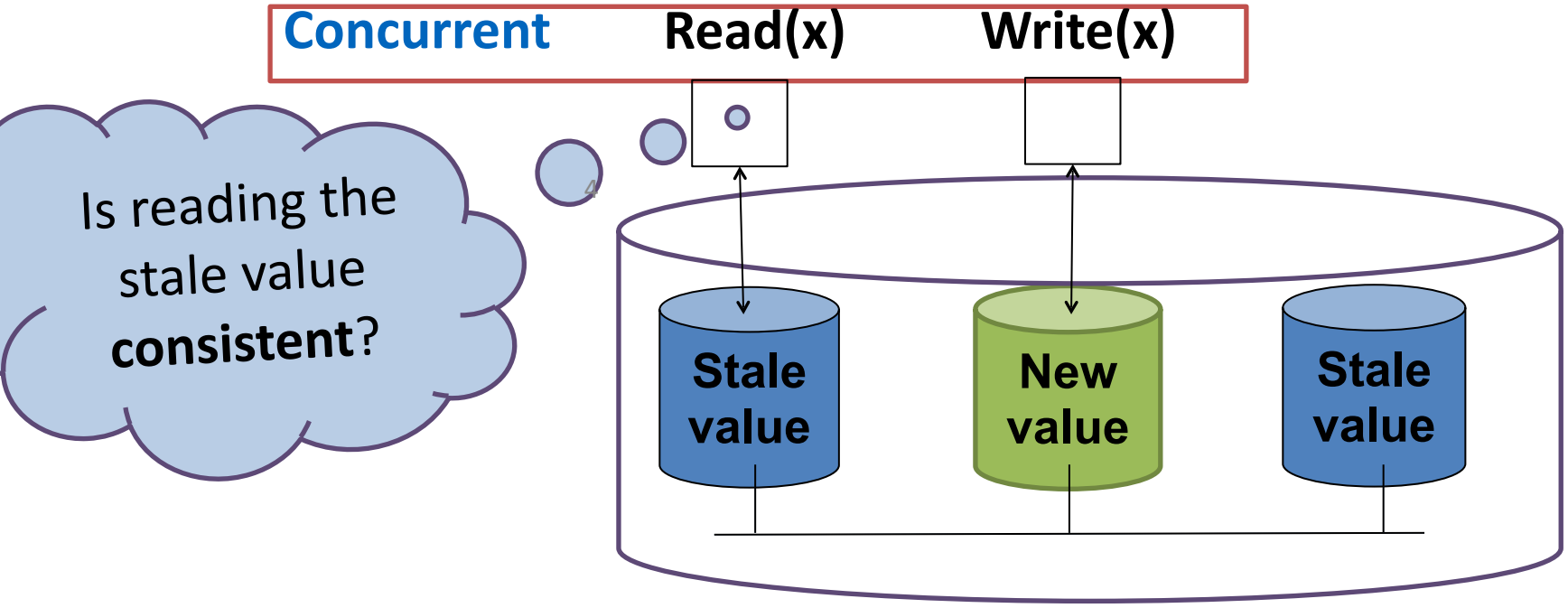
Concurrent Operations

- All operations must be applied in a **specific order** to all replicas
- **Global ordering** is **too costly** and **not scalable** (e.g., using a consensus algorithm)



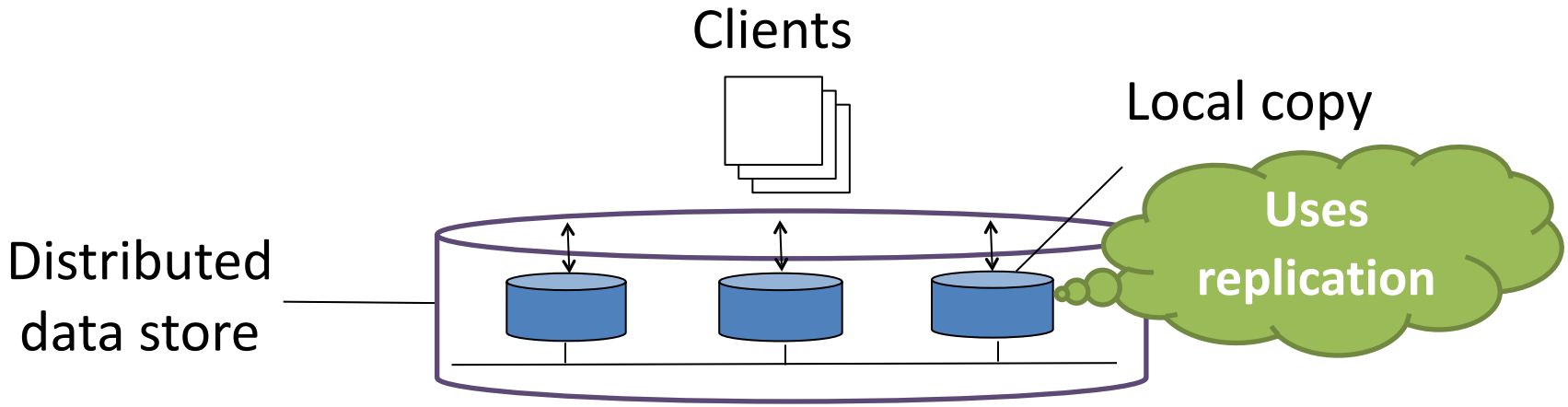
Concurrent Operations

- All operations must be applied in a **specific order** to all replicas
- **Global ordering** is **too costly** and **not scalable** (e.g., using a consensus algorithm)
- **Solution**: Avoid global ordering using **weaker** consistency requirements **suitable** for application



Consistency Models

- **Definition (data-centric consistency model)**
  - A **contract** between a distributed data store and a set of clients which specifies what **results of concurrent read/ write** operations can be



Distributed data store as synonym for replicas, distributed database, shared memory, shared files, etc.

Data-Centric Consistency Models

- **Data-centric** consistency models dictate the outcome of concurrent reads and writes:
  - Strict consistency
  - Sequential consistency
  - Linearizable consistency
  - Causal consistency
  - FIFO consistency
  - Weak consistency
  - …

Strict Consistency

- **Definition**: Any ***read*** on a data item *x* returns a value corresponding to the result of the **most recent *write*** on *x*

- Uni-processor systems have traditionally observed strict consistency, …

  – a = 1; a = 2; print(a); ***Output?***

- Definition assumes existence of **absolute global time** for unambiguous determination of "most recent“.

Strict Consistency

- **Definition**: Any ***read*** on a data item *x* returns a value corresponding to the result of the **most recent *write*** on *x*

- Uni-processor systems have traditionally observed strict consistency, …

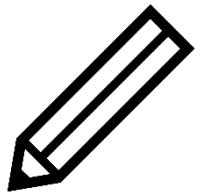
  but what about replicated systems?

  - a = 1; a = 2; print(a);

    Output?

- Definition assumes existence of **absolute global time** for unambiguous determination of "most recent“.

Sample execution I

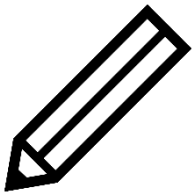


```
x = 0;
y = 0;

x = 1;
r = y;


Result:
x = 1 and r = 0
```

Sample execution II

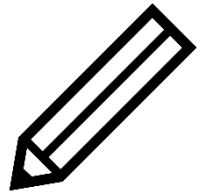


```
x = 0;
y = 0;

y = 1;
s = x;
```

Result:
y = 1 and s = 0

Sample execution III



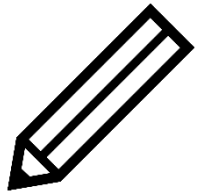
```
Shared variables
    x = 0;
    y = 0;
```

```
x = 1;
r = y;
```

```
y = 1;
s = x;
```

```
Result:
r = 0 and s = 1
r = 1 and s = 0
r = 1 and s = 1
```

Sample execution III



```
Shared variables
    x = 0;
    y = 0;
```

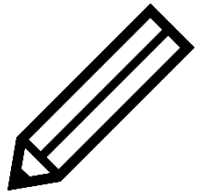
```
x = 1;
r = y;
```

```
y = 1;
s = x;
```

```
Result:
r = 0 and s = 1
r = 1 and s = 1

r = 0 and s = 0
```

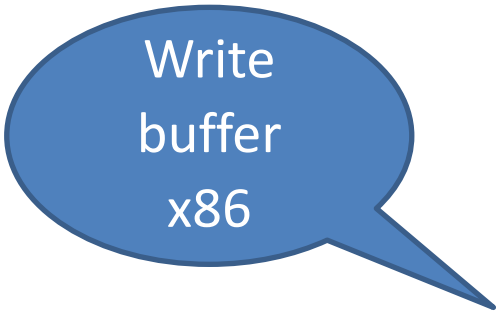
Sample execution III



```
Shared variables
    x = 0;
    y = 0;
```

```
x = 1;
r = y;
```

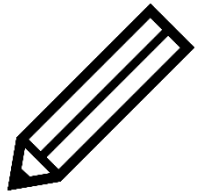
```
y = 1;
s = x;
```




```
Result:
r = 0 and s = 1
r = 1 and s = 0
r = 1 and s = 1

r = 0 and s = 0
```

Sample execution III



```
Shared variables
    x = 0;
    y = 0;
```

```
x = 1;
r = y;
```

```
y = 1;
s = x;
```

```
Result:
r = 0 and s = 1
r = 1 and s = 1
```

Write buffer x86

Compiler optimization / reordering

```
r = 0 and s = 0
```

Interpretation of Strict Consistency

- Under strict consistent, all writes are **instantaneously visible to all nodes** and **absolute global time order is maintained**

- If a **replica is updated**, …
  - all subsequent reads, **see the new value**, no matter how soon after the update the reads are done
  - and no matter which node is doing the reading and where it is located

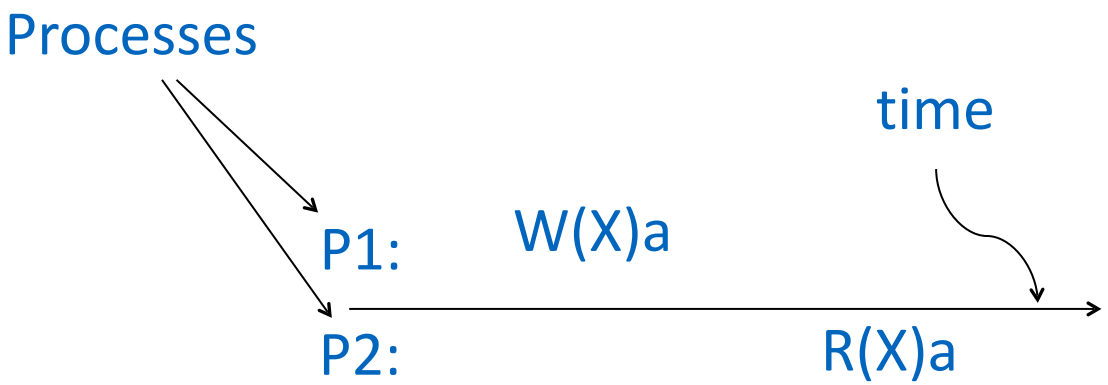
- Similarly, if a **read** is done, then it **gets the most recent value,** no matter how quickly the next write is done

Notation

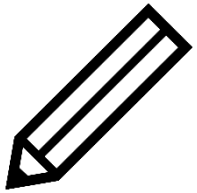
W(X)a: Represents **writing** value ***a*** to data ***X*** *(memory)*

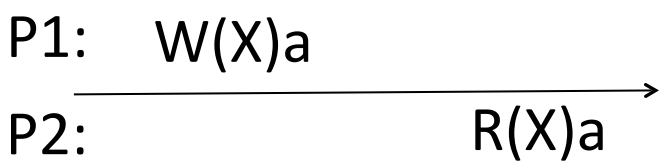
R(X)a: Represents **reading** data ***X***, which returns value ***a***

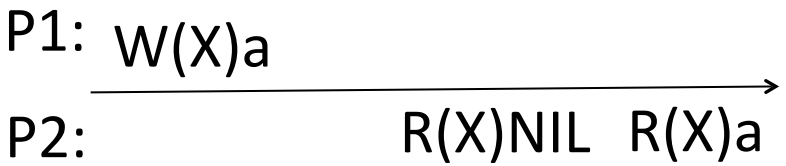
Initial value of ***X*** is ***NIL***



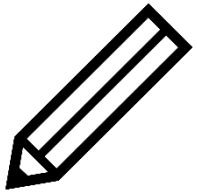
Strict Consistency Example

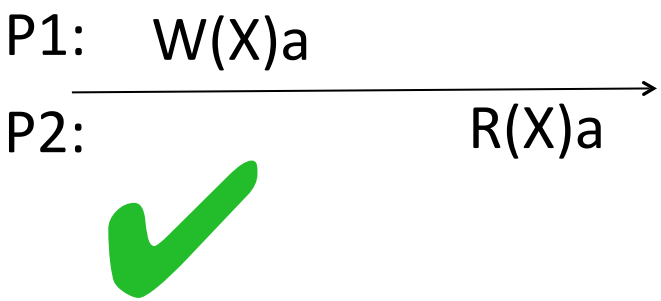


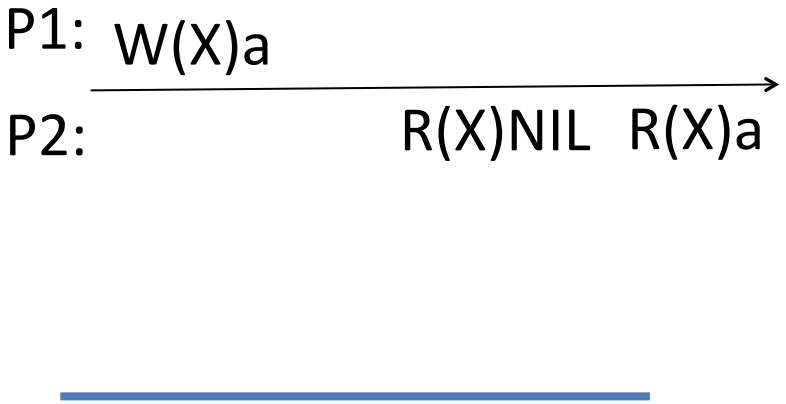




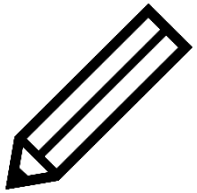
Strict Consistency Example

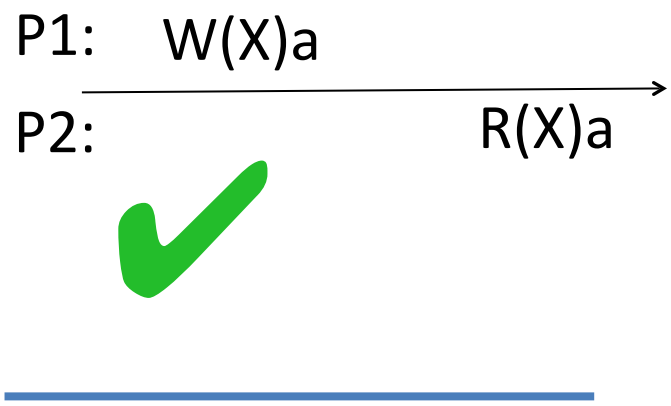


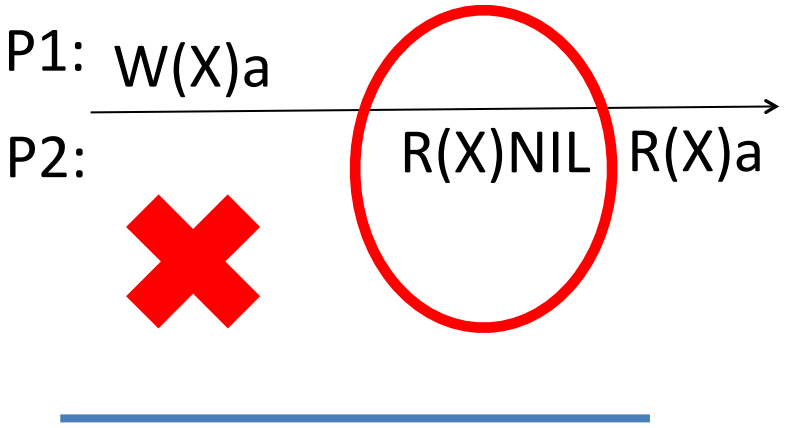




Strict Consistency Example

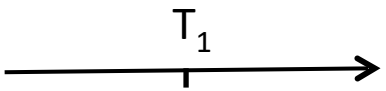


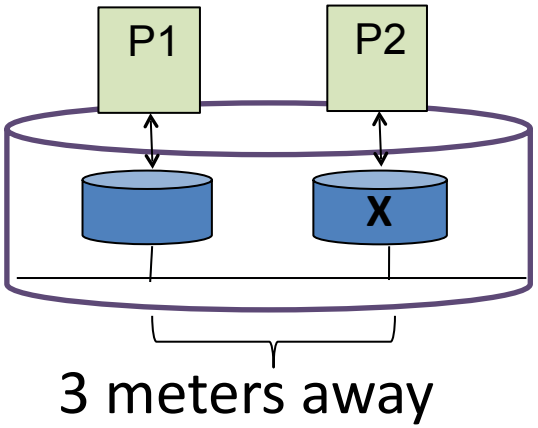




Strict Consistency: Thought Experiment

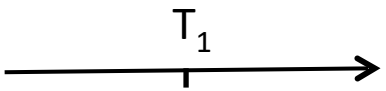
@P1: W(X)a at T_1 **@P2: R(X)a** at T_1 + 1ns



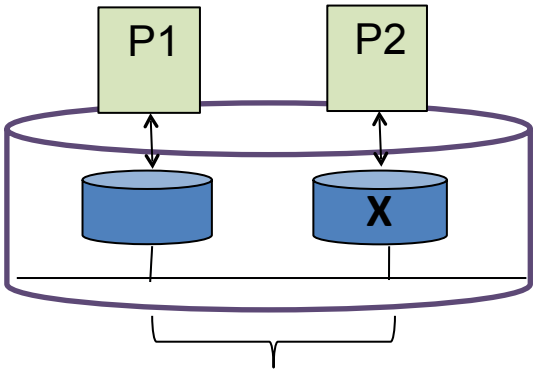


Strict Consistency: Thought Experiment

- To satisfy strict consistency, laws of physics may have to be violated! Obviously, not an option!!!
- Example:



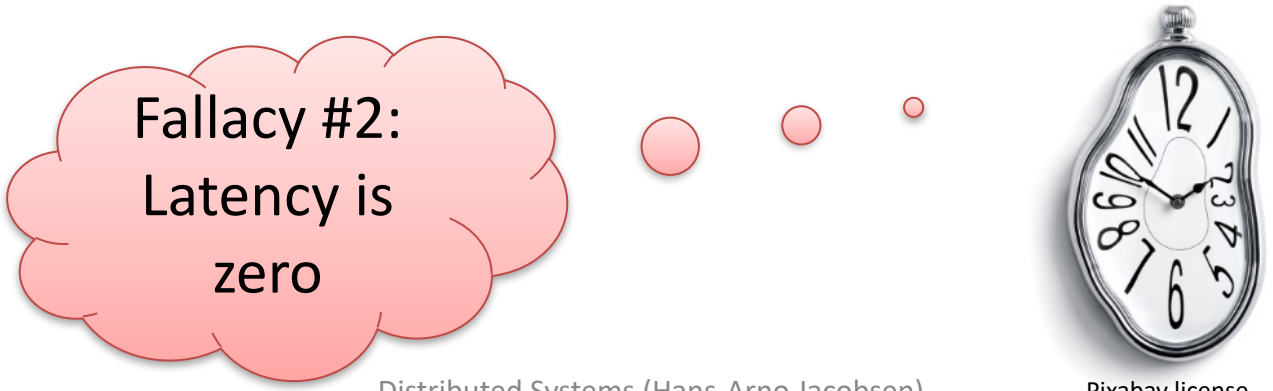
@P1: W(X)a at T_1 **@P2: R(X)a** at T_1 + 1ns



To realize strict consistency in this case, *W(X)a* would have to travel at 10 times the speed of light!

Strict Consistency: The Bad News

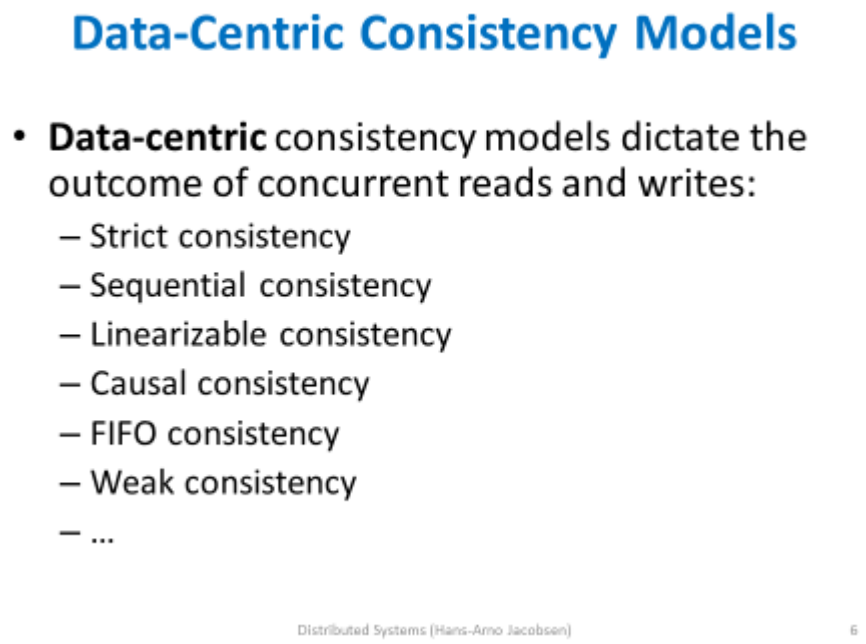
- It is **impossible** to instantaneously replicate operations
- It is **impossible to perfectly** synchronize clocks
  - How to accurately determine the time of each operation?



So, *what are we to do?*

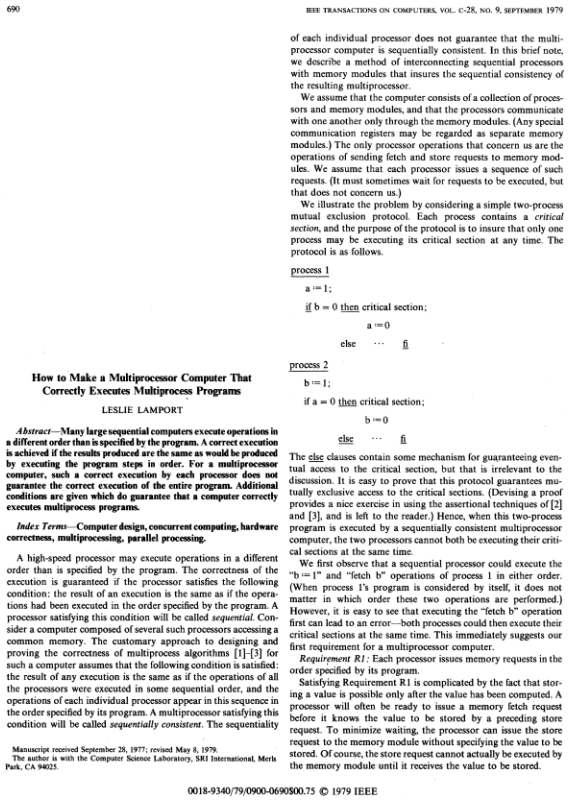
- Relax consistency and introduce models with weaker consistency requirements

Pixabay license



Self-study questions

- Verify the timing assumptions in the thought experiment on strict consistency?
- What aspect of a consistency model could be relaxed and why?
- Create some concurrent read and write sequences that conform to strict consistency.
- Create some concurrent operation sequences that violate strict consistency.
- See if the projected outcome of `r=0, s=0` can occur on an x86 or due to compiler optimizations.
- Would a quantum tunnel help us achieve stric consistency in a replicated system? ☺



690 IEEE TRANSACTIONS ON COMPUTERS, VOL. C-28, NO. 9, SEPTEMBER 1979

How to Make a Multiprocessor Computer That Correctly Executes Multiprocess Programs

LESLIE LAMPORT

Abstract—**Many large sequential computers execute operations in a different order than is specified by the program. A correct execution is achieved if the results produced are the same as would be produced by executing the program steps in order. For a multiprocessor computer, such a correct execution by each processor does not guarantee the correct execution of the entire program. Additional conditions are given which do guarantee that a computer correctly executes multiprocess programs.**

Index Terms—**Computer design, concurrent computing, hardware correctness, multiprocessing, parallel processing.**

A high-speed processor may execute operations in a different order than is specified by the program. The correctness of the execution is guaranteed if the processor satisfies the following condition: the result of an execution is the same as if the operations had been executed in the order specified by the program. A processor satisfying this condition will be called *sequential*. Consider a computer composed of several such processors accessing a common memory. The customary approach to designing and proving the correctness of multiprocess algorithms [1]-[3] for such a computer assumes that the following condition is satisfied: the result of any execution is the same as if the operations of all the processors were executed in some sequential order, and the operations of each individual processor appear in this sequence in the order specified by its program. A multiprocessor satisfying this condition will be called *sequentially consistent*. The sequentiality of each individual processor does not guarantee that the multiprocessor computer is sequentially consistent. In this brief note, we describe a method of interconnecting sequential processors with memory modules that insures the sequential consistency of the resulting multiprocessor.

Manuscript received September 28, 1977; revised May 8, 1979.
The author is with the Computer Science Laboratory, SRI International, Menlo Park, CA 94025.

We assume that the computer consists of a collection of processors and memory modules, and that the processors communicate with one another only through the memory modules. (Any special communication registers may be regarded as separate memory modules.) The only processor operations that concern us are the operations of sending fetch and store requests to memory modules. We assume that each processor issues a sequence of such requests. (It must sometimes wait for requests to be executed, but that does not concern us.)

We illustrate the problem by considering a simple two-process mutual exclusion protocol. Each process contains a *critical section*, and the purpose of the protocol is to insure that only one process may be executing its critical section at any time. The protocol is as follows.

```
process 1
    a := 1;
    if b = 0 then critical section;
                  a := 0
             else ···  fi

process 2
    b := 1;
    if a = 0 then critical section;
                  b := 0
             else ···  fi
```

The else clauses contain some mechanism for guaranteeing eventual access to the critical section, but that is irrelevant to the discussion. It is easy to prove that this protocol guarantees mutually exclusive access to the critical sections. (Devising a proof provides a nice exercise in using the assertional techniques of [2] and [3], and is left to the reader.) Hence, when this two-process program is executed by a sequentially consistent multiprocessor computer, the two processors cannot both be executing their critical sections at the same time.

We first observe that a sequential processor could execute the "b := 1" and "fetch b" operations of process 1 in either order. (When process 1's program is considered by itself, it does not matter in which order these two operations are performed.) However, it is easy to see that executing the "fetch b" operation first can lead to an error—both processes could then execute their critical sections at the same time. This immediately suggests our first requirement for a multiprocessor computer.

Requirement R1: Each processor issues memory requests in the order specified by its program.

Satisfying Requirement R1 is complicated by the fact that storing a value is possible only after the value has been computed. A processor will often be ready to issue a memory fetch request before it knows the value to be stored by a preceding store request. To minimize waiting, the processor can issue the store request to the memory module without specifying the value to be stored. Of course, the store request cannot actually be executed by the memory module until it receives the value to be stored.

