Distributed Systems

Fundamentals – Shared Store

Fabienne Boyer
Reprise du cours d'Olivier Gruber

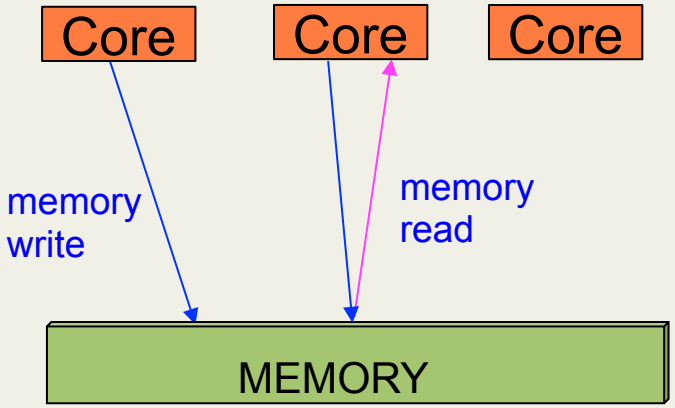
Université Joseph Fourier