SEQUENTIAL CONSISTENCY

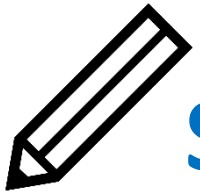
FORMALIZED IN 1979

Definition of Sequential Consistency

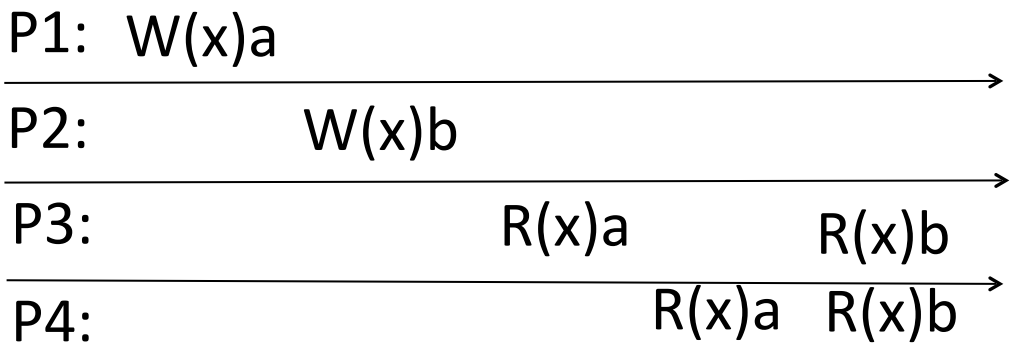
- The result of any execution is the same as if the operations by **all processes** on the data store were executed in some **sequential order** and …

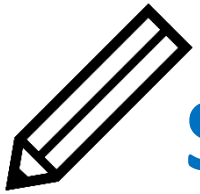
- … the operations of each individual process appear in this **sequence in the order specified by its program**

- Weaker than strict consistency: **logical** time instead of **physical** time

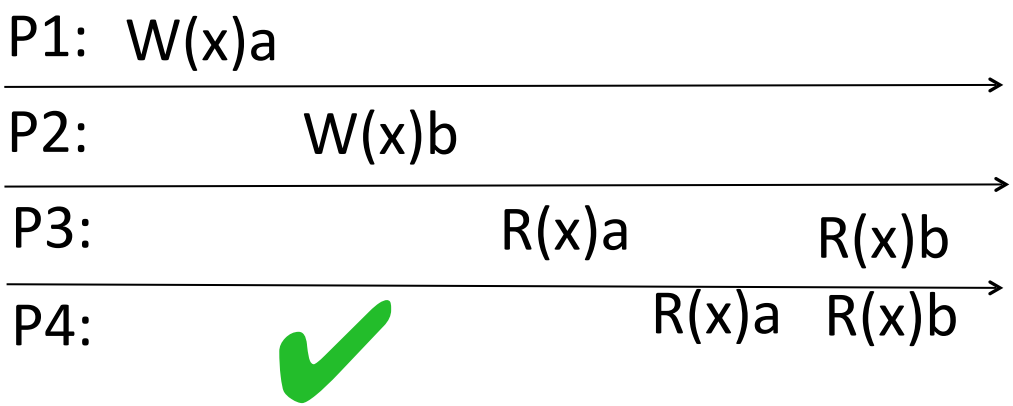


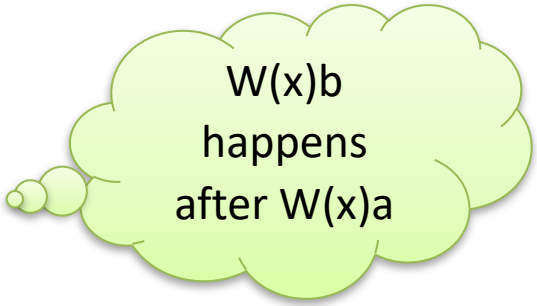
Sequential Consistency Example I



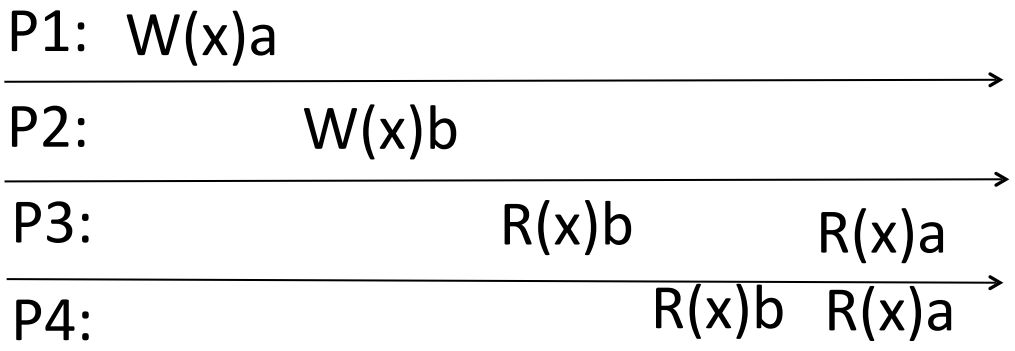


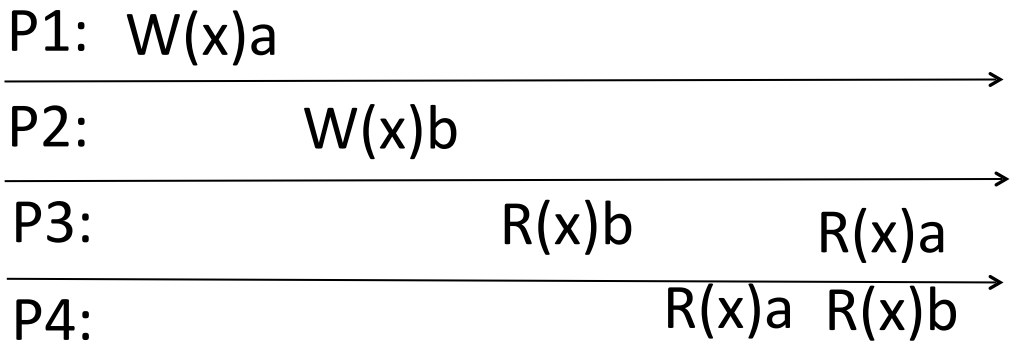
Sequential Consistency Example I



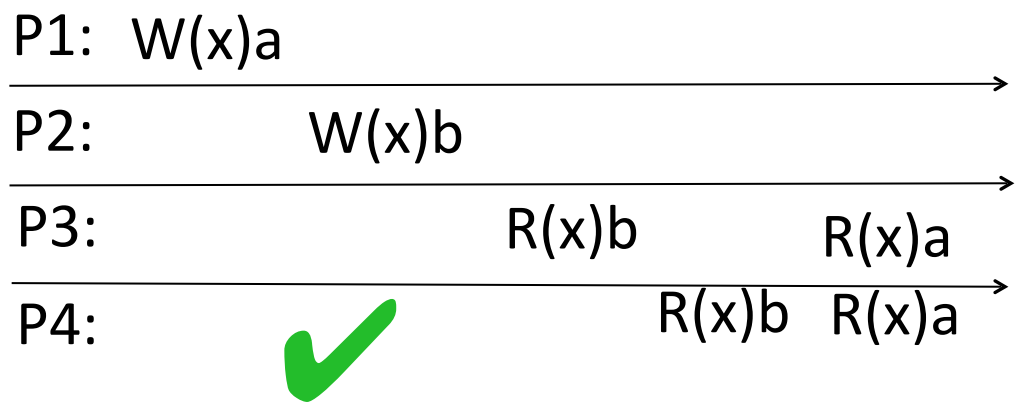


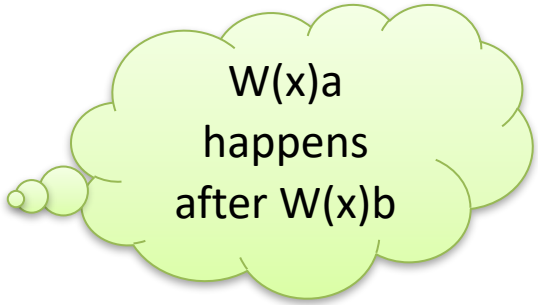
Sequential Consistency Example II

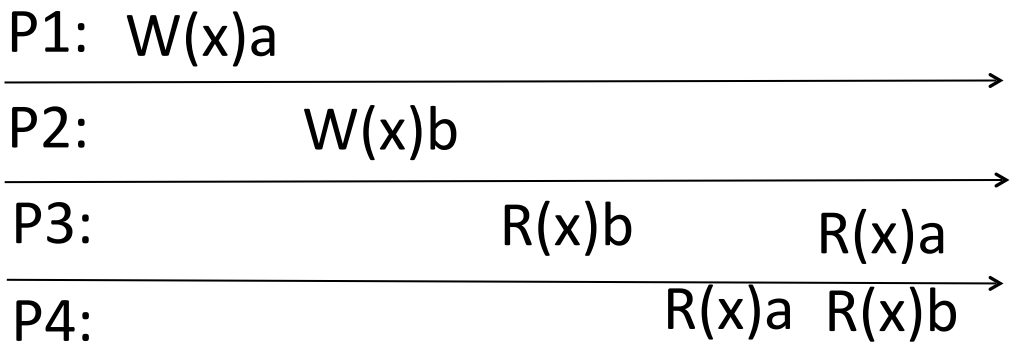




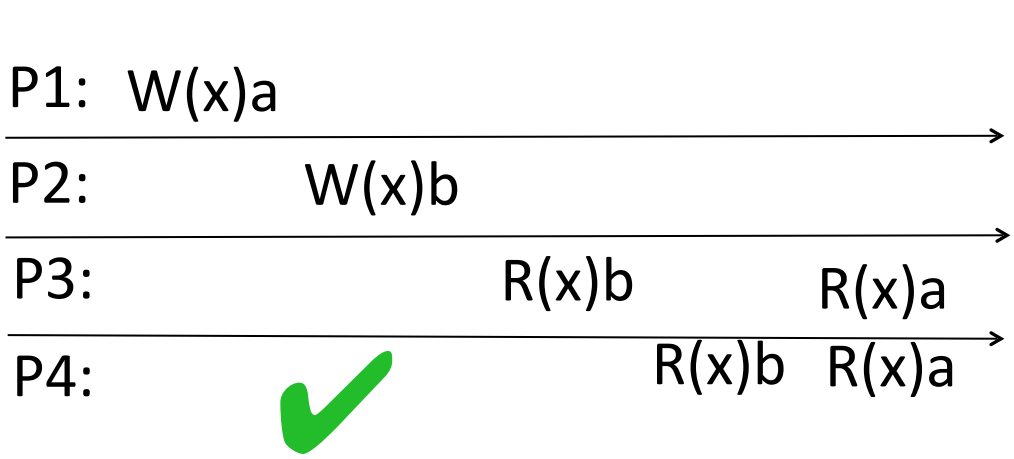
Sequential Consistency Example II

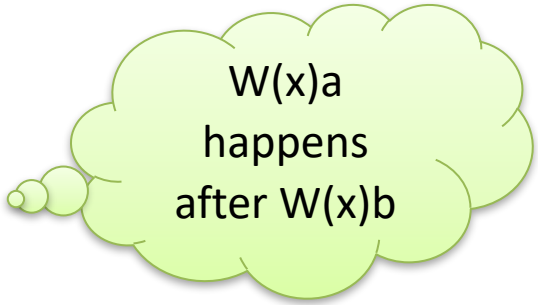


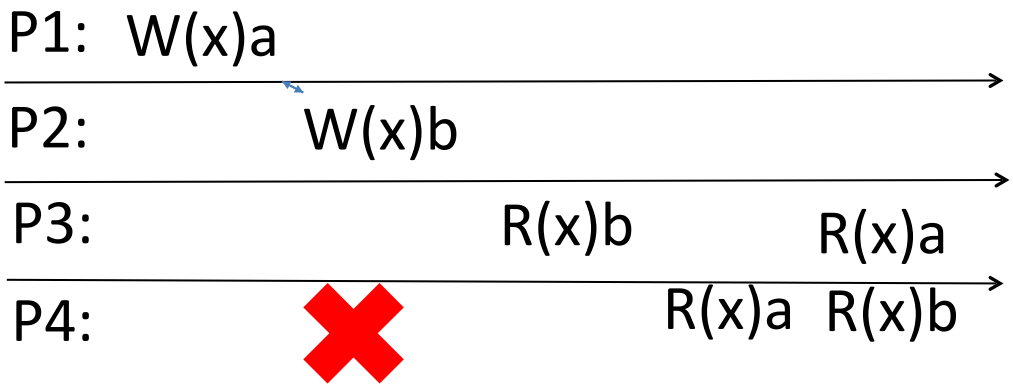




Sequential Consistency Example II





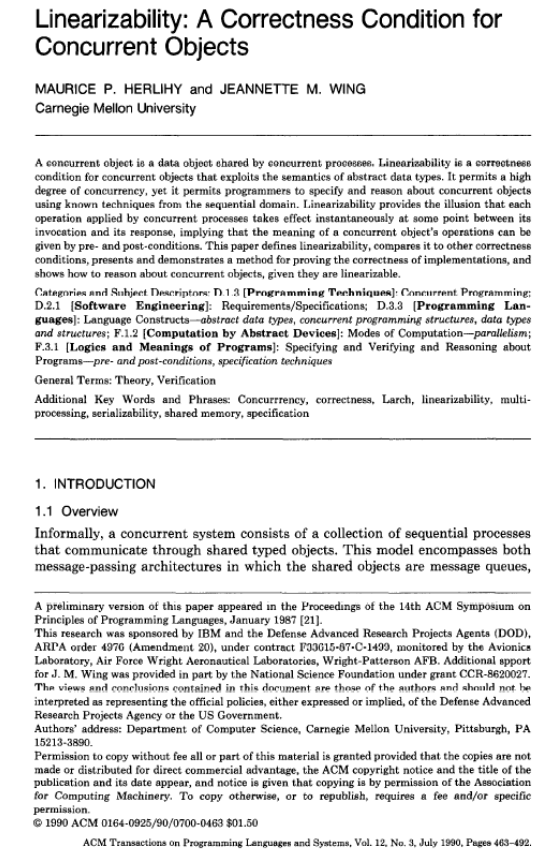


Self-study questions

- Create some concurrent read and write sequences that conform to sequential consistency.
- Create some concurrent operation sequences that violate sequential consistency.
- Is a strictly consistent system, sequentially consistent and vice versa, discuss?
- How could sequential consistency be implemented?

LINEARIZABILITY

FORMALIZED IN 1990



Linearizability: A Correctness Condition for Concurrent Objects

MAURICE P. HERLIHY and JEANNETTE M. WING
Carnegie Mellon University

A concurrent object is a data object shared by concurrent processes. Linearizability is a correctness condition for concurrent objects that exploits the semantics of abstract data types. It permits a high degree of concurrency, yet it permits programmers to specify and reason about concurrent objects using known techniques from the sequential domain. Linearizability provides the illusion that each operation applied by concurrent processes takes effect instantaneously at some point between its invocation and its response, implying that the meaning of a concurrent object's operations can be given by pre- and post-conditions. This paper defines linearizability, compares it to other correctness conditions, presents and demonstrates a method for proving the correctness of implementations, and shows how to reason about concurrent objects, given they are linearizable.

Categories and Subject Descriptors: D.1.3 [**Programming Techniques**]: Concurrent Programming; D.2.1 [**Software Engineering**]: Requirements/Specifications; D.3.3 [**Programming Languages**]: Language Constructs—*abstract data types, concurrent programming structures, data types and structures*; F.1.2 [**Computation by Abstract Devices**]: Modes of Computation—*parallelism*; F.3.1 [**Logics and Meanings of Programs**]: Specifying and Verifying and Reasoning about Programs—*pre- and post-conditions, specification techniques*

General Terms: Theory, Verification

Additional Key Words and Phrases: Concurrrency, correctness, Larch, linearizability, multiprocessing, serializability, shared memory, specification

1. INTRODUCTION

1.1 Overview

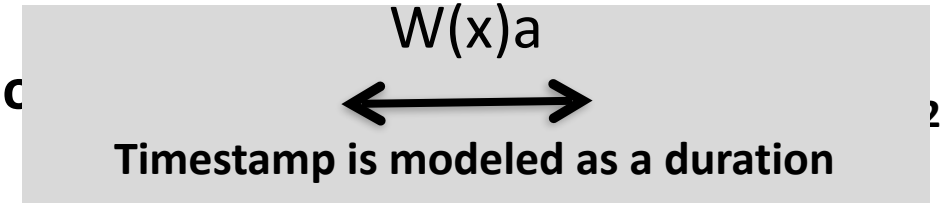
Informally, a concurrent system consists of a collection of sequential processes that communicate through shared typed objects. This model encompasses both message-passing architectures in which the shared objects are message queues,

A preliminary version of this paper appeared in the Proceedings of the 14th ACM Symposium on Principles of Programming Languages, January 1987 [21].
This research was sponsored by IBM and the Defense Advanced Research Projects Agents (DOD), ARPA order 4976 (Amendment 20), under contract F33615-87-C-1499, monitored by the Avionics Laboratory, Air Force Wright Aeronautical Laboratories, Wright-Patterson AFB. Additional spport for J. M. Wing was provided in part by the National Science Foundation under grant CCR-8620027. The views and conclusions contained in this document are those of the authors and should not be interpreted as representing the official policies, either expressed or implied, of the Defense Advanced Research Projects Agency or the US Government.
Authors' address: Department of Computer Science, Carnegie Mellon University, Pittsburgh, PA 15213-3890.

ACM Transactions on Programming Languages and Systems, Vol. 12, No. 3, July 1990, Pages 463–492.

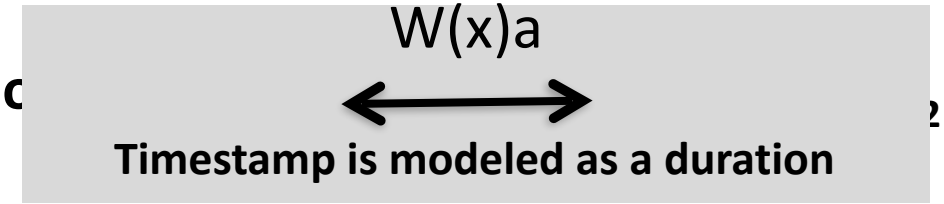
Definition of Linearizability

- 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.
- Here, $\mathbf{ts_{OP}(x)}$ denotes **timestamp** assigned to operation OP that is performed on data item *x,* and OP is either read (R) or write (W)
- If $\mathbf{ts_{OP_1}(x) < ts_{OP_2}(y)}$, then **c** **$_2$(y)** in this sequence



Definition of Linearizability

- 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 sequen** **Sequential consistency** individual process appe ified by its program.
- Here, $\mathbf{ts_{OP}(x)}$ denotes **timestamp** assigned to operation OP that is performed on data item *x,* and OP is either read (R) or write (W)
- If $\mathbf{ts_{OP_1}(x) < ts_{OP_2}(y)}$, then **c** $\mathbf{_2(y)}$ in this sequence



Linearizability

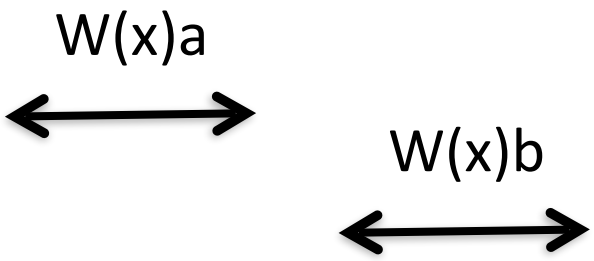
- Linearizability is weaker than strict consistency, but stronger than sequential consistency.

- Like strict consistency, **assumes global time,** but not absolute global time

- Assumes processes in the system have **physical clocks** synchronized to within an **bounded error** (captured by timestamp's duration)

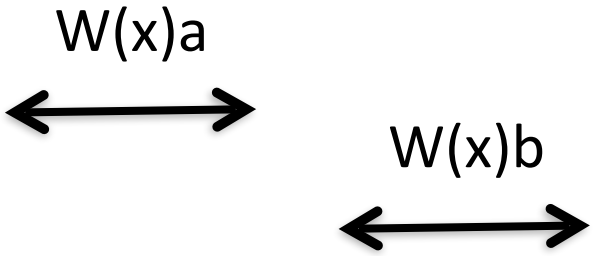
Linearizability

- If ***W(x)b*** was the **most recent** write operation and there is no other write operation **overlapping** with *W(x)b*, then any **later** read should return ***b***

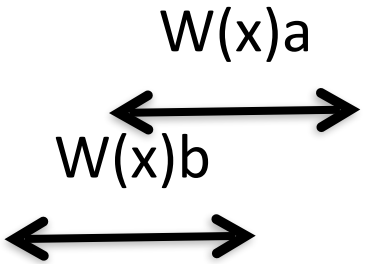


Linearizability

- If ***W(x)b*** was the **most recent** write operation and there is no other write operation **overlapping** with *W(x)b*, then any **later** read should return ***b***



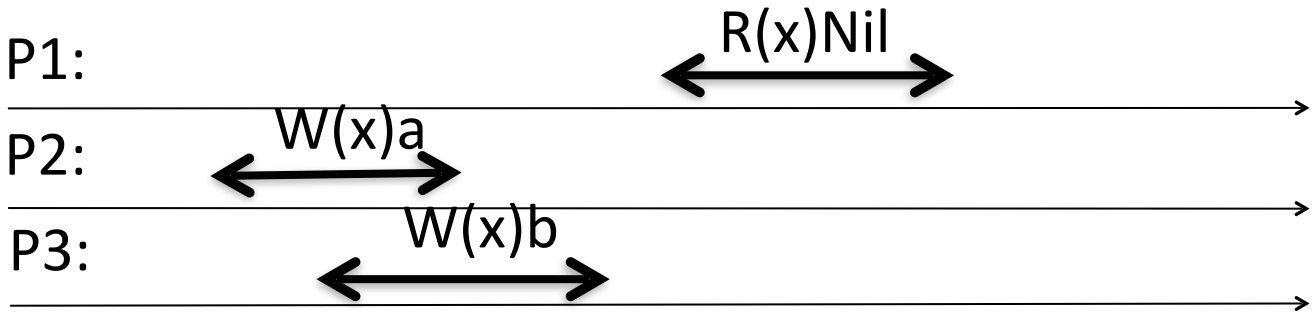
- If *W(x)a* and *W(x)b* were two **most-recent overlapping** write operation, then **any later read** should **return either *a* or *b*,** not something else

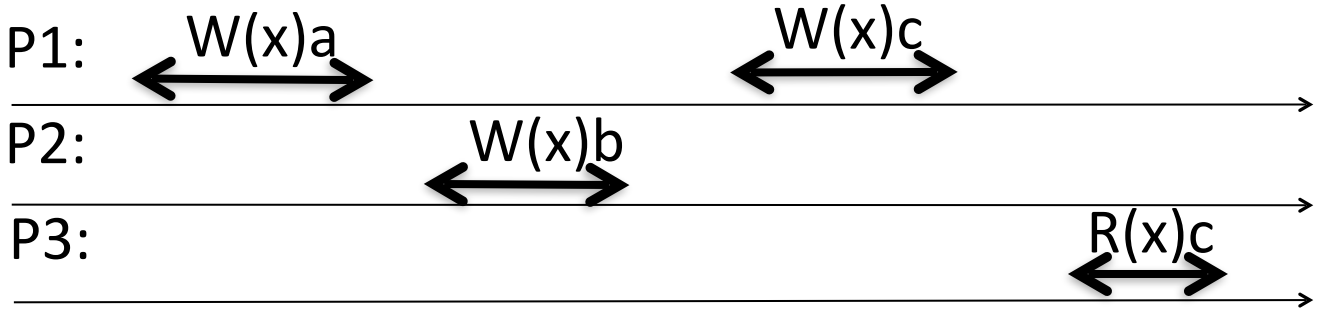


Linearizability

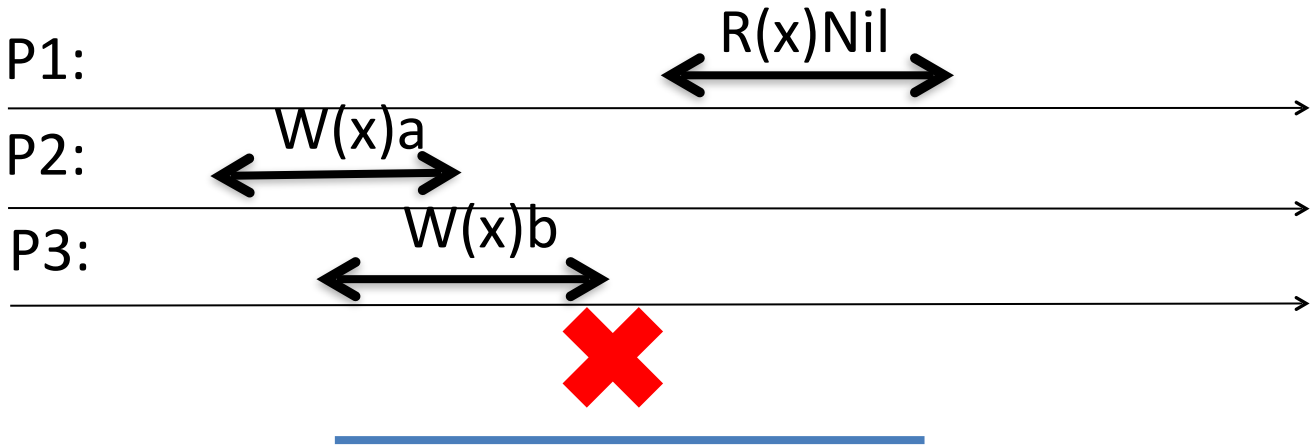
- A linearizable data store is also **sequentially consistent**
- I.e., linearizability is more restrictive (**stronger**)
- Difference is ordering according to a set of synchronized clocks
- Linearizability prevents **stale reads**, since it guarantees correct reads after a write is completed

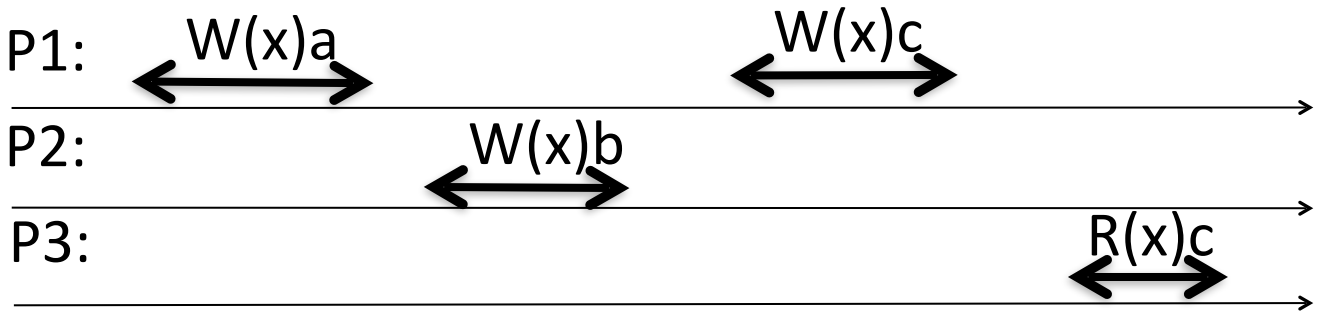
Example 1



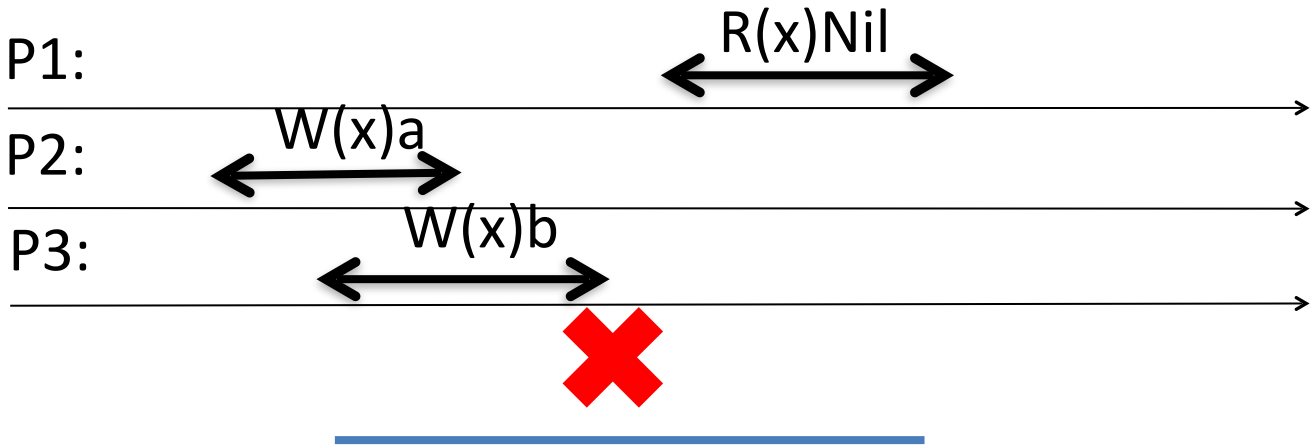


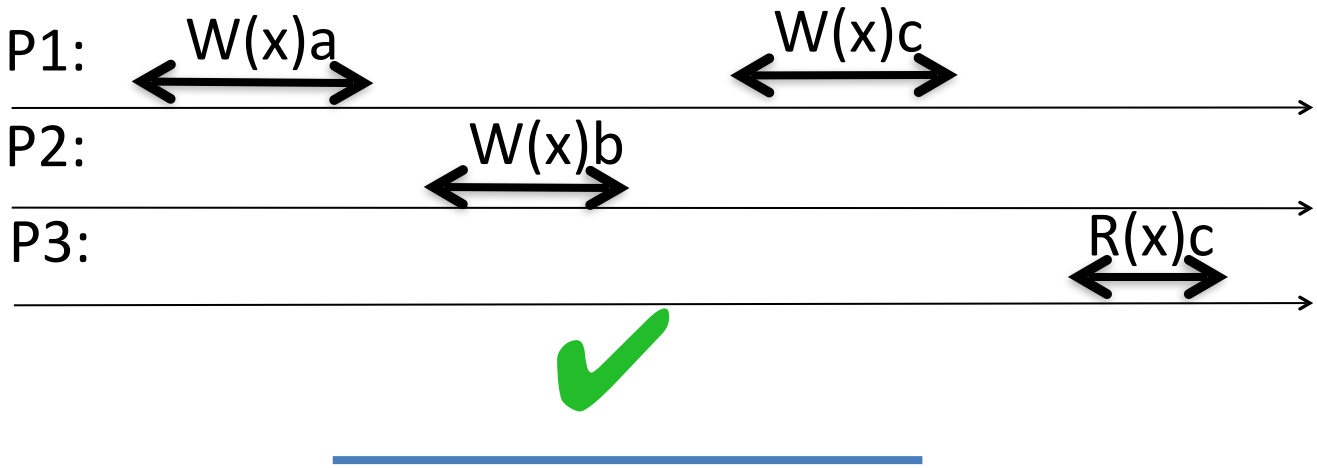
Example 1



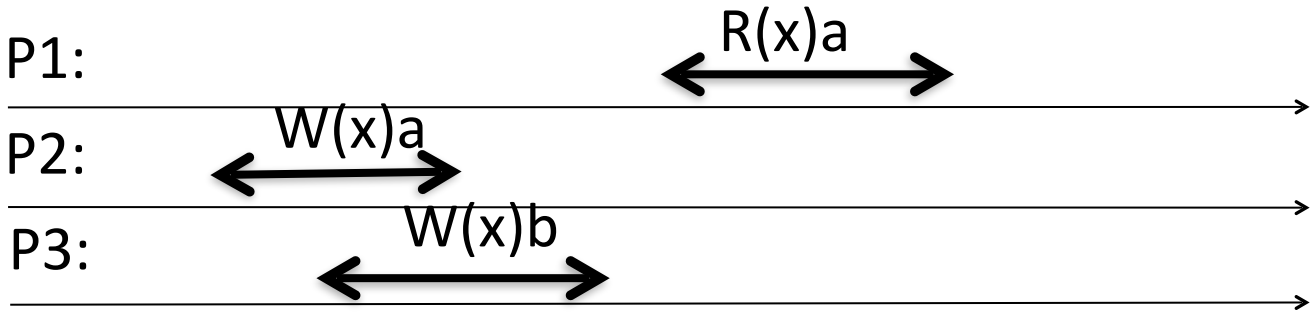


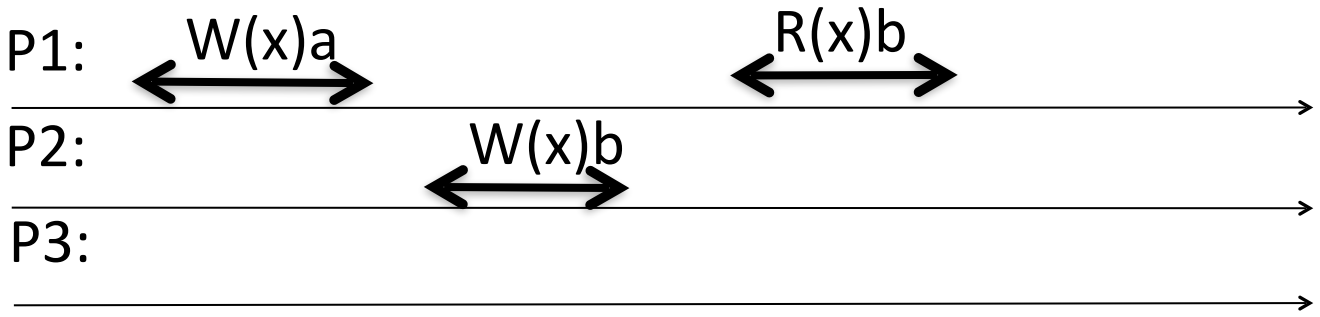
Example 1



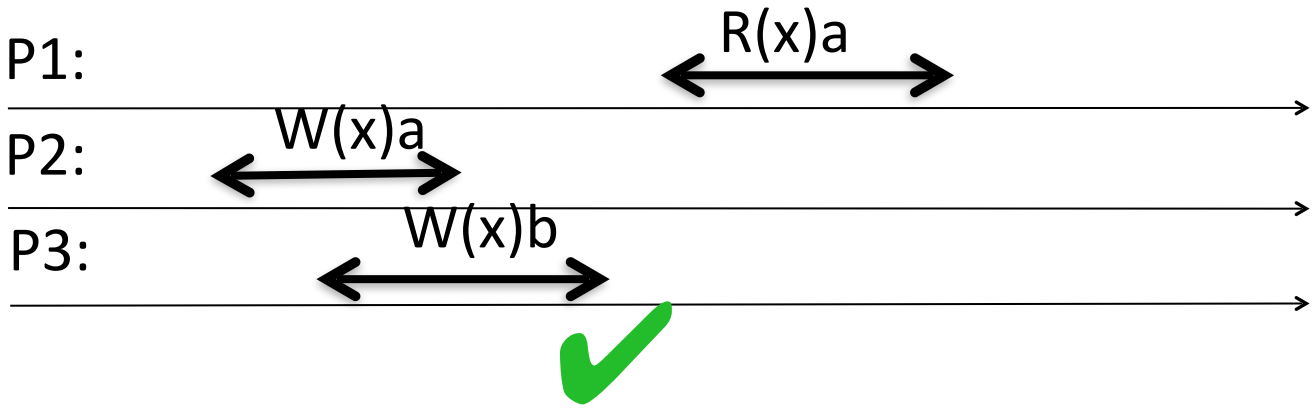


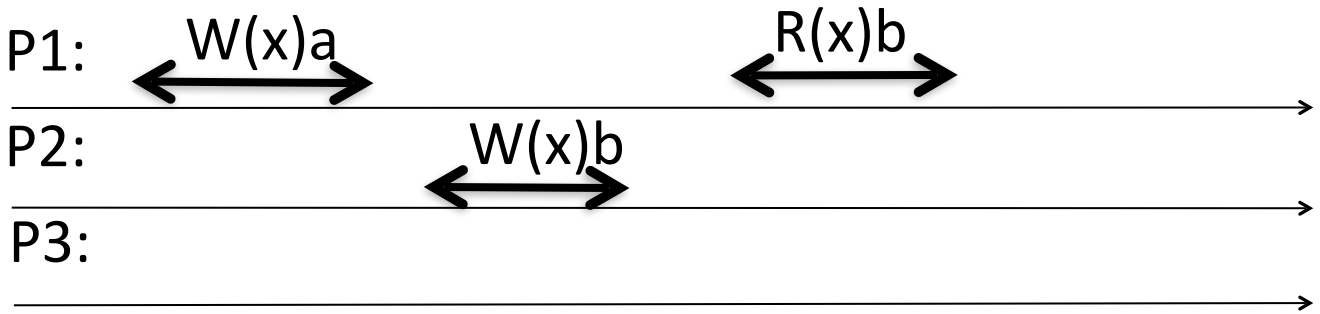
Example 2



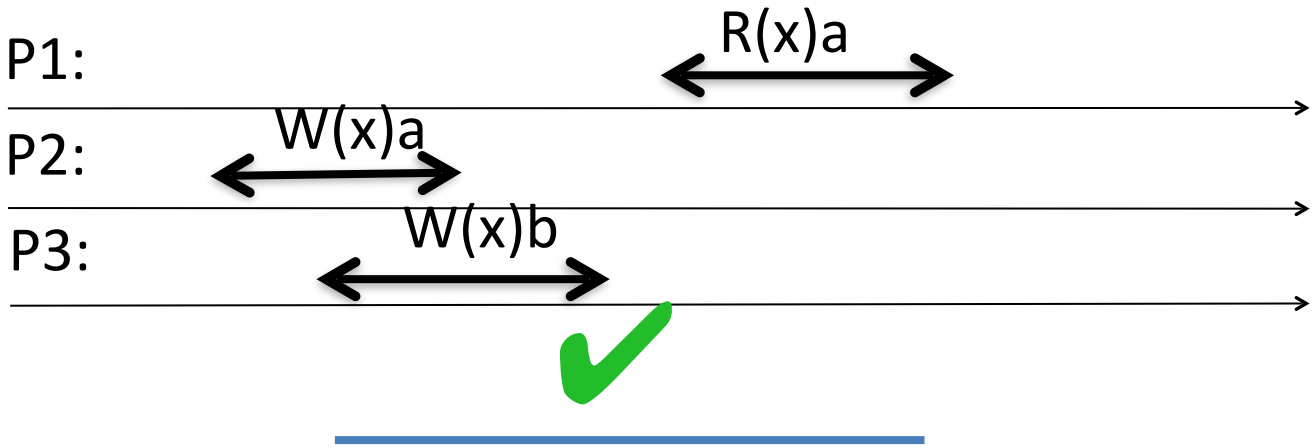


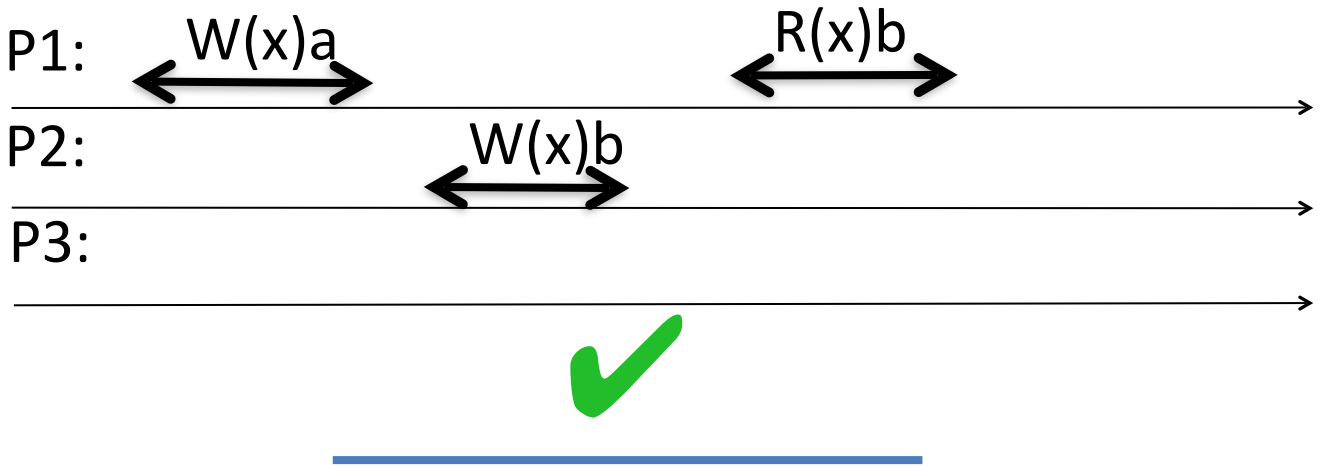
Example 2



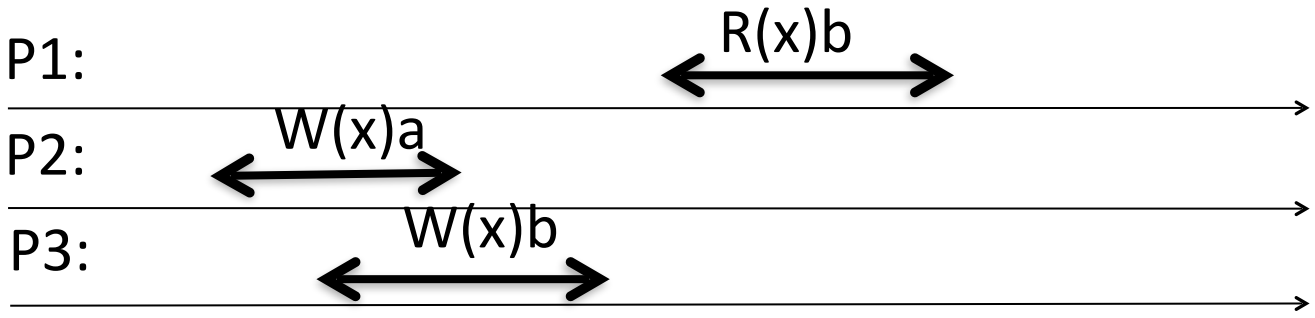


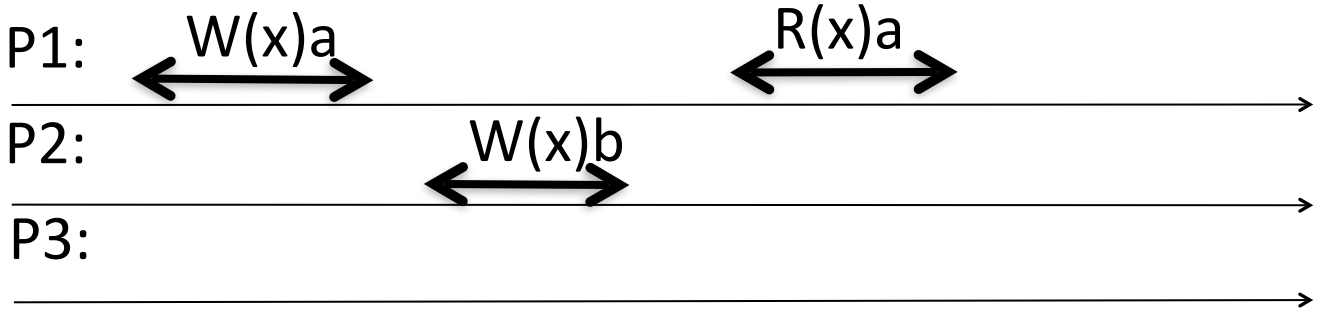
Example 2



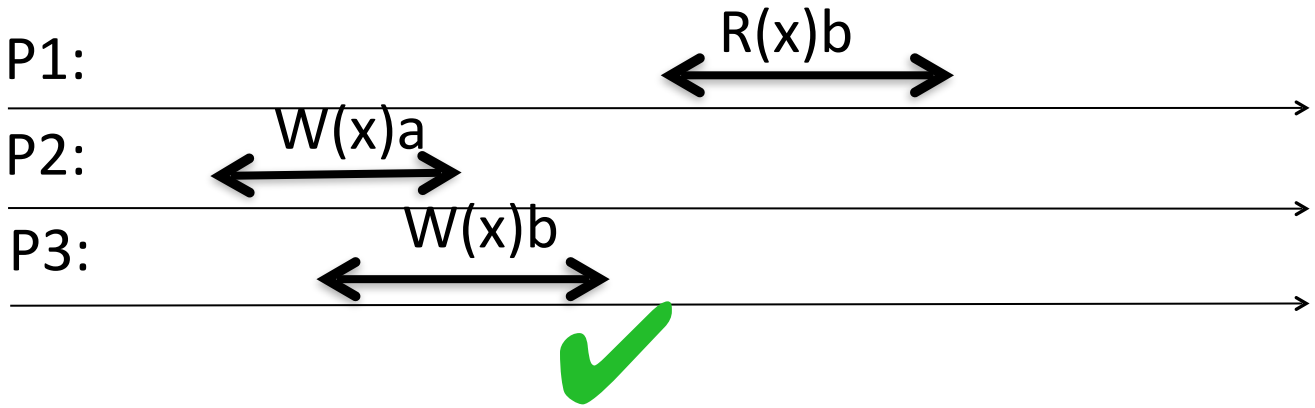


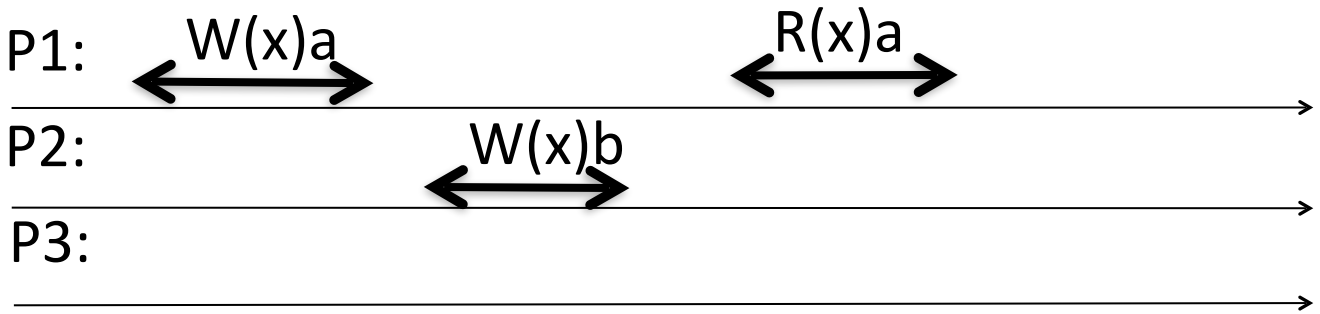
Example 3



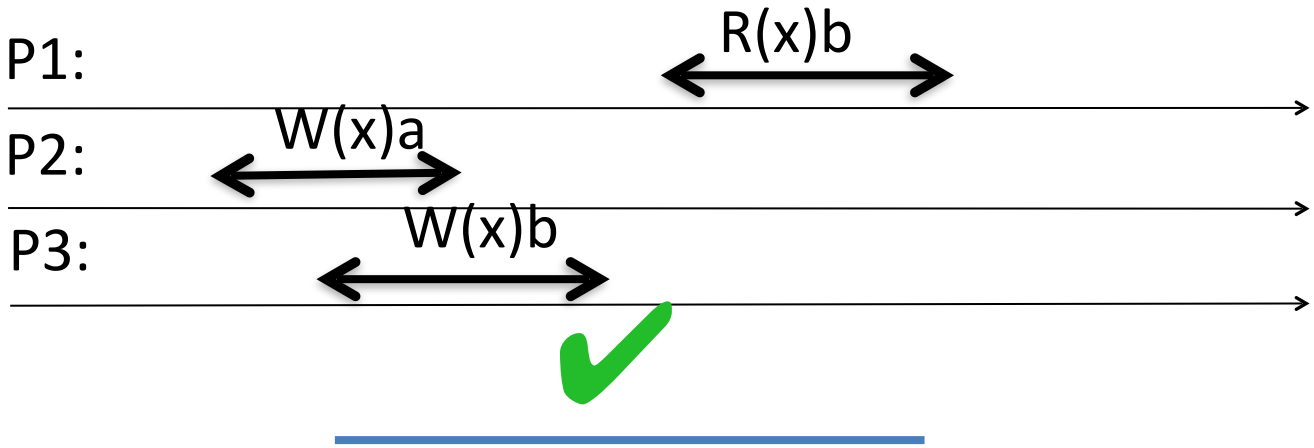


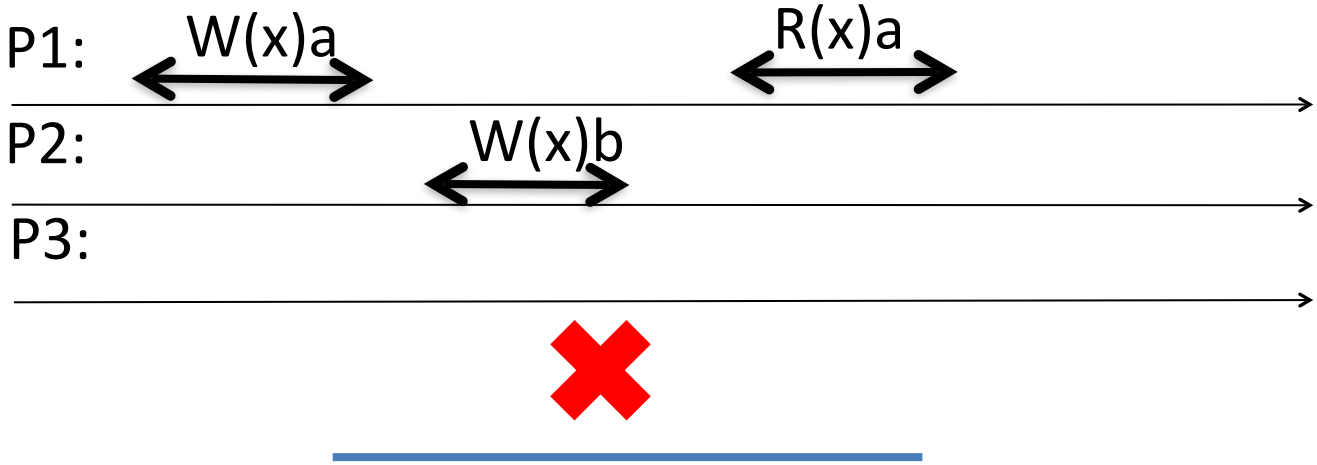
Example 3



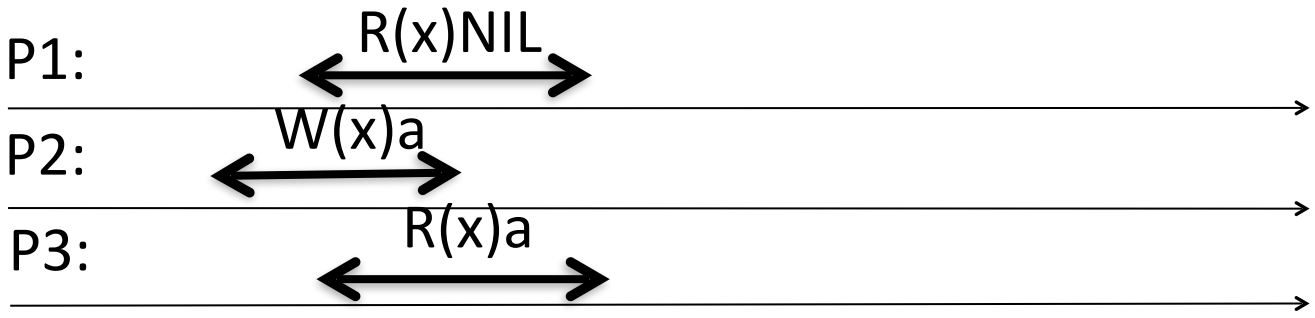


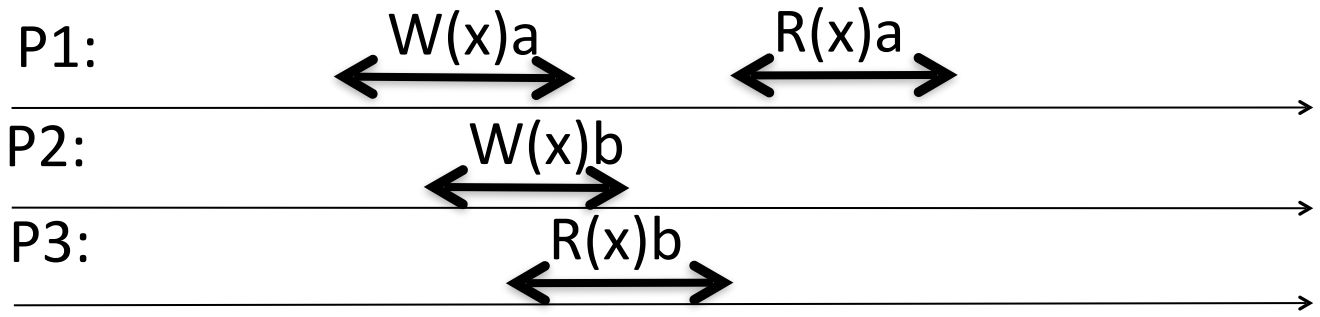
Example 3



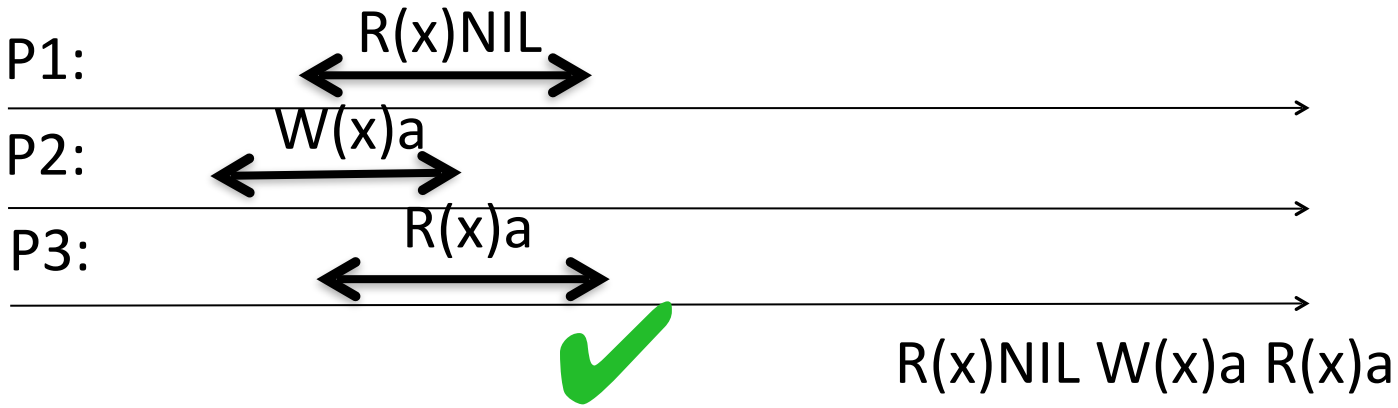


Example 4

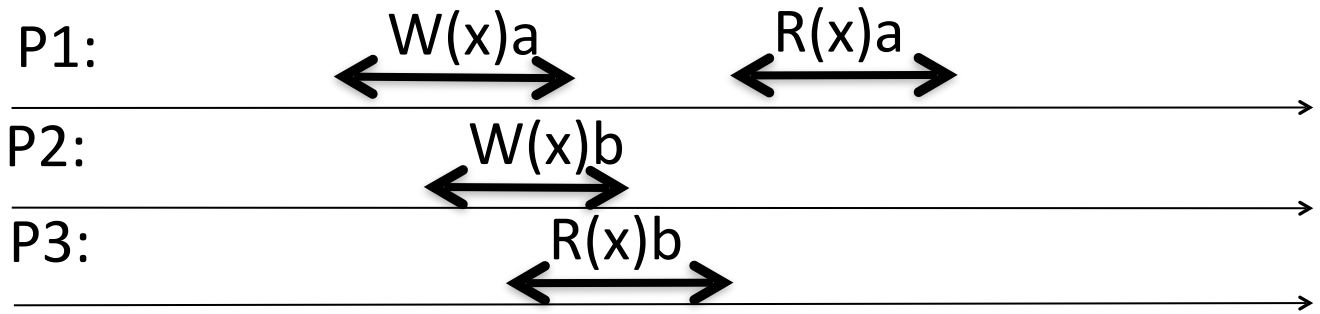




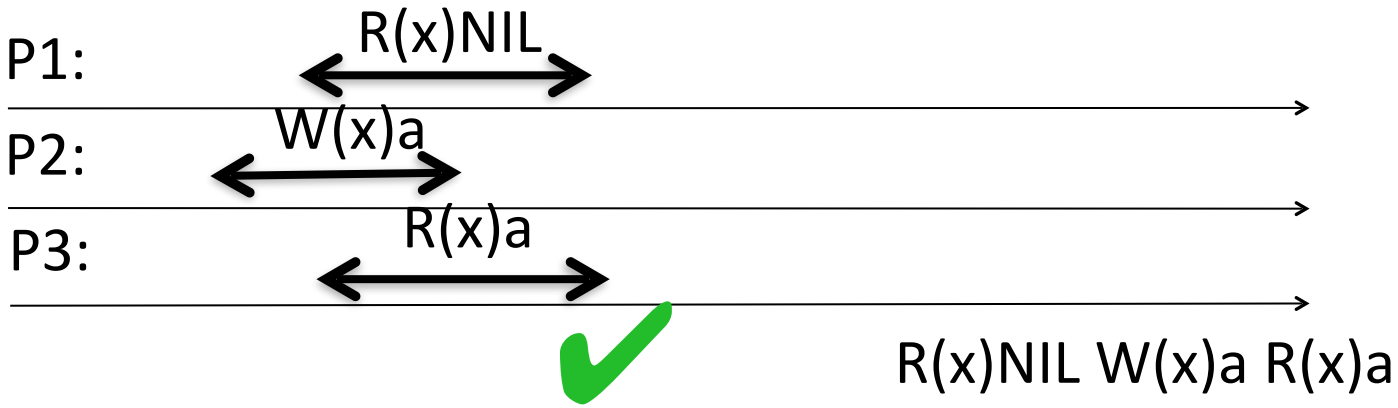
Example 4

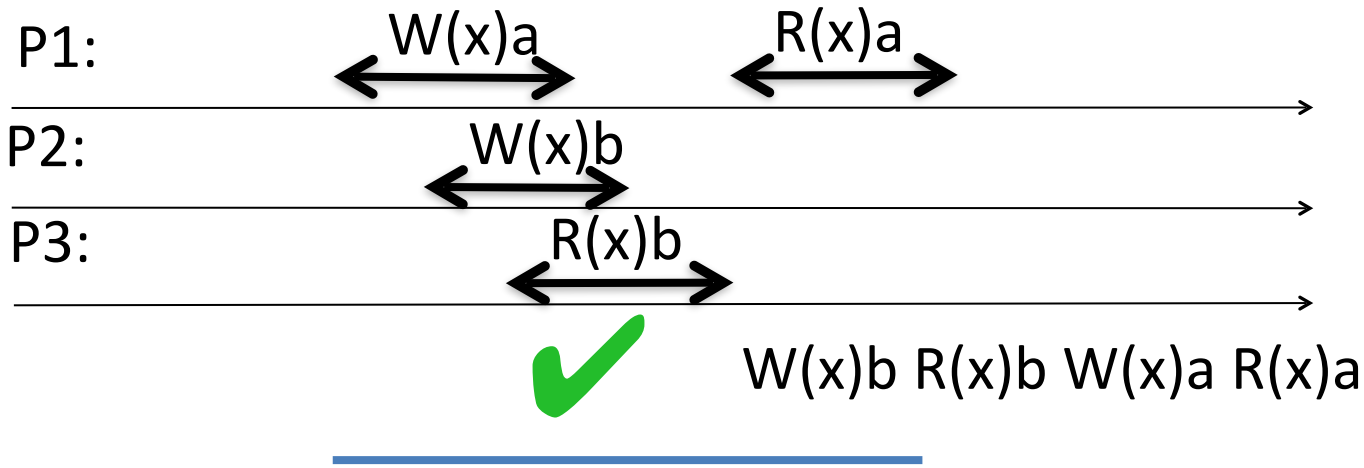


R(x)NIL W(x)a R(x)a

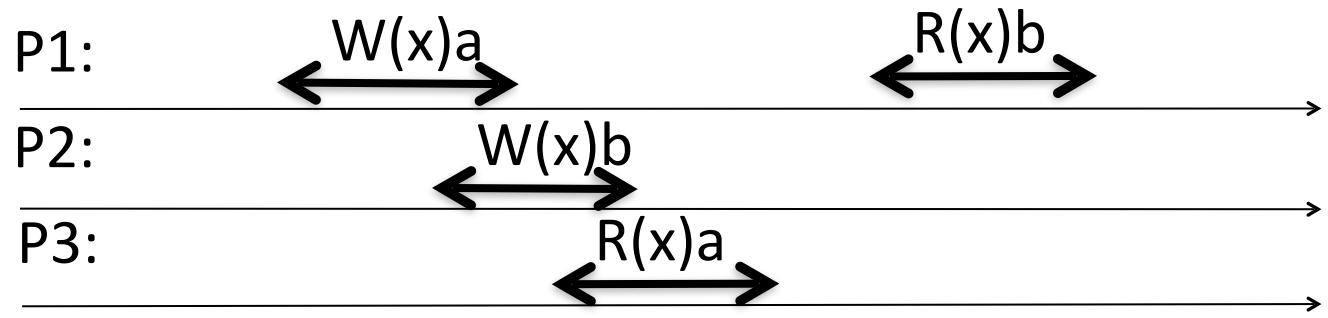


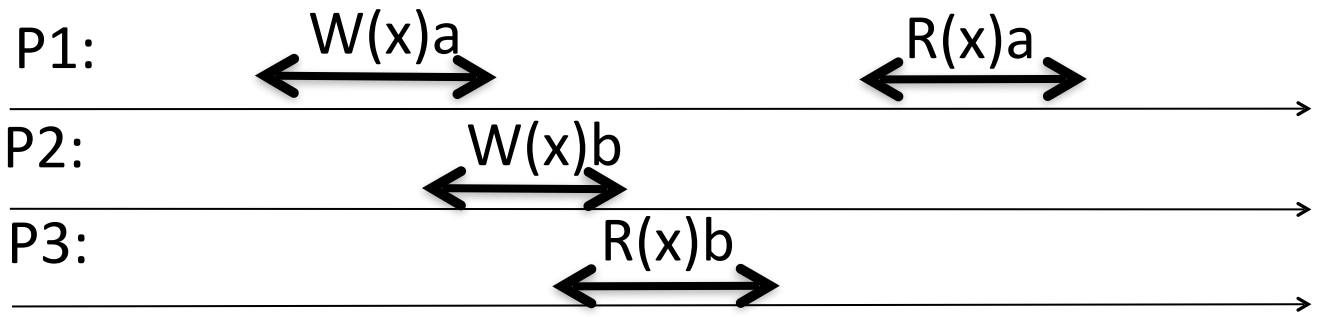
Example 4


P1:
R(x)NIL
P2:
W(x)a
P3:
R(x)a
R(x)NIL W(x)a R(x)a

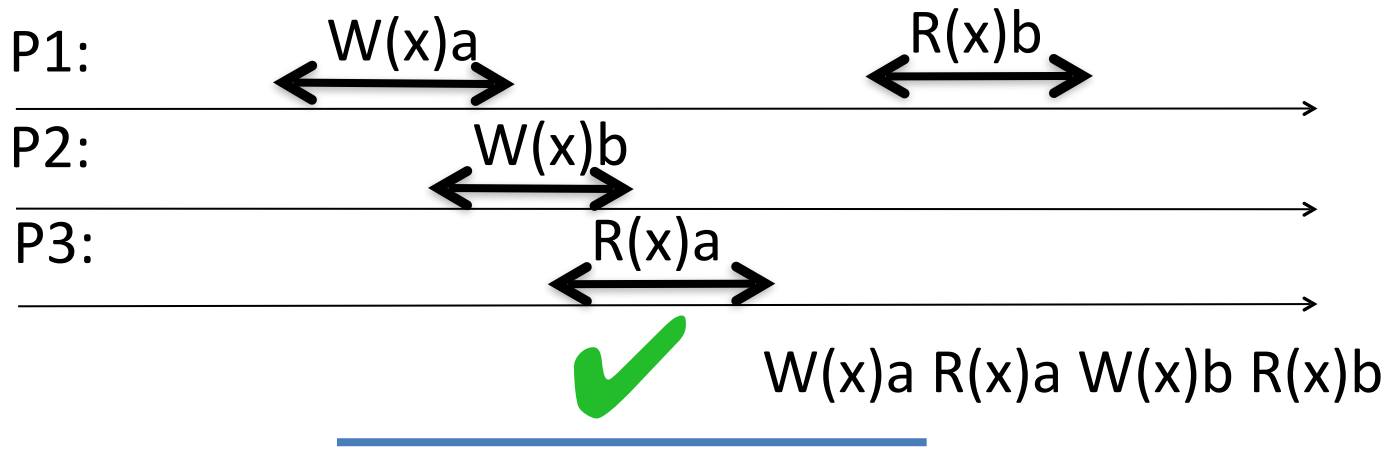

P1:
W(x)a
R(x)a
P2:
W(x)b
P3:
R(x)b
W(x)b R(x)b W(x)a R(x)a

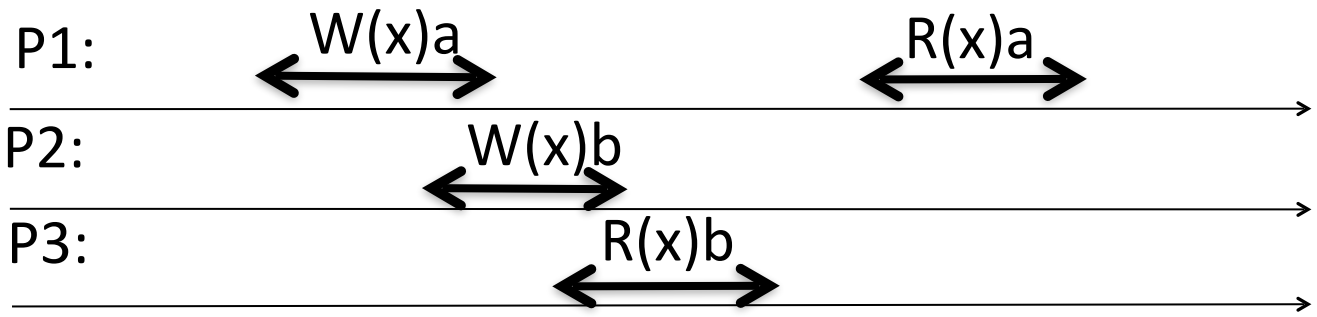
Example 5



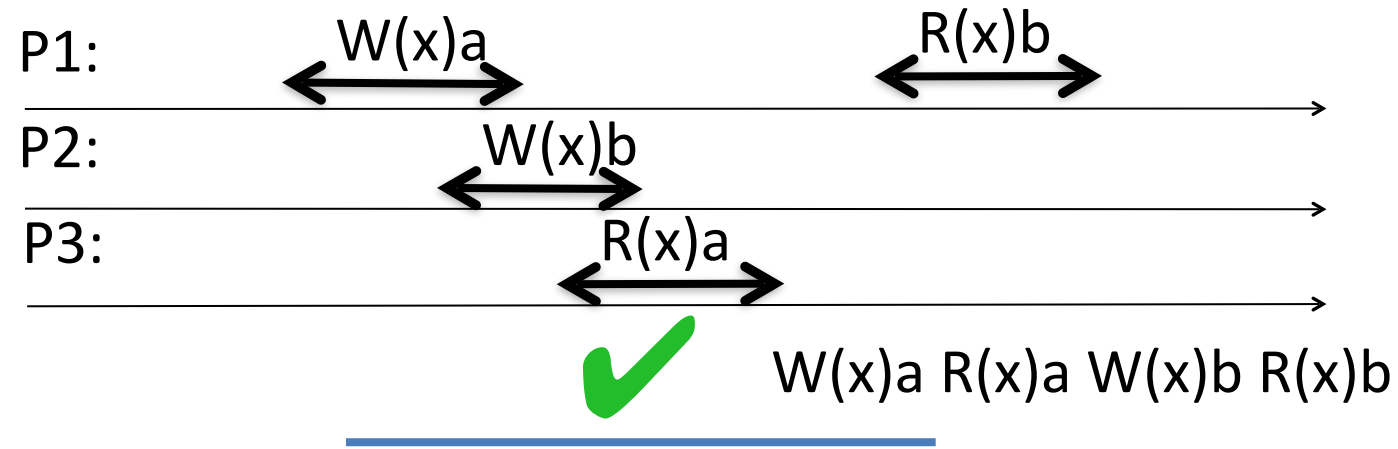


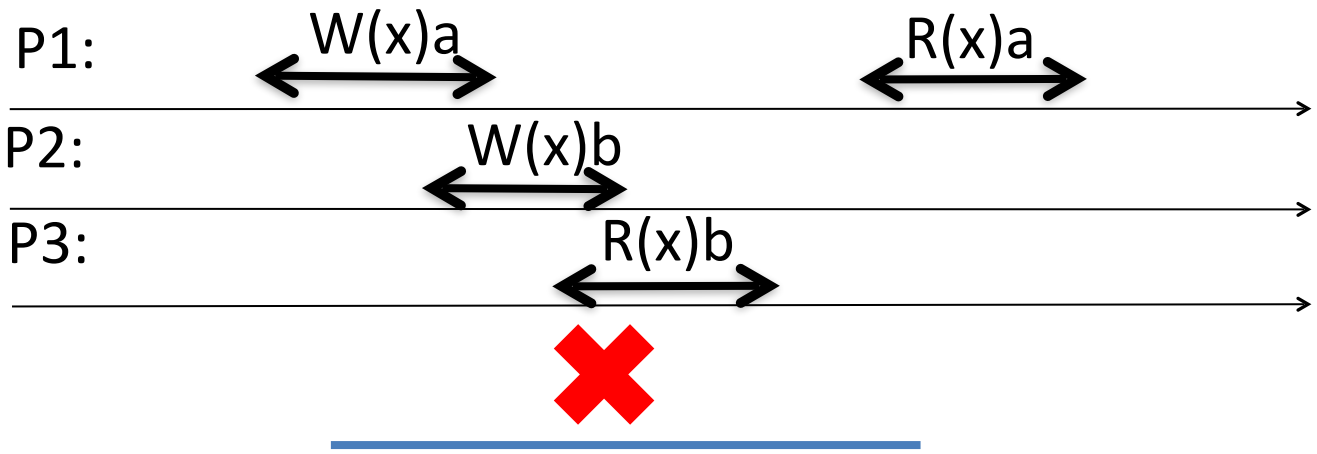
Example 5



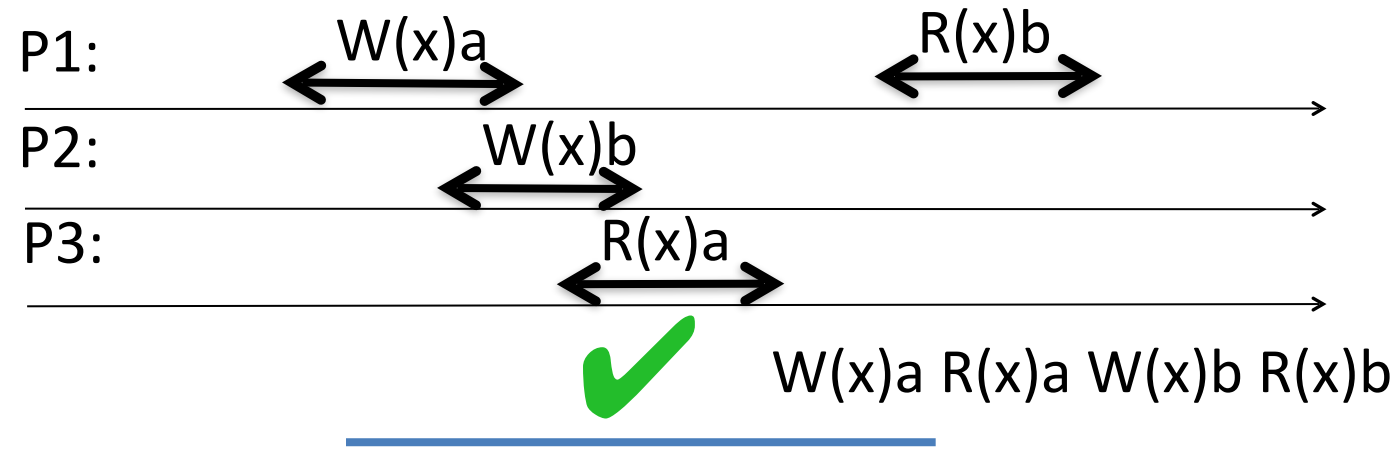


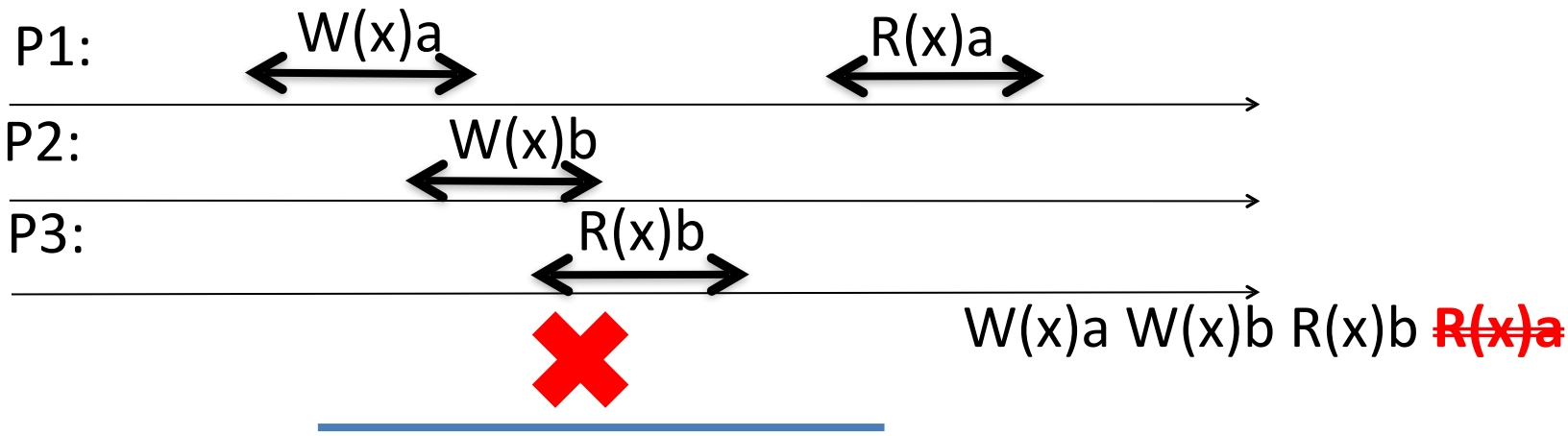
Example 5



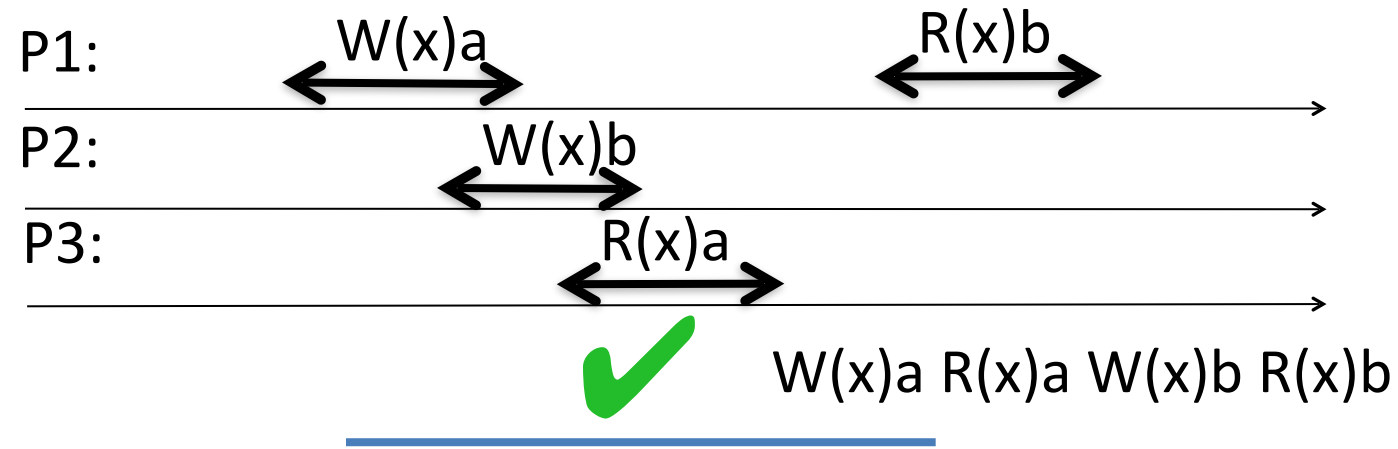


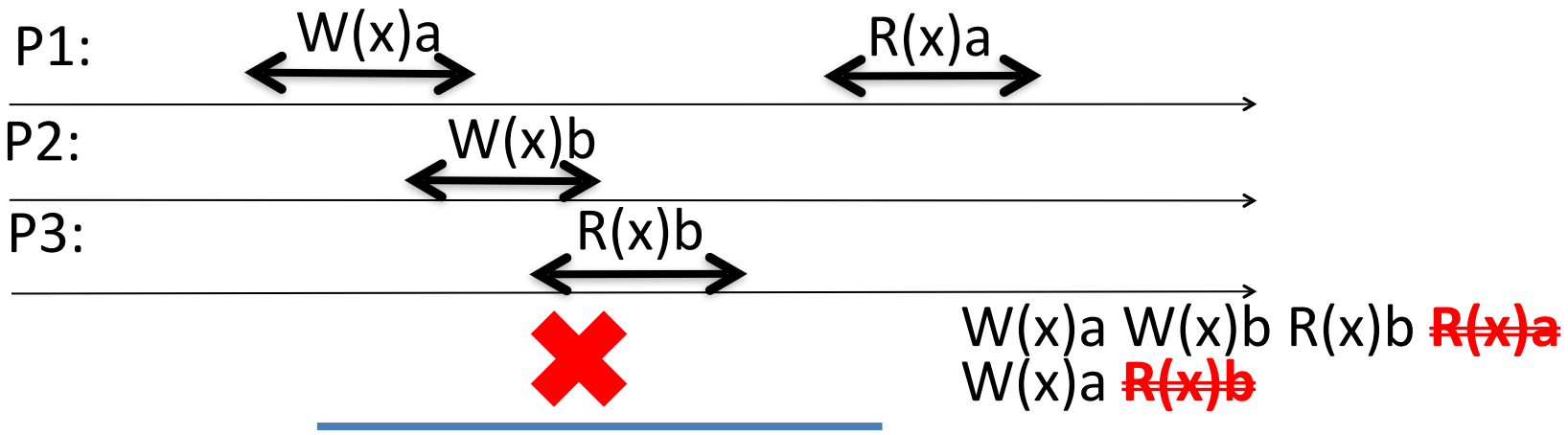
Example 5


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

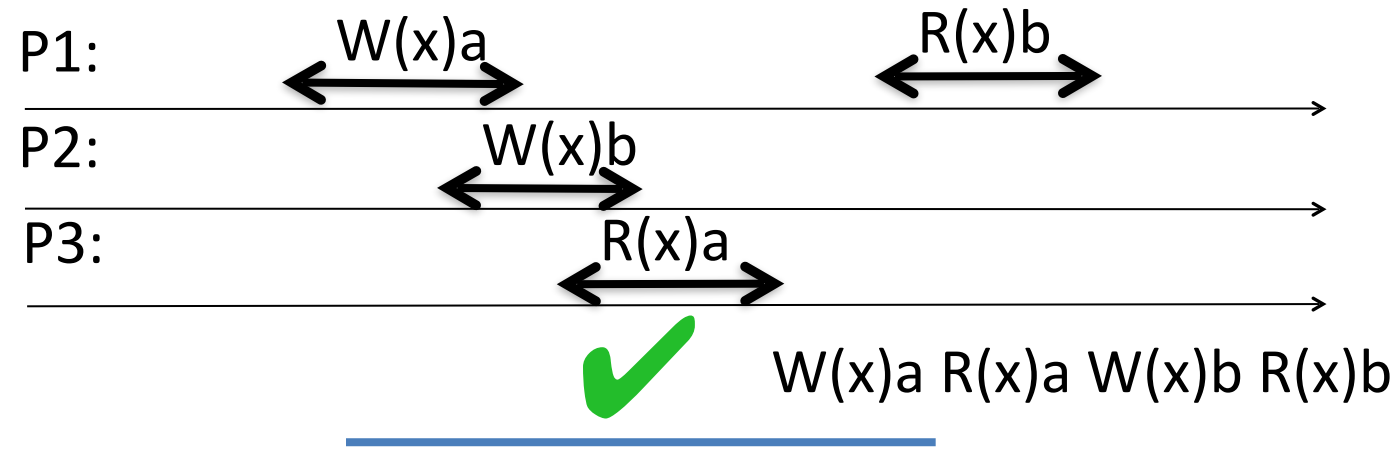

P1:
W(x)a
R(x)a
P2:
W(x)b
P3:
R(x)b
W(x)a W(x)b R(x)b R(x)a

Example 5

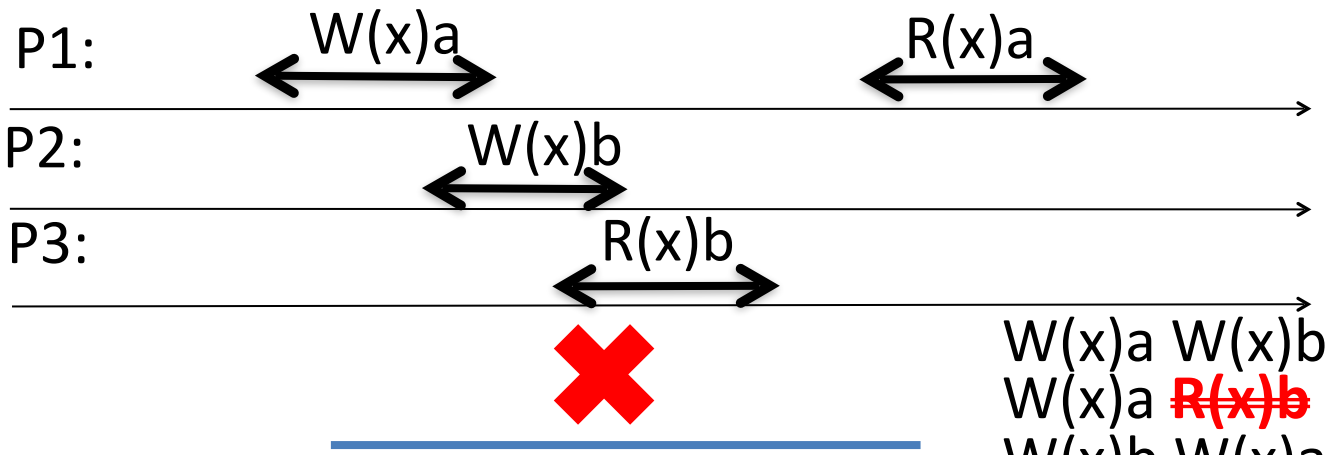

P1:
W(x)a
R(x)b
P2:
W(x)b
P3:
R(x)a
W(x)a R(x)a W(x)b R(x)b


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

Example 5

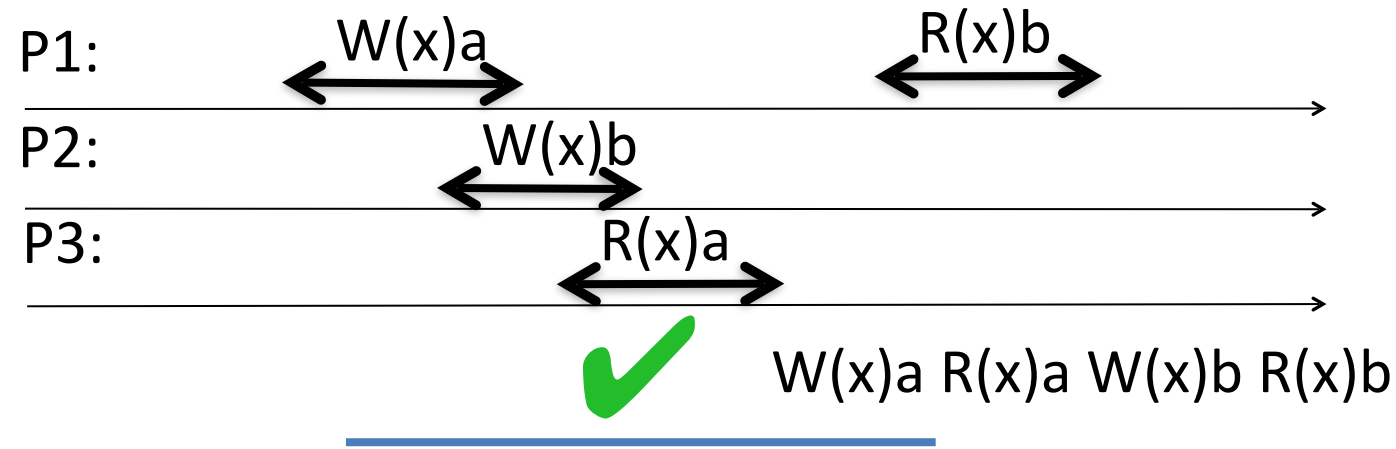


W(x)a R(x)a W(x)b R(x)b

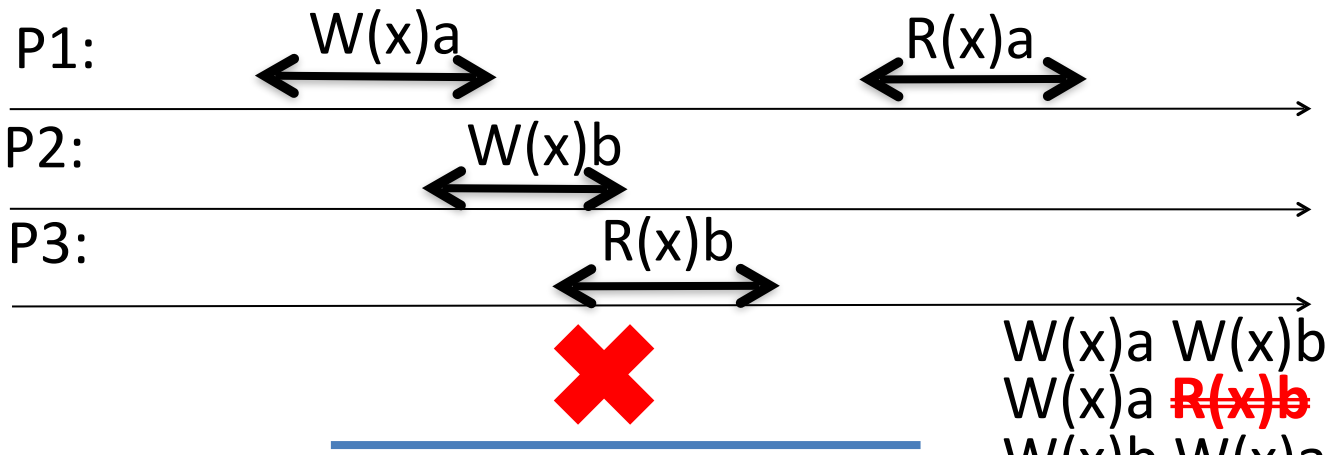


W(x)a W(x)b R(x)b ~~R(x)a~~
W(x)a ~~R(x)b~~
W(x)b W(x)a R(x)a ~~R(x)b~~

Example 5



✔ W(x)a R(x)a W(x)b R(x)b



✖

W(x)a W(x)b R(x)b ~~R(x)a~~
W(x)a ~~R(x)b~~
W(x)b W(x)a R(x)a ~~R(x)b~~
W(x)b W(x)a ~~R(x)b~~

Self-study questions

- Create some concurrent read and write sequences that conform to linearizability.
- Create some concurrent operation sequences that violate linearizability.
- Is a sequentially consistent data store linearizable, argue for or against?
- Is a strictly consistent system, sequentially consistent and vice versa, discuss?
- How could linearizability be implemented?

Pixabay.com

CAUSAL CONSISTENCY

Intuition for Causal Consistency

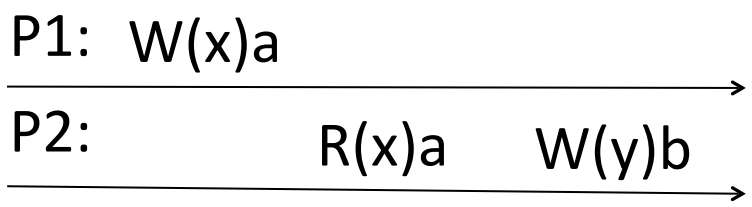
- **Weaker** than sequential consistency
- Distinguish events (writes) that are potentially **causally related and those that are not** (concurrent writes)
- Similar to **happened-before** relation (cf. Lamport clock), but with read and write operations instead of events
- If Write *B* is influenced (caused) by an earlier Write A (A→B), causal consistency requires that **every process first sees effect of *A* then *B***
- **Concurrent writes** may be seen in a different order by different processes

Causal Relationship

- **Read followed by write** in same process, both are **causally related**
  - **Example**: R(x)a → W(y)b (e.g., it may be that y=f(x))
- Read is **causally related** to write that provided value read got (different processes)
  - **Example**: W(x)a → R(x)a
- Transitivity: if $Op_1 \rightarrow Op_2$, $Op_2 \rightarrow Op_3$, then $Op_1 \rightarrow Op_3$
- **Independent writes** by **different processes** are not causally related - they are **concurrent**
  - **Example:** W(x)a || W(x)b

“Potentially” Causally Related

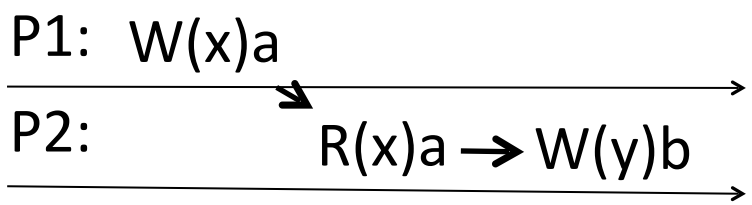
- **Example:**



- Writing of x in P1 and reading of x in P2 are causally related
- Reading of *x* in P2 and writing of *y* in P2 are **potentially** causally related
- Computation of *y* may have depended on value of *x* read by P2 (written by P1); e.g., *y = f(x)*
- On the other hand, *y* may not have depended on *x*, yet potential causality still holds in our formalization!
- There is a causal chain between writing of x in P1 and writing of y in P2 (via reading of x in P2)

“Potentially” Causally Related

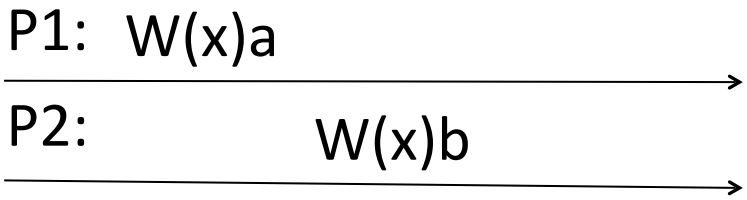
- **Example:**



- Writing of x in P1 and reading of x in P2 are causally related
- Reading of *x* in P2 and writing of *y* in P2 are **potentially** causally related
- Computation of *y* may have depended on value of *x* read by P2 (written by P1); e.g., *y = f(x)*
- On the other hand, *y* may not have depended on *x*, yet potential causality still holds in our formalization!
- There is a causal chain between writing of x in P1 and writing of y in P2 (via reading of x in P2)

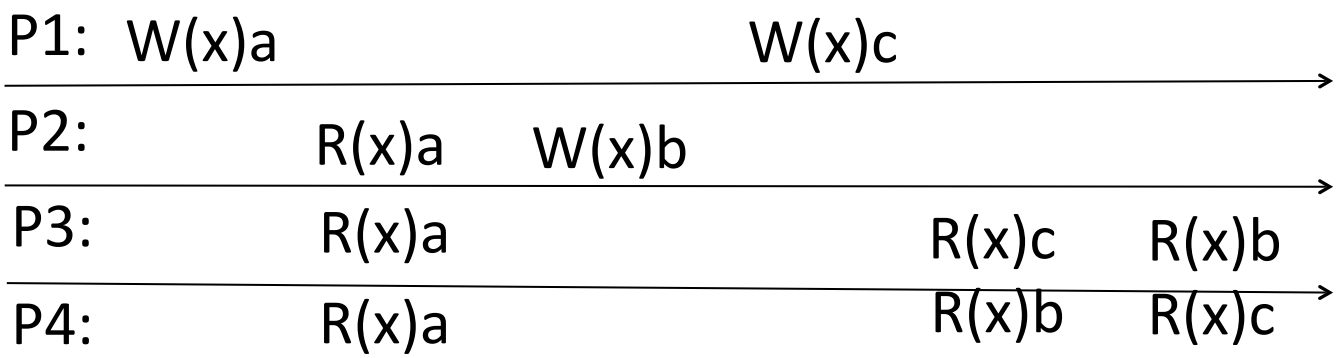
Definition of Causal Consistency

- "Potentially" causally related writes must be seen by all processes in the same order
- **Concurrent writes** may be seen in a different order by different processes



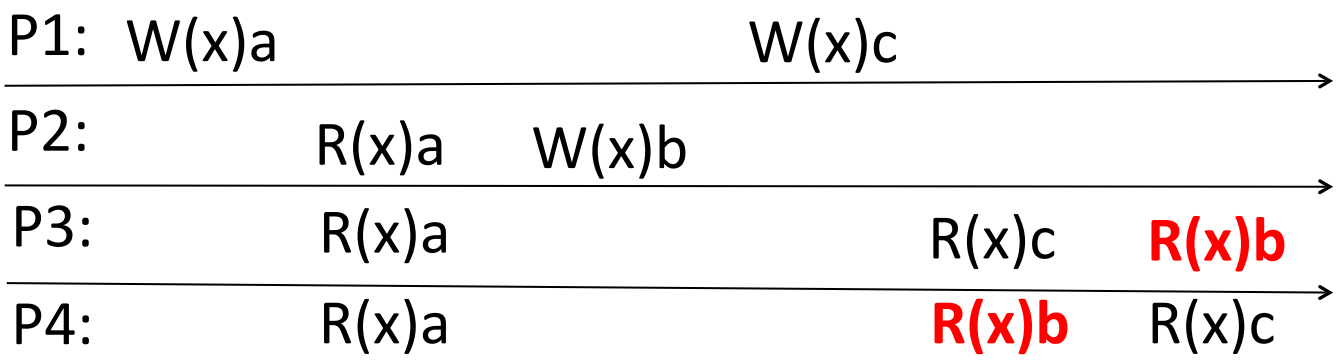
- Program order must be met

Causal Consistency Example I



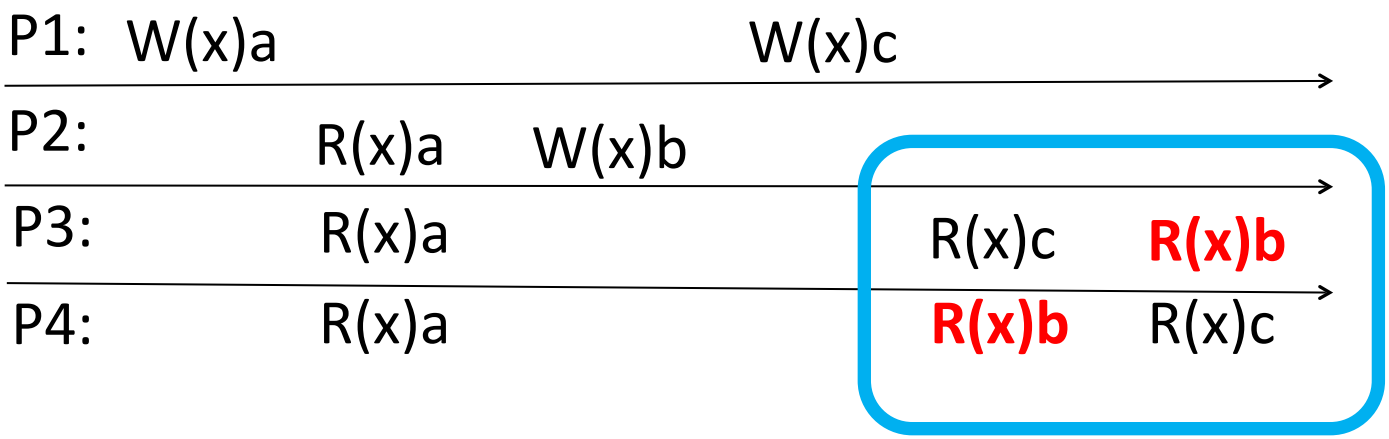
Strictly, sequentially or causally consistent?

Causal Consistency Example I



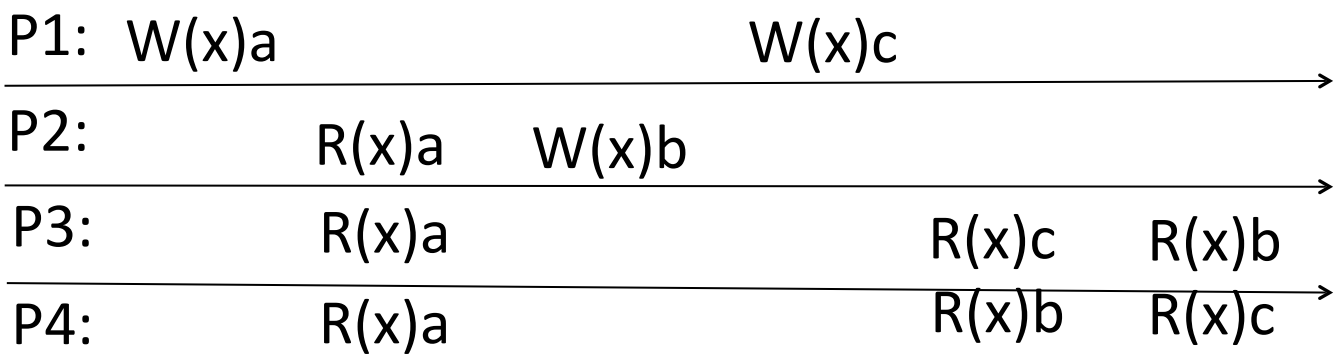
Neither **strictly**, nor sequentially consistent, …

Causal Consistency Example I

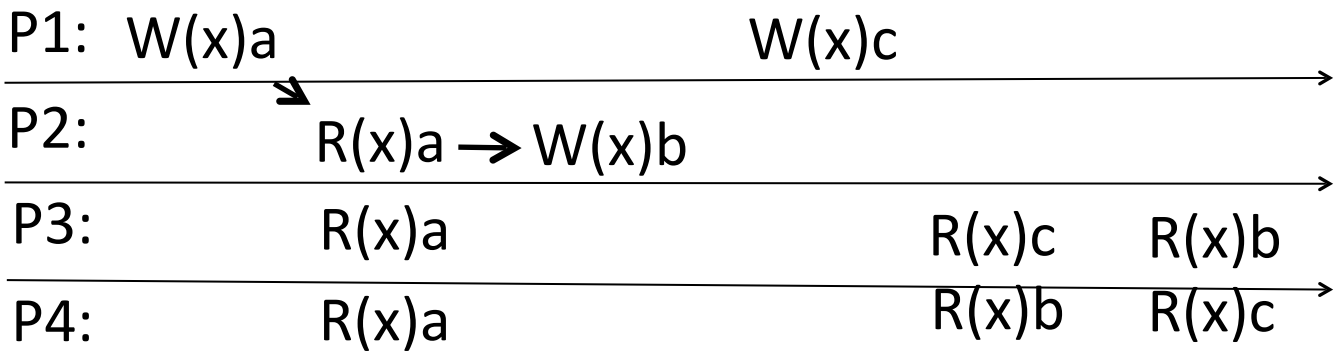


Neither **strictly**, nor sequentially consistent, …

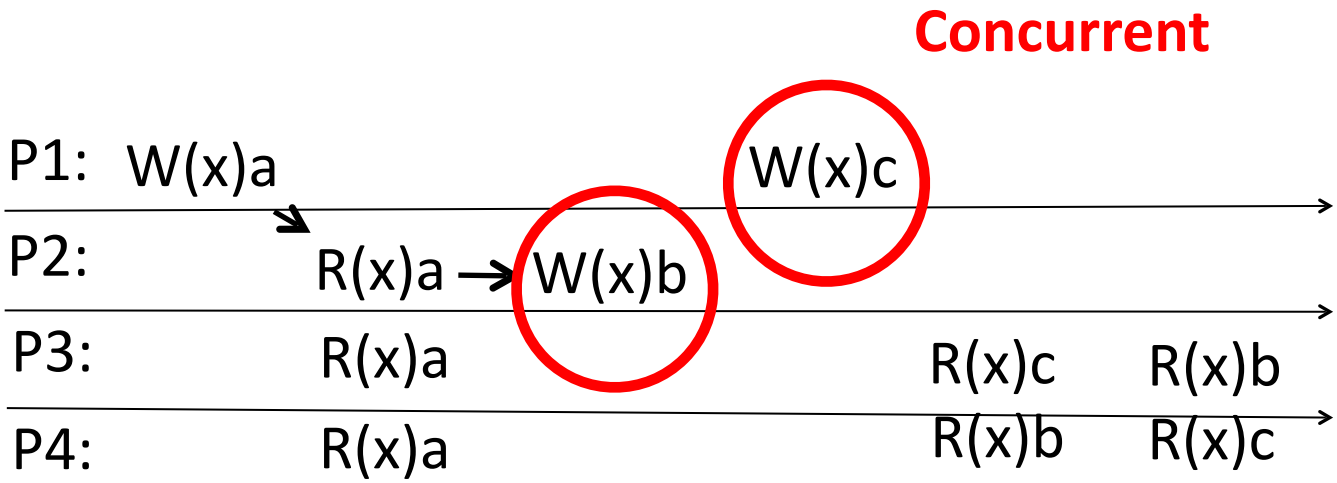
Causal Consistency Example I



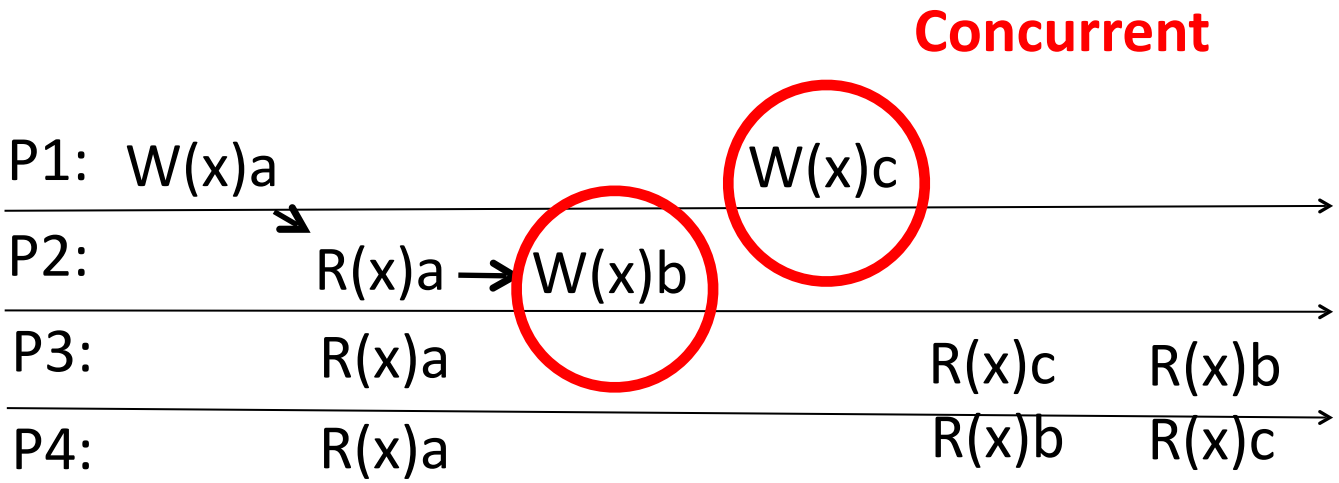
Causal Consistency Example I



Causal Consistency Example I

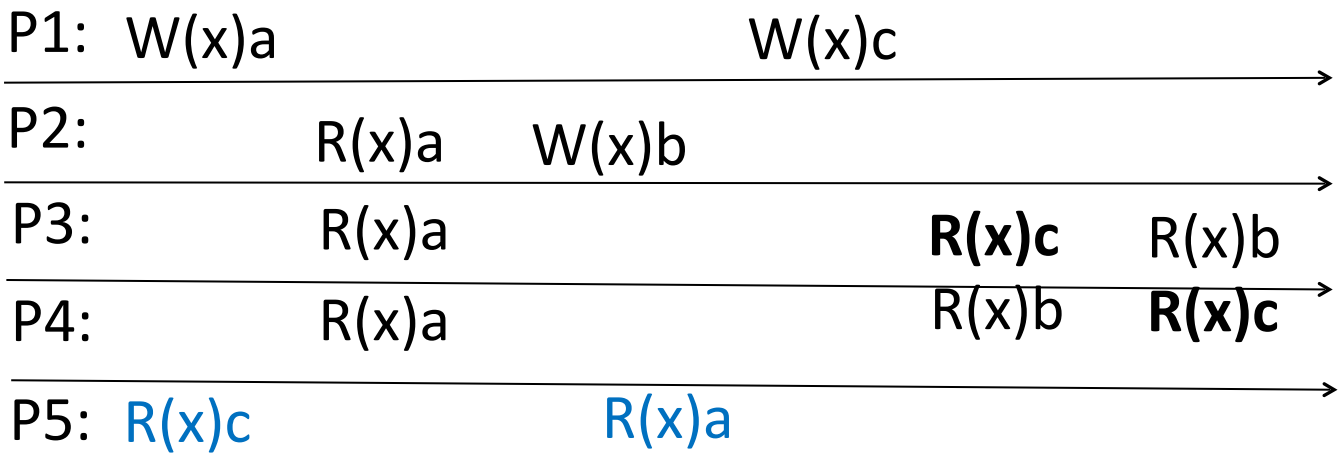


Causal Consistency Example I

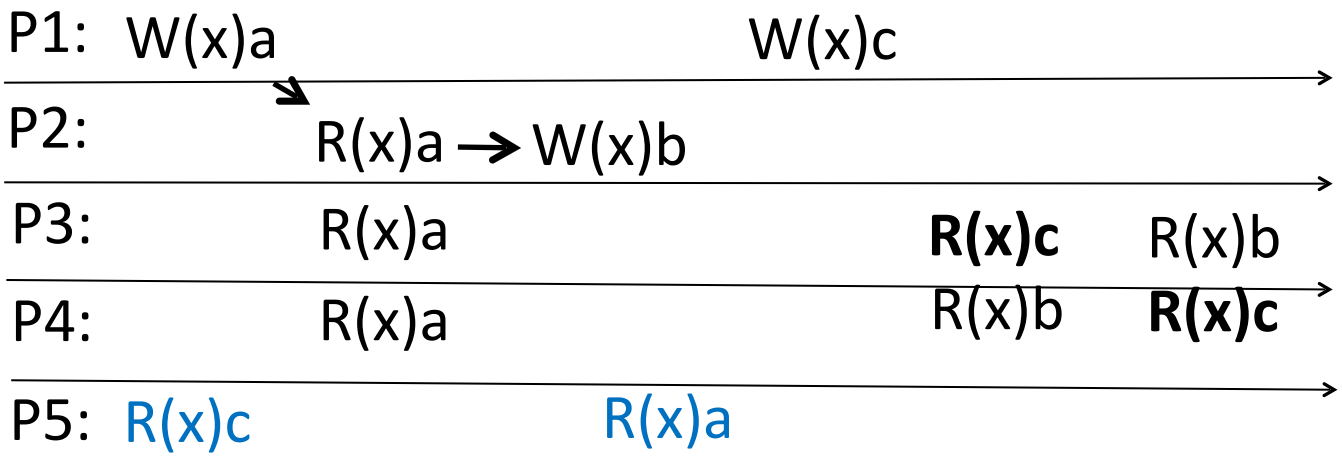


Neither strictly, nor sequentially consistent,
but causally consistent!

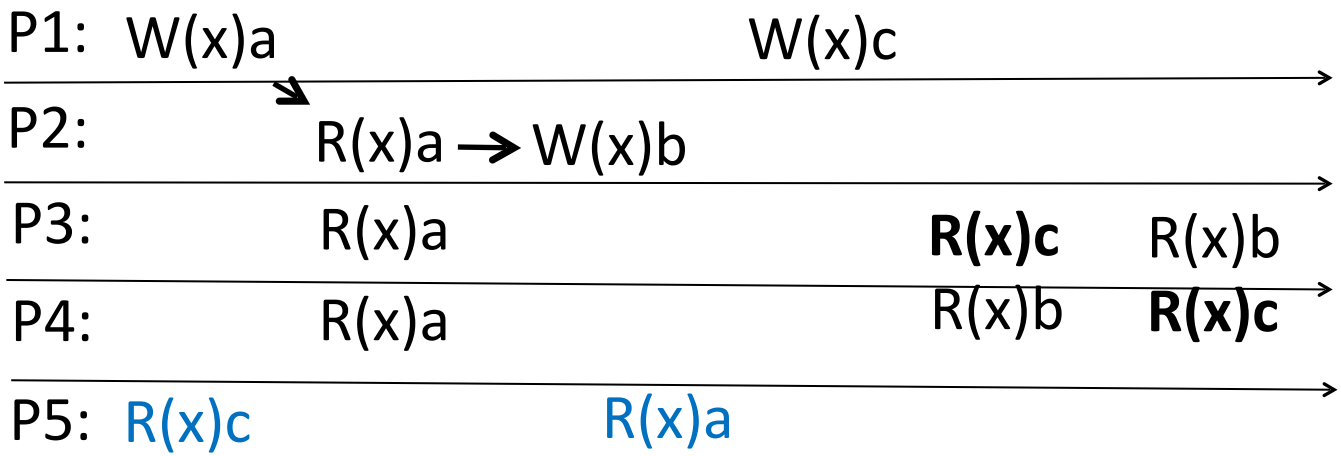
Causal Consistency Example I’



Causal Consistency Example I’



Causal Consistency Example I’



Not causally consistent, W(x)a happened-before W(x)c in P1

Self-study questions

- Create some concurrent read and write sequences that conform to causal consistency.
- Create some concurrent operation sequences that violate causal consistency.
- Is a sequentially consistent data store causally consistent, argue for or against?
- Is a causally consistent data store, sequentially consistent, argue for or against?
- How could causal consistency be implemented?

FIFO CONSISTENCY

FIRST IN – FIRST OUT

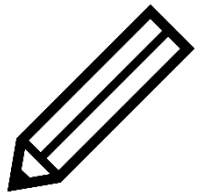
Definition of FIFO Consistency

- Writes by a single process are seen by all other processes in the order in which they were issued

- Writes from different processes may be seen in a different order by different processes

- Easy to maintain: Simply send writes in FIFO order from each process to each replica (e.g., using TCP)

FIFO Consistency Example I



P1: W(x)a

P2: R(x)a **W(x)b** **W(x)c**

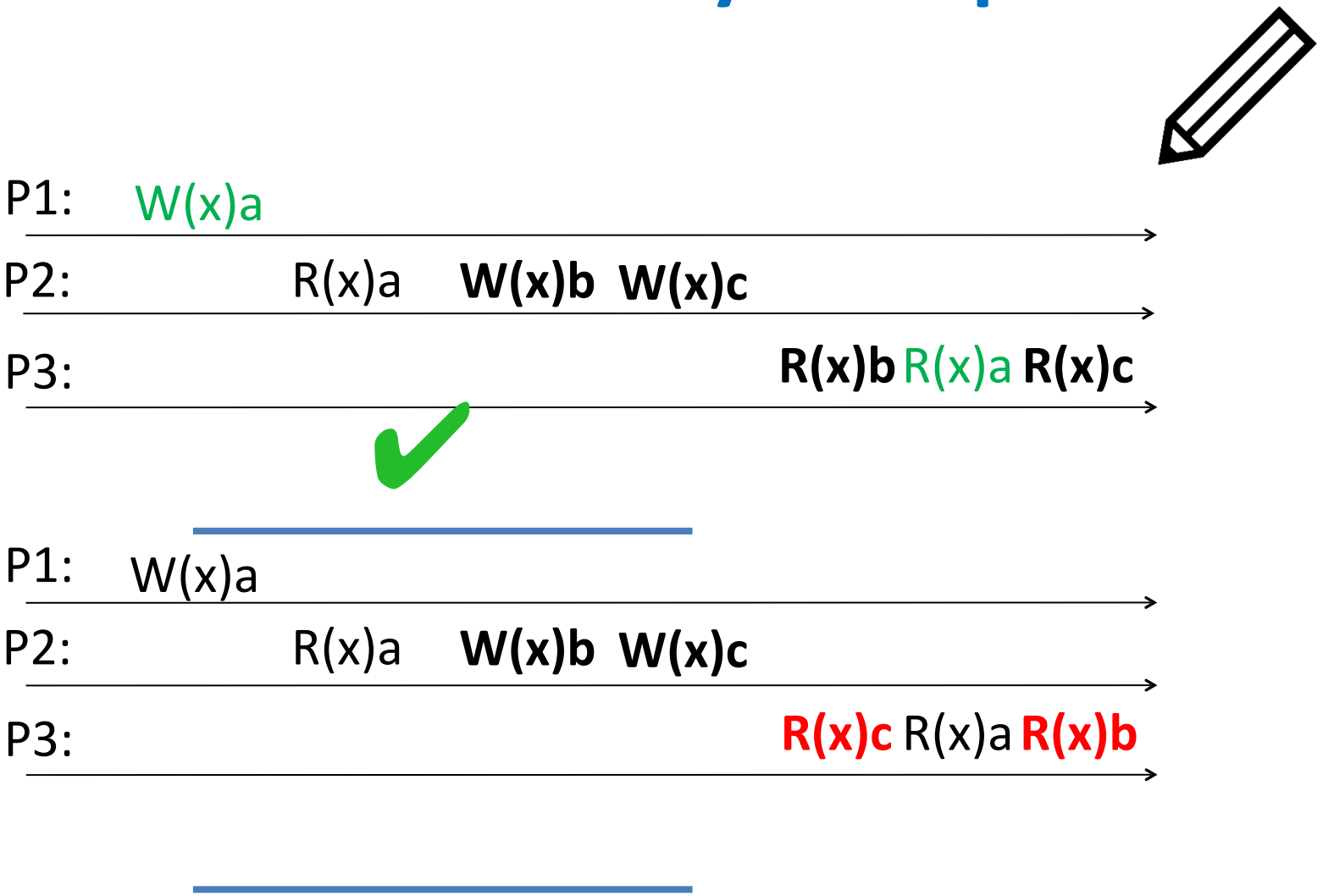
P3: **R(x)b** R(x)a **R(x)c**

P1: W(x)a

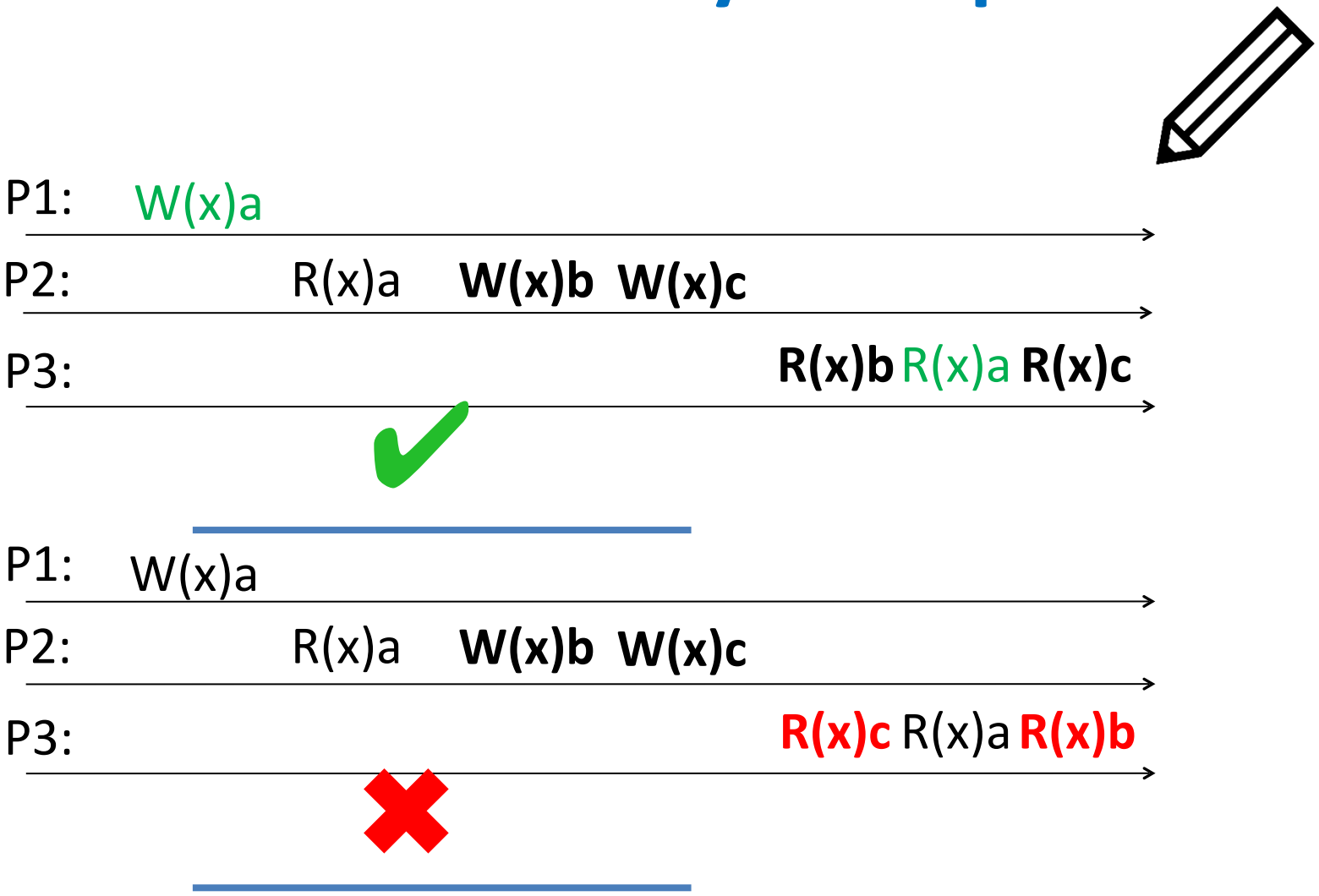
P2: R(x)a **W(x)b** **W(x)c**

P3: **R(x)c** R(x)a **R(x)b**

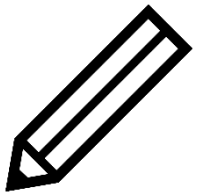
FIFO Consistency Example I

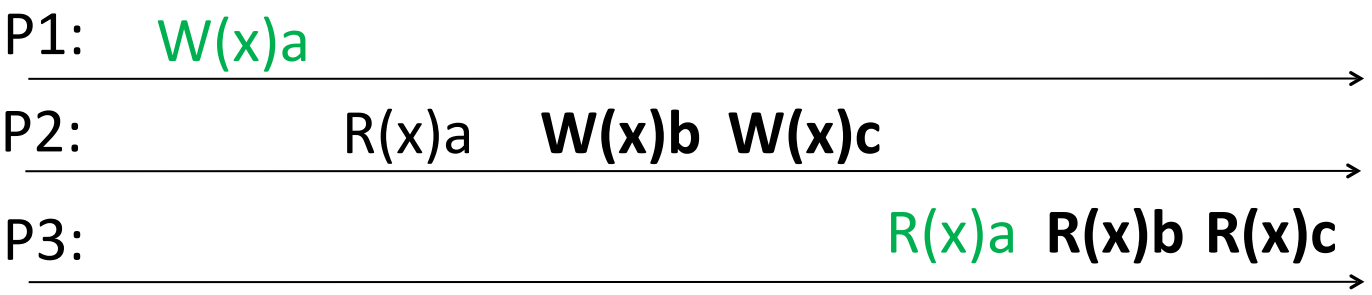


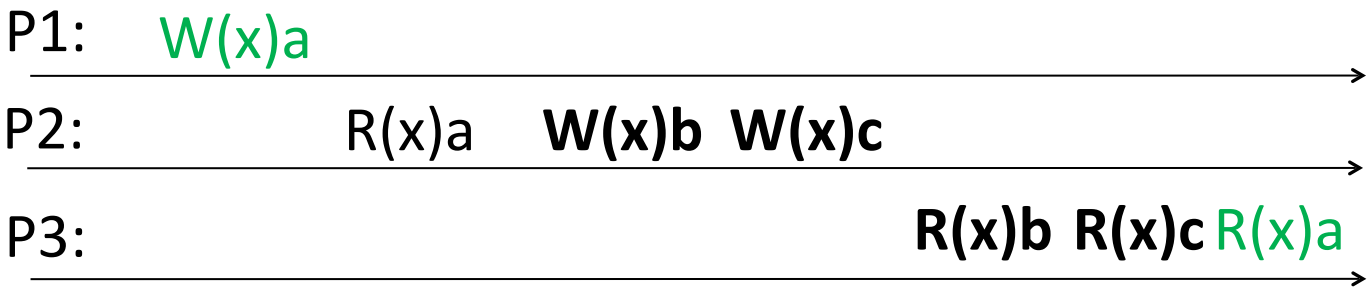
FIFO Consistency Example I



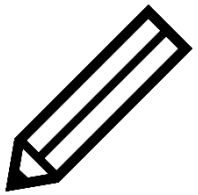
FIFO Consistency Example II

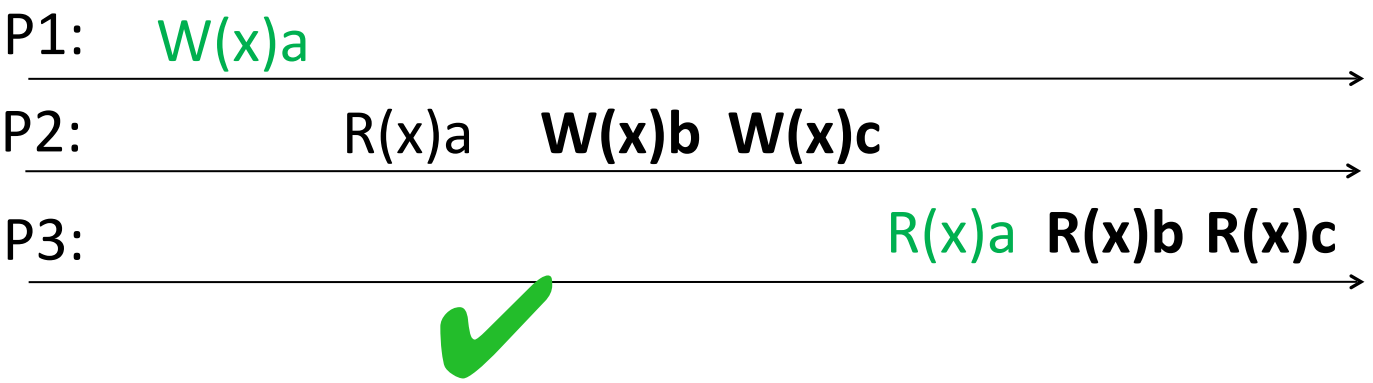


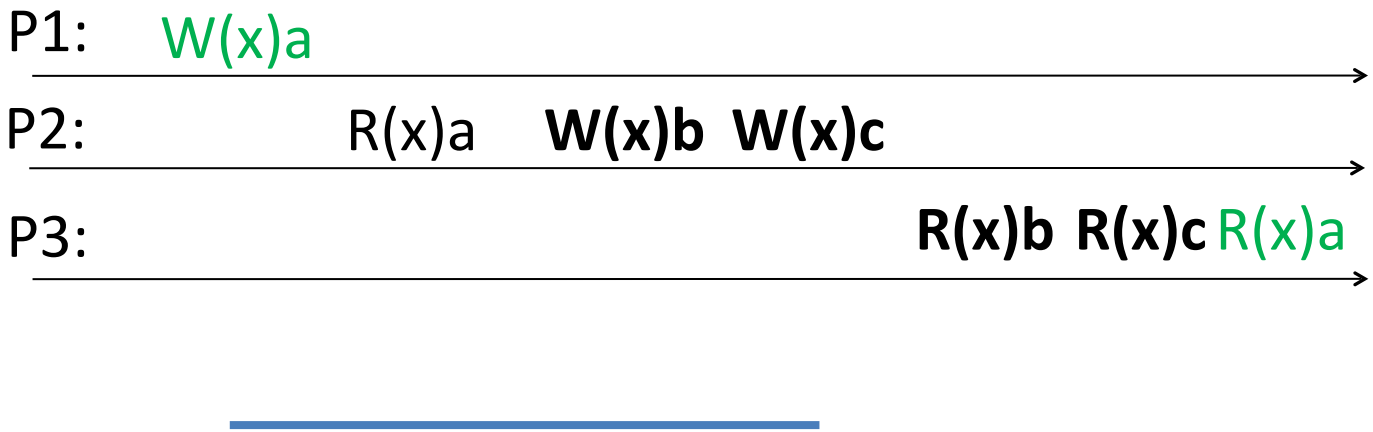




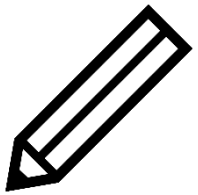
FIFO Consistency Example II

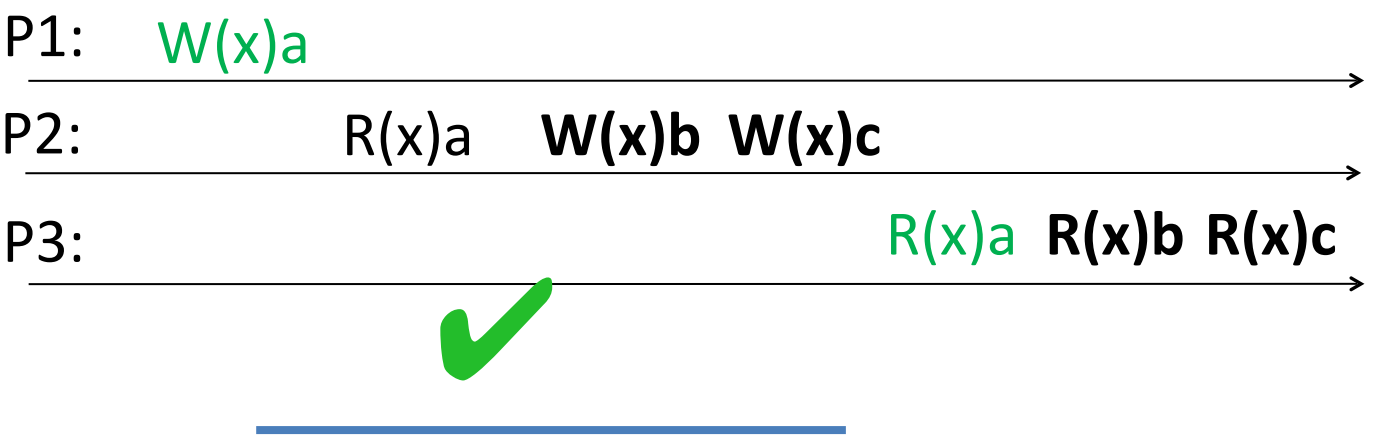


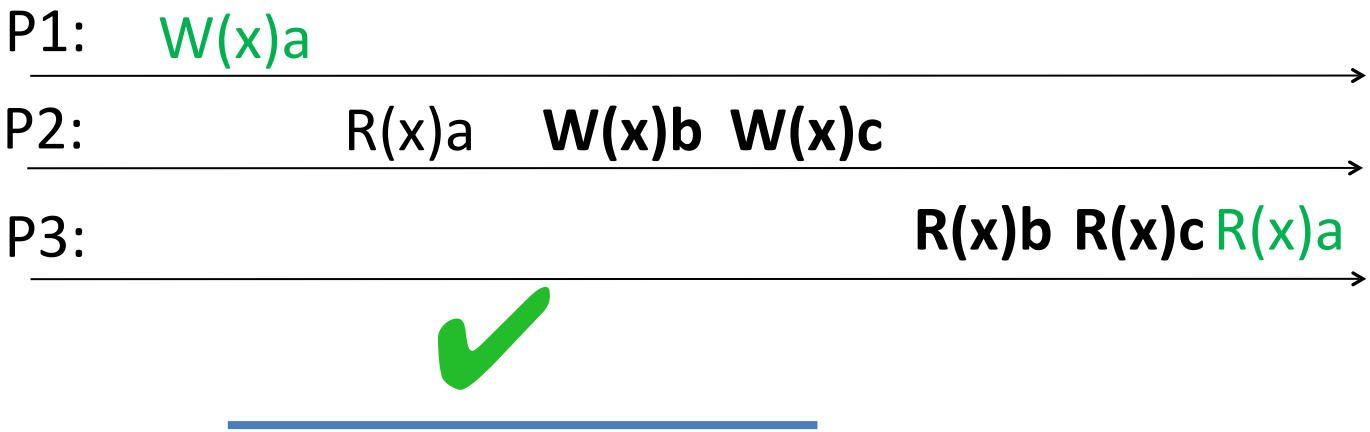




FIFO Consistency Example II







Self-study Questions

- Create some concurrent read and write sequences that conform to FIFO consistency.
- Create some concurrent operation sequences that violate FIFO consistency.
- Is a sequentially consistent system, FIFO consistent, argue for or against?
- Is a causally consistent system, FIFO consistent, argue for or against?
- Find sample applications of FIFO consistency.

Pixabay.com

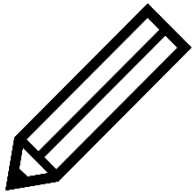
WEAK CONSISTENCY

Weak Consistency

- Not all processes need to see all writes, let alone in same order
- Based on **synchronization variable *S*** with single operation ***Synchronize(S)***
- Any process can perform read/write operations and synchronize
- Order of operations **before synchronization** is not consistent
- After synchronization, all processes see **same outcome of operations** preceding synchronization point

Weak Consistency Interpretation

- Process forces the just written value out to all synchronizing replicas
- Process can be sure to get the most recent value written before it reads
- Model enforces **consistency on a group of operations** as opposed to individual reads and writes
- Care about the **effect of a group** of reads and writes



Weak Consistency Example I

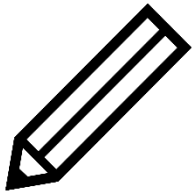
P1: W(x)a W(x)b S

P2: R(x)a R(x)b S

P3: R(x)b R(x)a S

P1: W(x)a W(x)b S

P2: S R(x)a



Weak Consistency Ex

P2 & P3 have yet to synchronize, no guarantees about values read

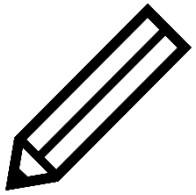
P1: W(x)a W(x)b S

P2: R(x)a R(x)b S

P3: R(x)b R(x)a S

P1: W(x)a W(x)b S

P2: S R(x)a



Weak Consistency Ex

P2 & P3 have yet to synchronize, no guarantees about values read

P1: W(x)a W(x)b S

P2: R(x)a R(x)b S

P3: R(x)b R(x)a S

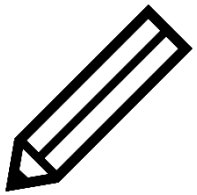
✔

P1 propagates value b

P1: W(x)a W(x)b S

P2: S R(x)a

✖



Weak Consistency Example II

Agreement on value
is implementation dependent.

P1: W(x)a W(x)b S R(x)a

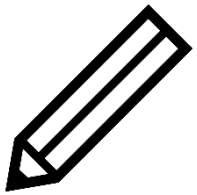
P2: W(x)a S R(x)a

P3: R(x)b R(x)a S

P1: W(x)a W(x)b S R(x)b

P2: W(x)a S R(x)b

P3: R(x)b R(x)a S



Weak Consistency Example II

Agreement on value
is implementation dependent.

P1: W(x)a W(x)b S R(x)a

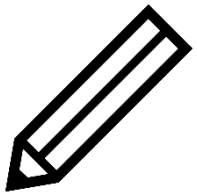
P2: W(x)a S R(x)a

P3: R(x)b R(x)a S

P1: W(x)a W(x)b S R(x)b

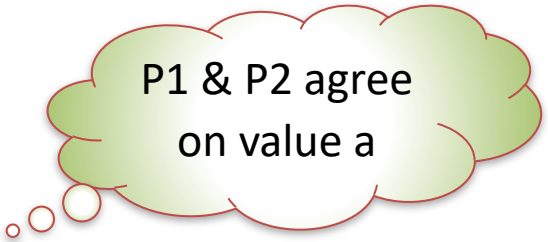
P2: W(x)a S R(x)b

P3: R(x)b R(x)a S



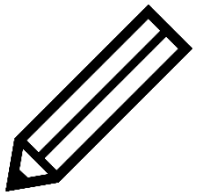
Weak Consistency Example II

Agreement on value
is implementation dependent.



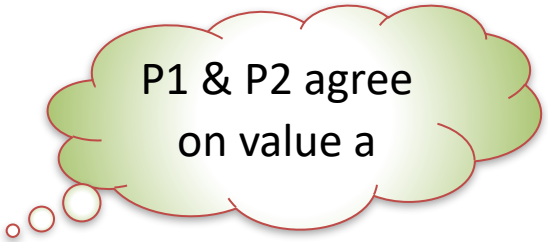
P1:	W(x)a	W(x)b		S		R(x)a	
P2:		W(x)a		S	R(x)a		
P3:					R(x)b	R(x)a	S

P1:	W(x)a	W(x)b		S		R(x)b	
P2:		W(x)a		S	R(x)b		
P3:					R(x)b	R(x)a	S



Weak Consistency Example II

Agreement on value
is implementation dependent.



P1: W(x)a W(x)b S R(x)a

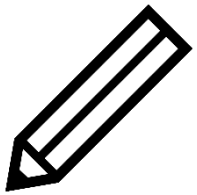
P2: W(x)a S R(x)a

P3: R(x)b R(x)a S

P1: W(x)a W(x)b S R(x)b

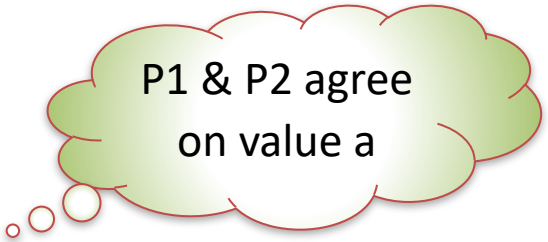
P2: W(x)a S R(x)b

P3: R(x)b R(x)a S



Weak Consistency Example II

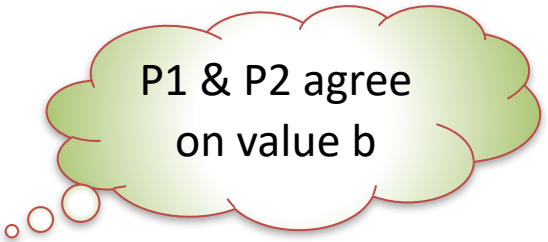
Agreement on value
is implementation dependent.



P1: W(x)a W(x)b S R(x)a

P2: W(x)a S R(x)a

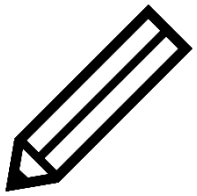
P3: R(x)b R(x)a S



P1: W(x)a W(x)b S R(x)b

P2: W(x)a S R(x)b

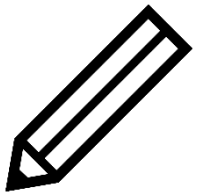
P3: R(x)b R(x)a S



Weak Consistency Example III

P1:	W(x)a	W(x)b	S		R(x)a	
P2:		W(x)a	S	R(x)b		
P3:				R(x)b	R(x)a	S

P1:	W(x)a	W(x)b	S		R(x)b	
P2:		W(x)a	S	R(x)a		
P3:				R(x)b	R(x)a	S



Weak Consistency Example III

P1: W(x)a W(x)b S R(x)a

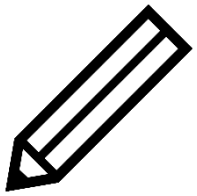
P2: W(x)a S R(x)b

P3: R(x)b R(x)a S

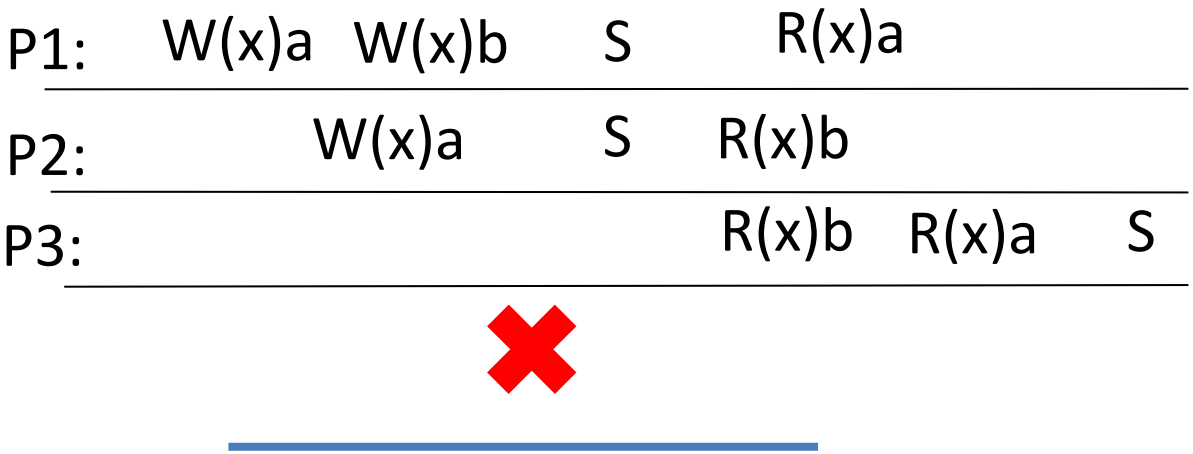
P1: W(x)a W(x)b S R(x)b

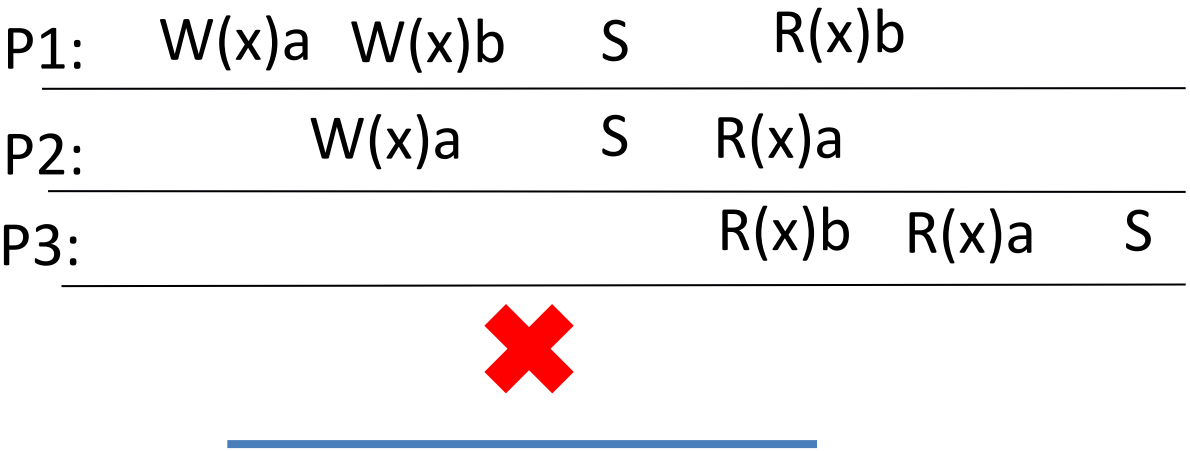
P2: W(x)a S R(x)a

P3: R(x)b R(x)a S



Weak Consistency Example III





Self-study Questions

- Create some concurrent read and write sequences that conform to weak consistency.
- Can weak consistency emulate any of the other consistency models we study?
- How can a weakly consistent data store be implemented?
- Find sample applications of weak consistency.

DATA-CENTRIC CONSISTENCY MODELS SUMMARY

Summary of Consistency Models

Strongest

Consistency	Description
Strict	**Absolute time** ordering of all writes
Linear-izability	All processes see all **writes in same order**. Accesses **ordered** according to a (non-unique) global timestamp
Sequential	All processes see all **writes in same order** Accesses are **not ordered in time**
Causal	All processes see **causally-related writes in same order; concurrent writes must not be ordered**
FIFO	All processes **see writes from each other in the order they were issued**; writes from different processes must not be seen in that order
Weak	Shared data can be counted on to be consistent only **after synchronizing**

Weakest

Where does … come into the picture?

- **Linearizability**: Strongest distributed solution, possible with eager replication (synchronous), chain replication, supported in Hbase, Bigtable
- **Sequential**: System-wide consistent reads, e.g., everyone sees replies to a post in same order
- **Causal**: Everyone sees posts before replies
- **FIFO**: Reading all messages from each friend in order but not across friends
- **Weak**: Responsibility left to developer who must explicitly enforce synchronization

Self-study Questions

- Create some concurrent read and write sequences that conform to different consistency models.
- How do consistency models relate to one another, if at all?
- Find sample applications for each consistency model.

CC

CLIENT-CENTRIC CONSISTENCY

Client-centric Consistency

- So far, goal was to maintain consistency in presence of **concurrent read** and **write** operations
- There are use cases with **no (few) concurrent writes to the same key,** or consistency of write operations are secondary
  - **DNS**: **No write-write conflicts** since there is a single authority updating each domain (disjoint partitions)
  - **Key-value stores**: **Usually no write-write conflicts** since updates partitioned by keys, e.g., Dynamo, Cassandra (also, optimistic concurrency control)
  - **WWW**: Heavy use of client-side caching, reading stale pages is acceptable in many cases

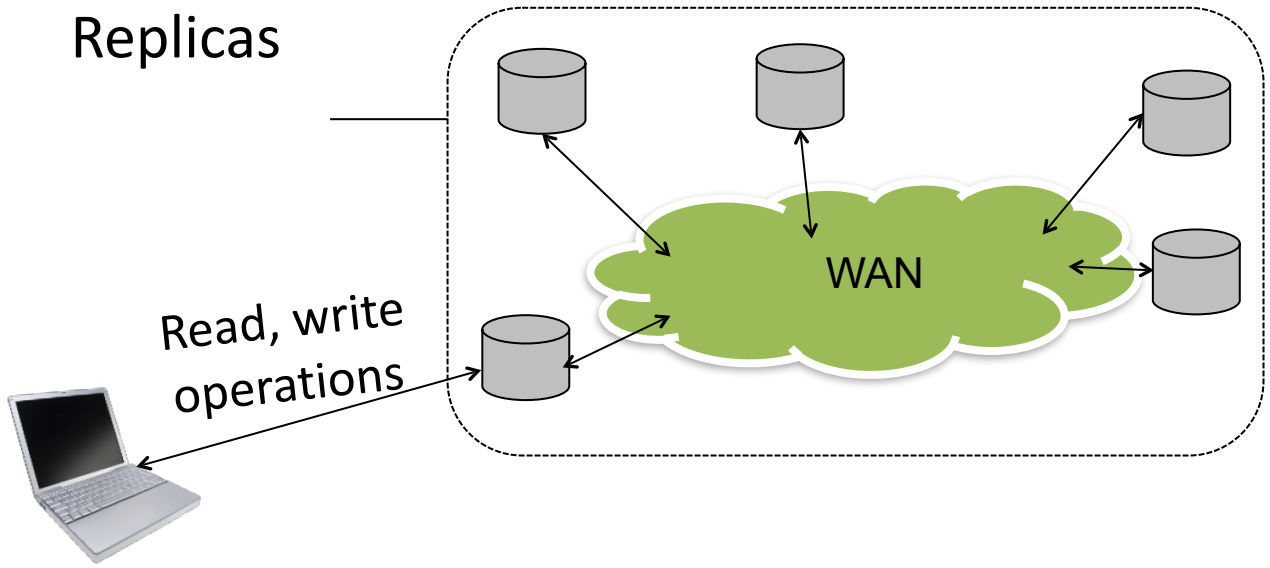
Client-centric Consistency

- **Client-centric** consistency puts emphasis on maintaining a **consistent view** for a **single process**, instead of on data stored in system
- Emphasis on **read-write conflicts**, and we assume **write-write** conflicts do not exist (i.e., no (few) concurrent writes)
- Client-centric consistency models describe what happens when a **single client** writes/reads from **multiple replicas**

Eventual Consistency

- Eventual consistency states that all replicas **eventually** converge when write operations stop
  - E.g., lazy replication using gossiping (cf. replication)
- Very weak form of consistency with no time bound, but **highly available** (i.e., always return a value, but could be stale)
- Works fine if a client always reads from **same** replica…
- …but gives "weird" results if client reads from **multiple replicas, due to**:
  - Client mobility
  - Replica failure

Mobile Users Use Case (Bayou System)

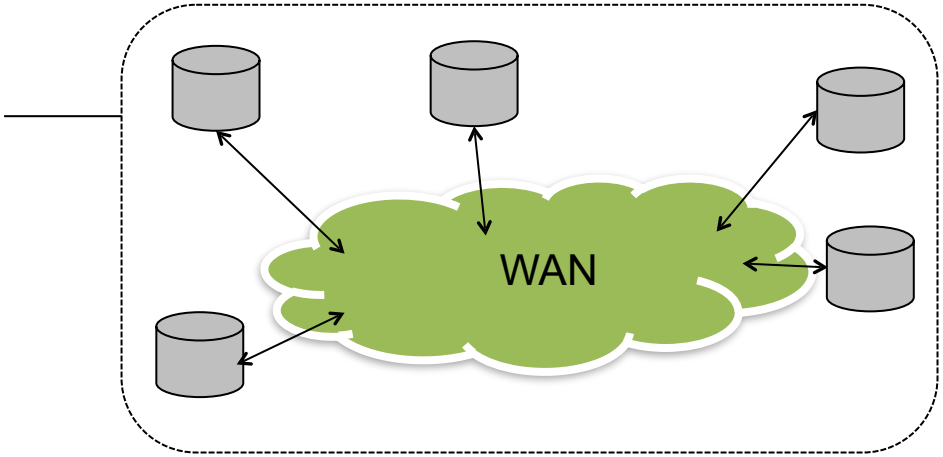


Replicas maintain
client-centric consistency

- System developed by Xerox PARC in mid 90's
- Designed to work under network partitions
- Replicas may take some time to converge

Mobile Users Use Case (Bayou System)

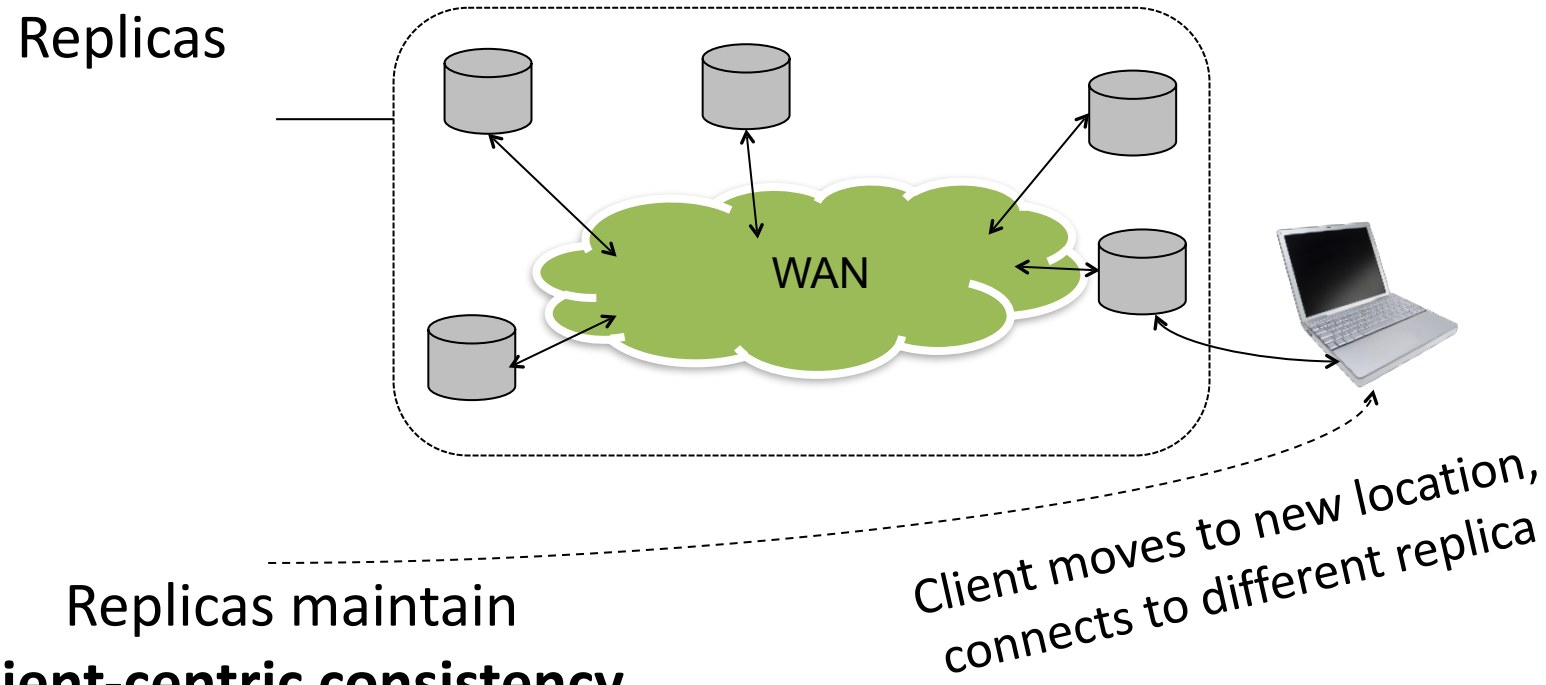
Replicas



Replicas maintain
client-centric consistency

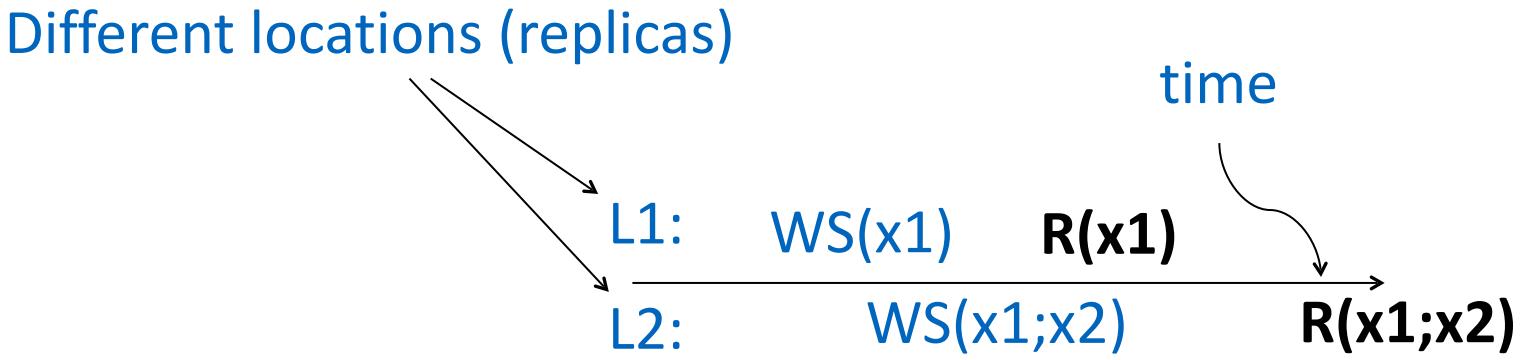
- System developed by Xerox PARC in mid 90's
- Designed to work under network partitions
- Replicas may take some time to converge

Mobile Users Use Case (Bayou System)



- System developed by Xerox PARC in mid 90's
- Designed to work under network partitions
- Replicas may take some time to converge

Notation



- **Reads** and **writes** by **one client** at **different locations**
- WS represents a series of write operations at L_i
- E.g., WS*(x1)* denotes that only *x1* has been applied; WS*(x1;x2)* denotes that *x1* and then later *x2* have been applied
- R(*x1*;*x2*) means read returns a value formed by *x1* followed by x2
- W(*x1*), R(*x1*) are write and read operations

Client-centric Consistency Models

- Read your writes consistency
- Monotonic reads consistency
- Writes follow reads consistency
- Monotonic writes consistency

Read-Your-Writes Consistency

The following consistency models always state:

A data store provides "***XYZ***" ***consistency*** if

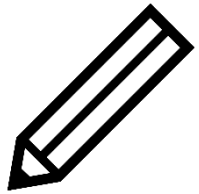
...

in the same process potentially issueing reads or writes **against different replicas.**

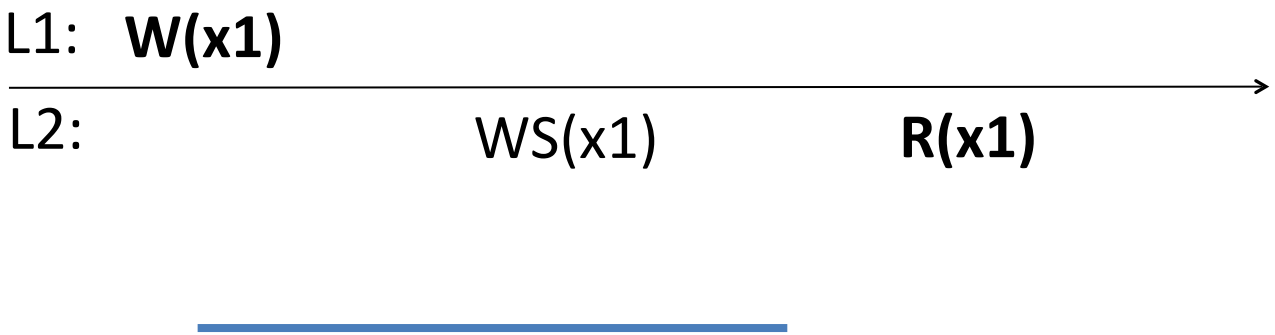
Read Your Writes Consistency

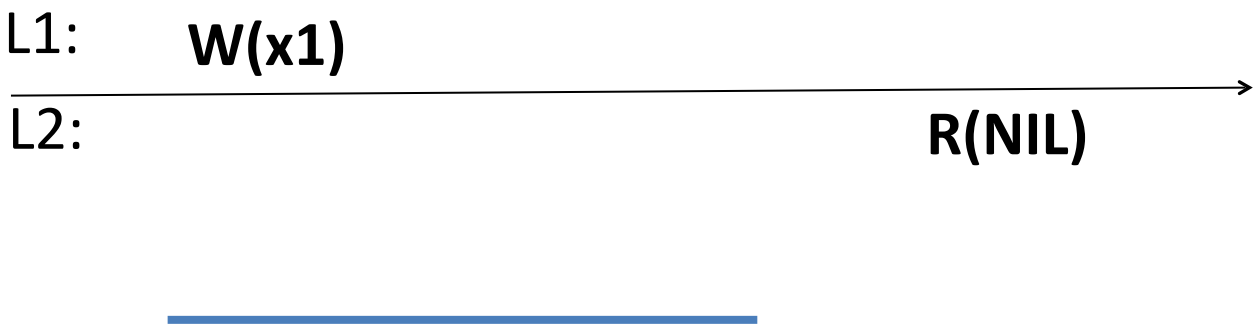
- Informally, … ***W*** … ***R*** … (by same process!)
- If a ***read*** follows a ***write*** then the effect of the write is included in the set of writes read by *R*
- **A write operation is always completed before a successive read operation by the same process**, no matter where the read operation takes place

Read Your Writes

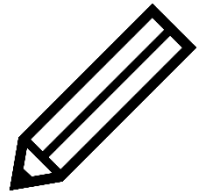


Example: Updating your password on a cluster

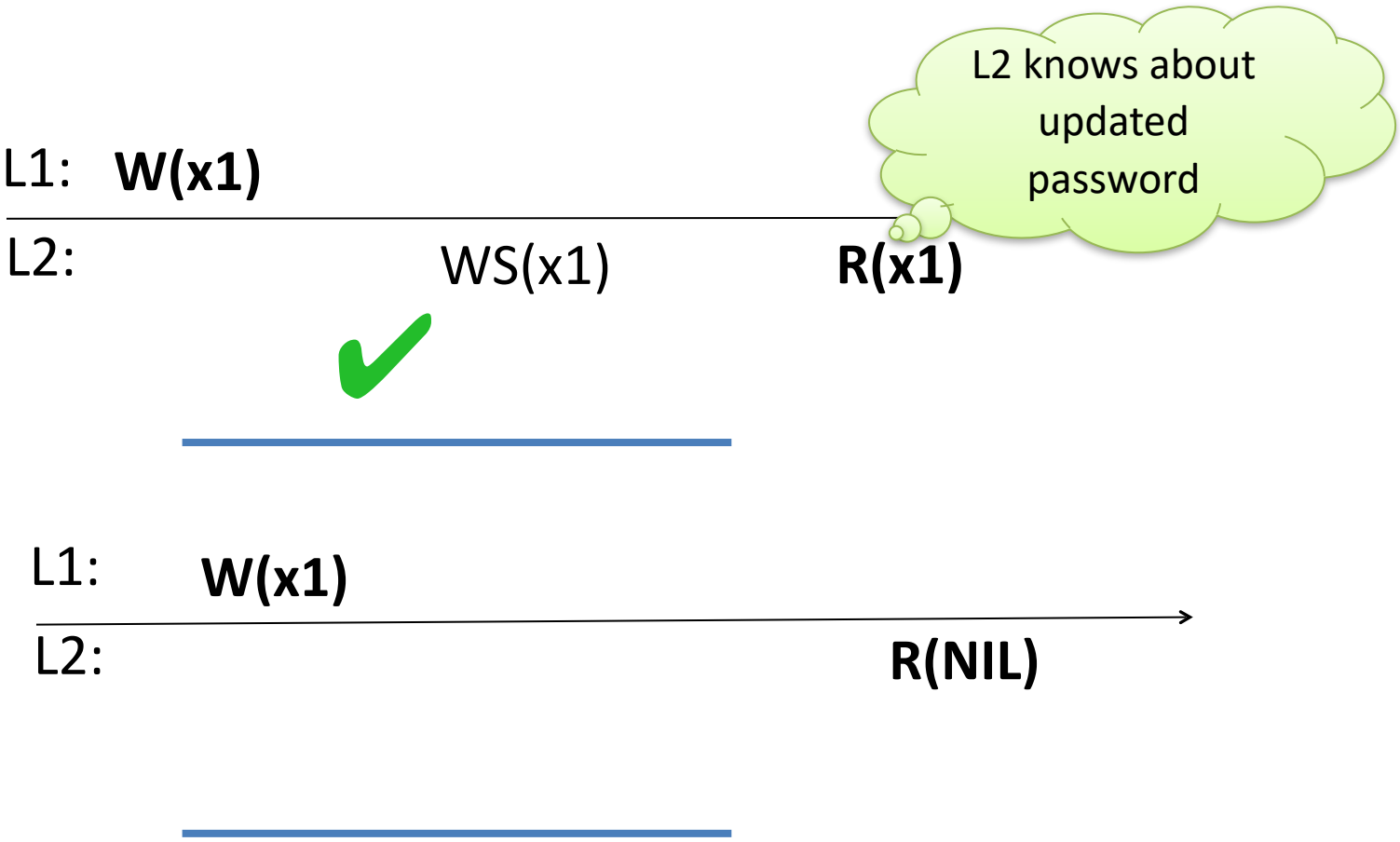




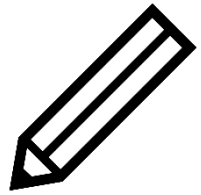
Read Your Writes



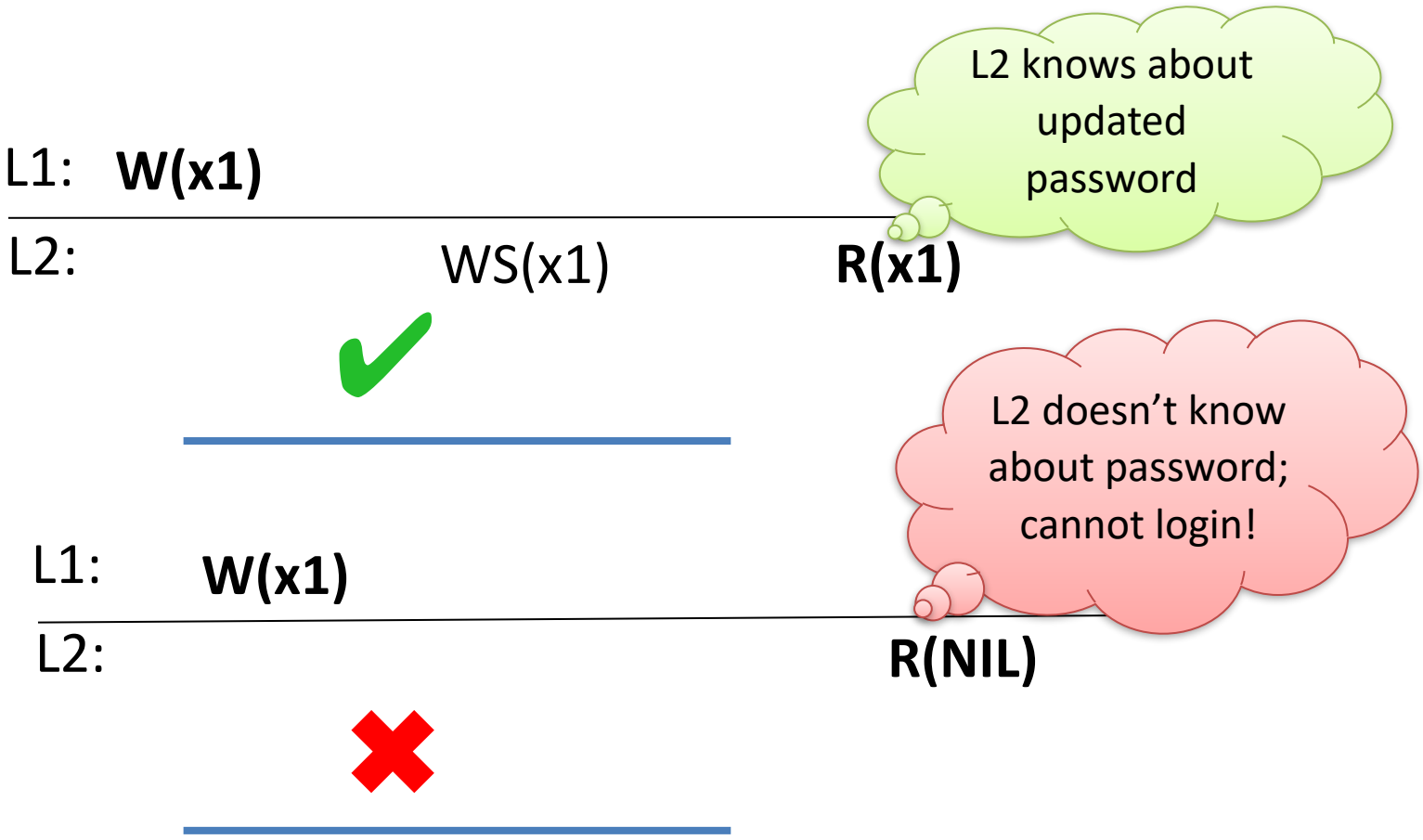
Example: Updating your password on a cluster



Read Your Writes



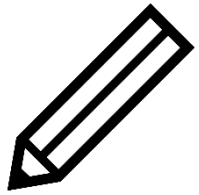
Example: Updating your password on a cluster



Monotonic-Reads Consistency

- … $\boldsymbol{WS_1(..)}$ **R** … $WS_2(..)$ **R** … (s.t. $WS_1(..) \subseteq WS_2(..)$)
- If a process reads a value formed by a set of operations, WS_1, any successive read operation of that process always returns a value formed from a superset of WS_2
- A process always sees **more recent data** (but not necessarily fresh!)
- If a process reads again from another replica that replica must have already received the relevant older operations

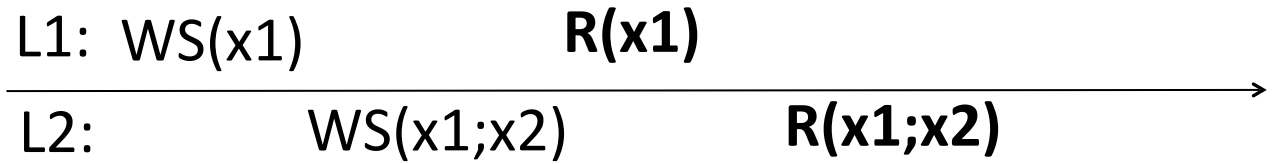
Example: Monotonic-Reads

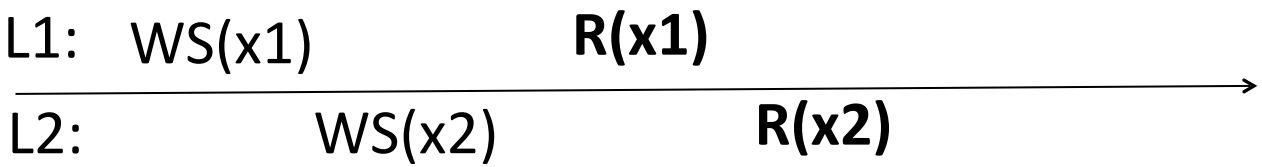


Example: E-mail client

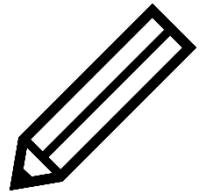
Each read returns list of emails

Each write adds one e-mail





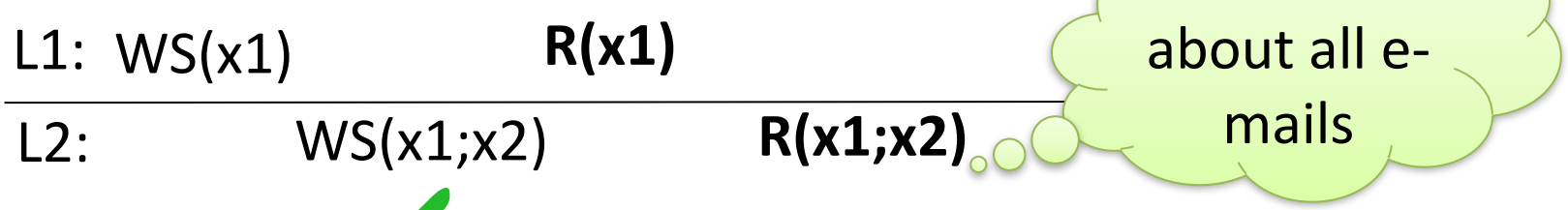
Example: Monotonic-Reads

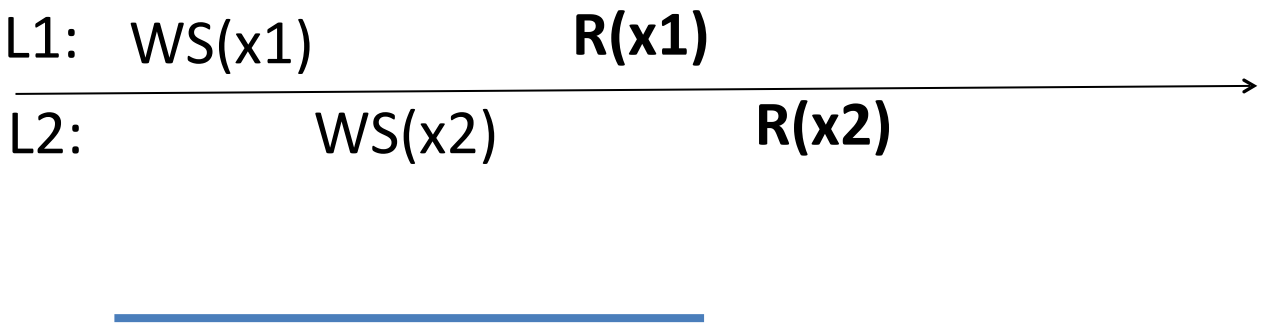


Example: E-mail client

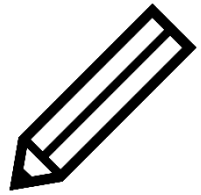
Each read returns list of emails

Each write adds one e-mail





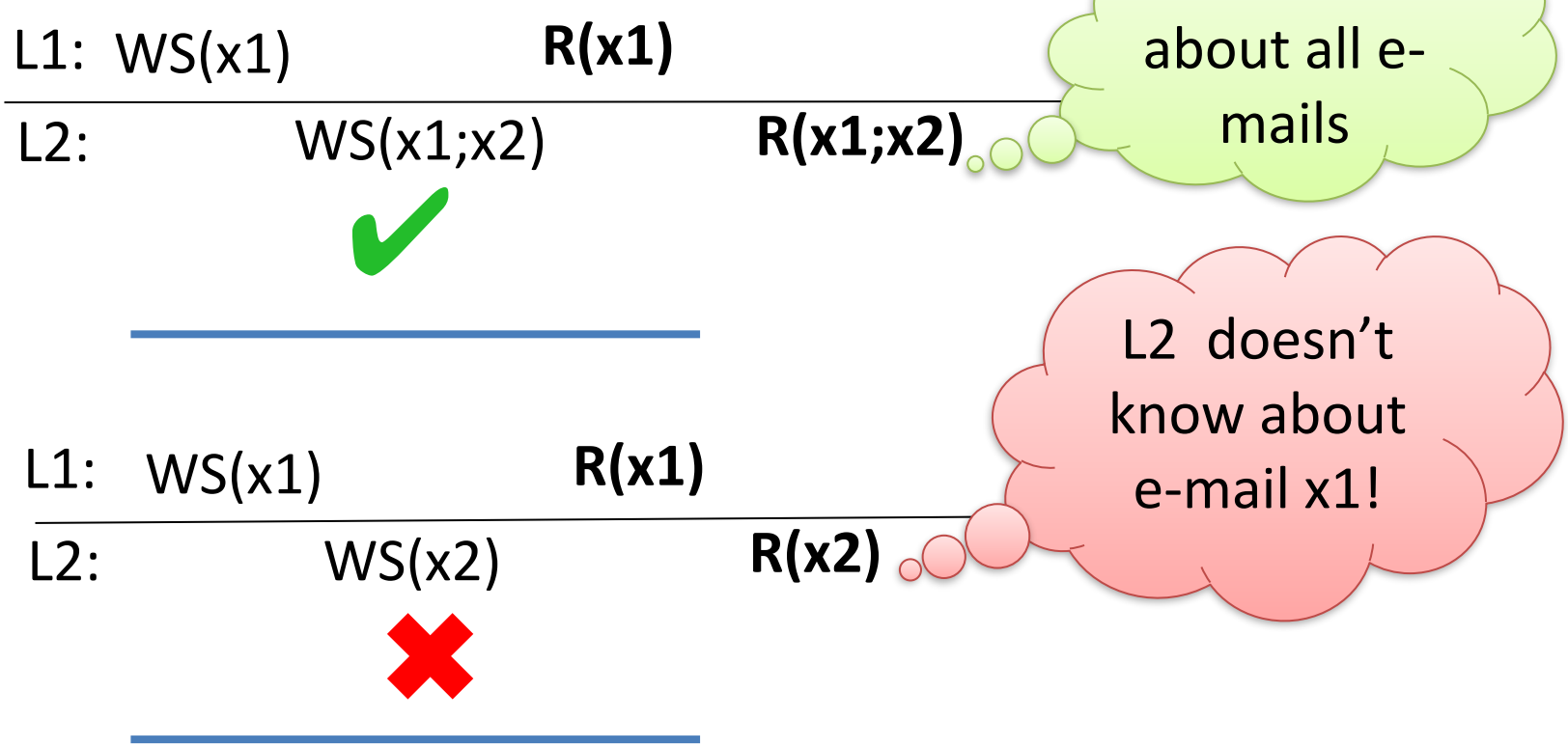
Example: Monotonic-Reads



Example: E-mail client

Each read returns list of emails

Each write adds one e-mail



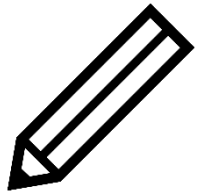
Writes Follow Reads Consistency

- *… WS(X1, …, Xn)* ***R1*** *… WS(X1, …, Xn)* ***W2*** *…*

- If a read *R1* precedes a write *W2* then at any replica, if *W2* is written, it is preceded by the write set read by *R1*

- Any successive write operation by a process will be performed on a state that is **up to date** with the value **most recently read** by that process

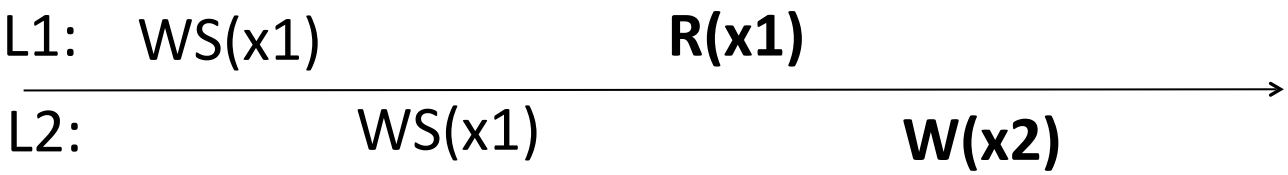
Writes Follow Reads

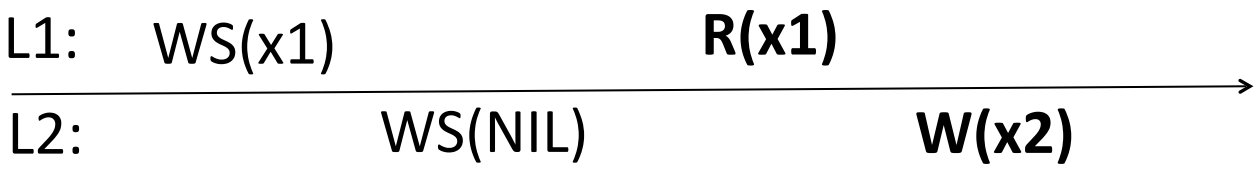


Example: Facebook Wall

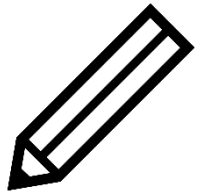
Reading a post

Replying to a post





Writes Follow Reads

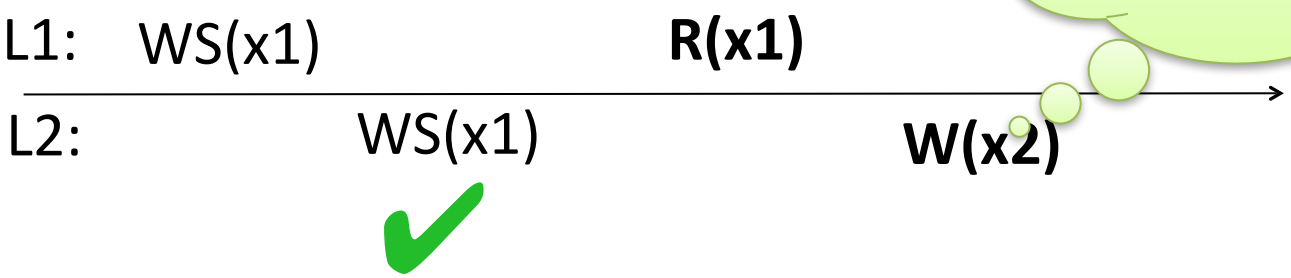


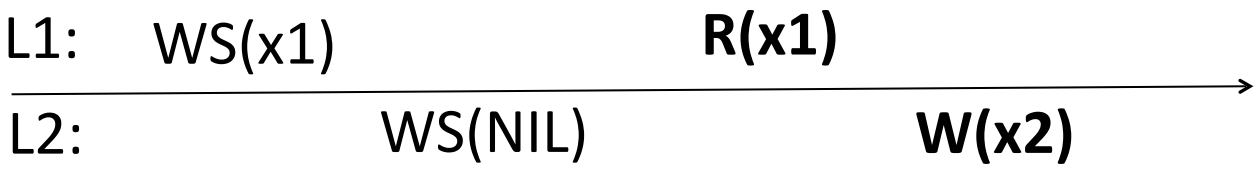
Example: Facebook Wall

Reading a post

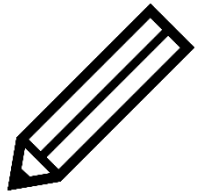
Replying to a post

L2 knows post (x1), writes reply (x2) to known post





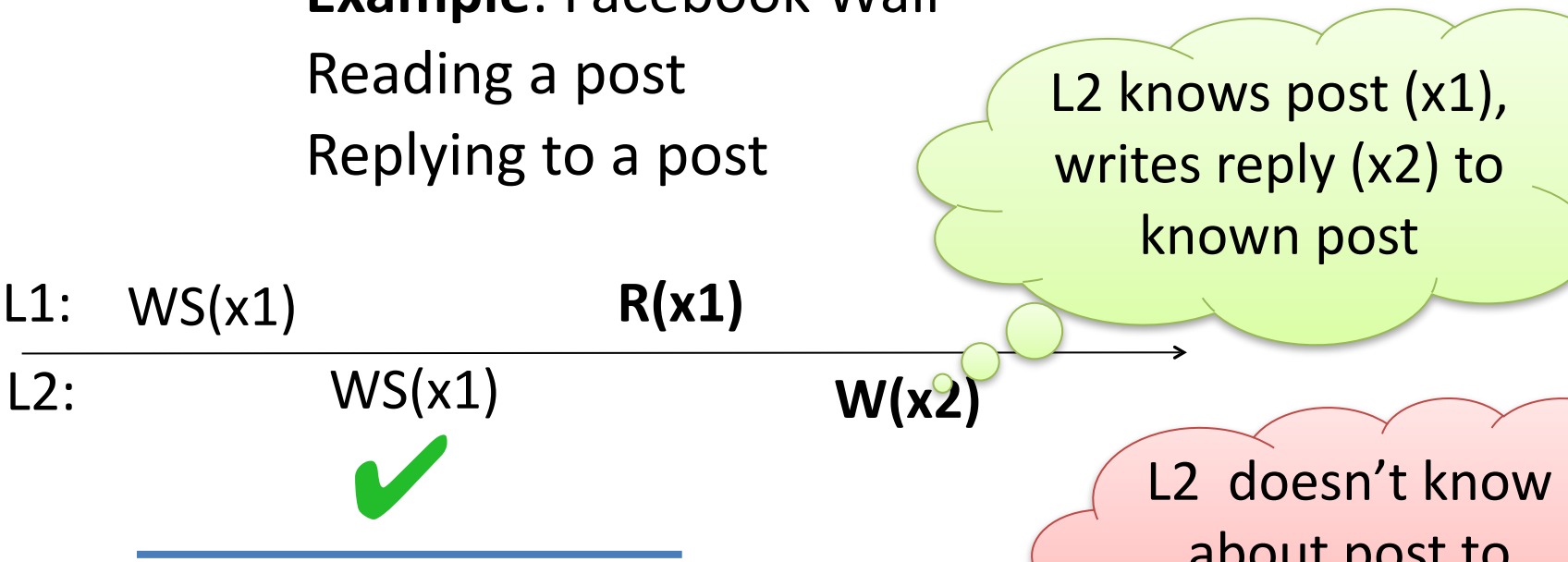
Writes Follow Reads



Example: Facebook Wall

Reading a post

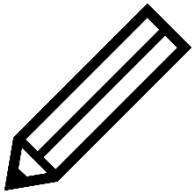
Replying to a post



Monotonic-Writes Consistency

- Informally, … ***W1*** … ***W2*** …
- If a write *W1* precedes a write *W2* then on any replica, *W1* must precede *W2*
- In other words**, writes by the same process** are performed in the same order at every replica
- Resembles FIFO consistency model

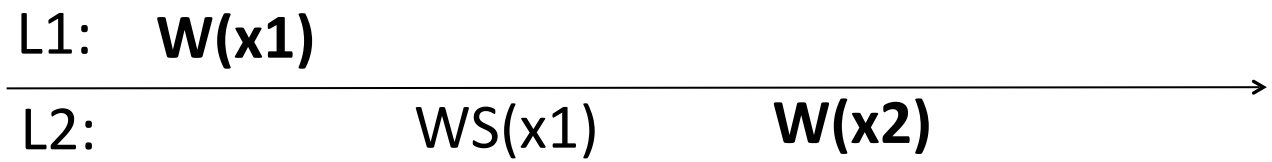
Monotonic Writes

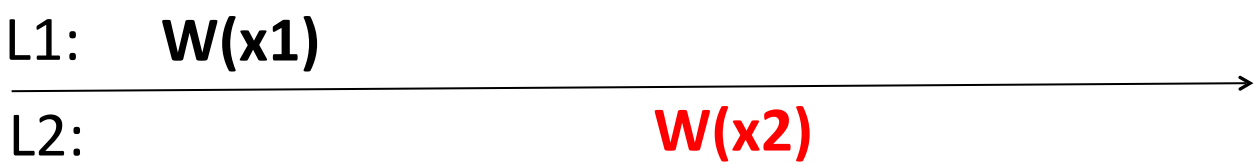


Example: Replicating software code

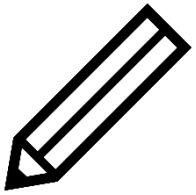
W(x1) adds a new method to a program

W(x2) calls new method





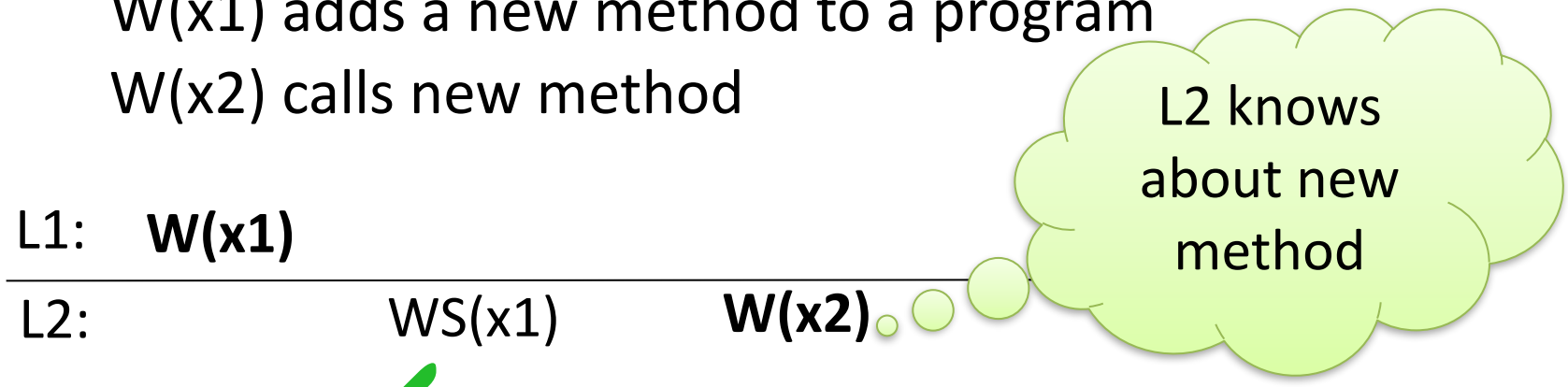
Monotonic Writes



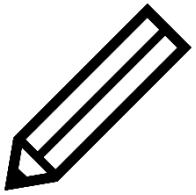
Example: Replicating software code

W(x1) adds a new method to a program

W(x2) calls new method



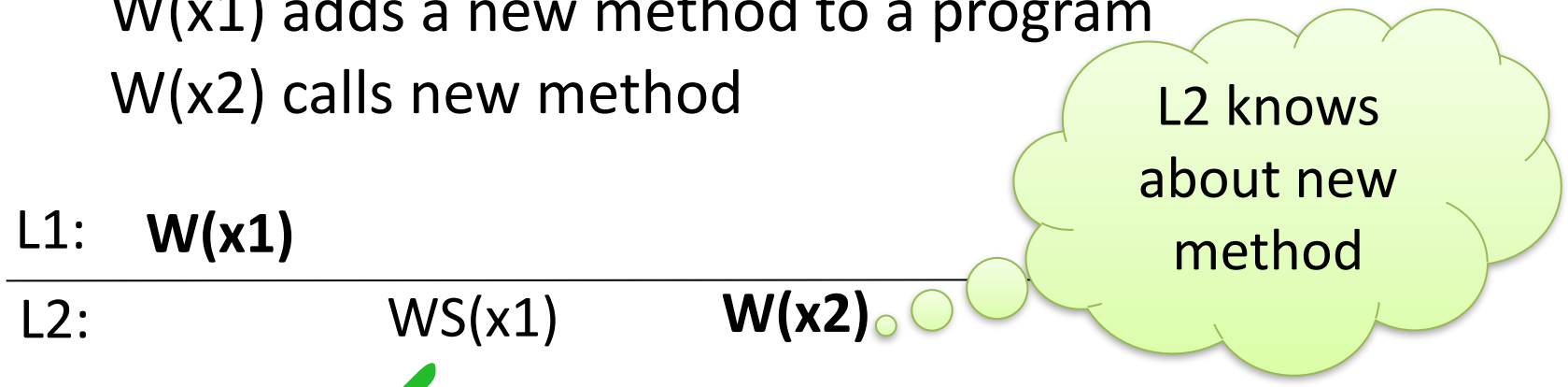
Monotonic Writes

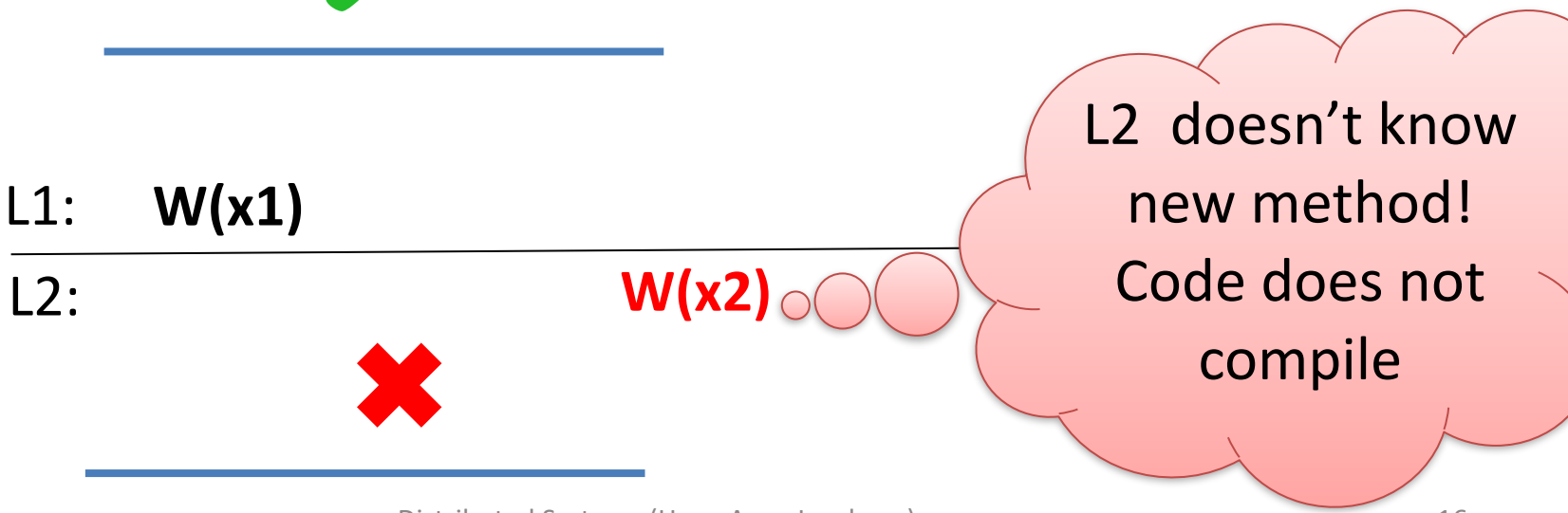


Example: Replicating software code

W(x1) adds a new method to a program

W(x2) calls new method





Monotonic writes

- Informally, writes are applied in the order in which they arrive
- For example,
  - x = 100
  - W1 is x = x + x * 10 %
  - W2 is x = x + 100
- W1, W2
    - 210
- W2, W1
    - 220

Summary: Client-centric consistency

- Eventual consistency can be used when:
  - No or few concurrent writes on the same key occur
  - Used in Dynamo, Cassandra, Riak, et al.
- Eventual consistency provides high availability and scalability
- Client-centric consistency dictates how reads and writes should look from the client's perspective
  - Read your writes
  - Monotonic-reads
  - Monotonic-writes
  - Writes follow reads

Self-study Questions

- Write some concrete operation sequences that illustrate write-write and read-write conflicts in a data store.
- Create some concurrent read and write sequences that conform to different client-centric consistency models.
- Is there a correspondence among data-centric and client-centric consistency models? Could one be used to emulate the other, argue for or against.
- How do client-centric consistency models relate to one another, if at all?
- Find sample applications for each consistency model.
- How can client-centric consistency models be implemented?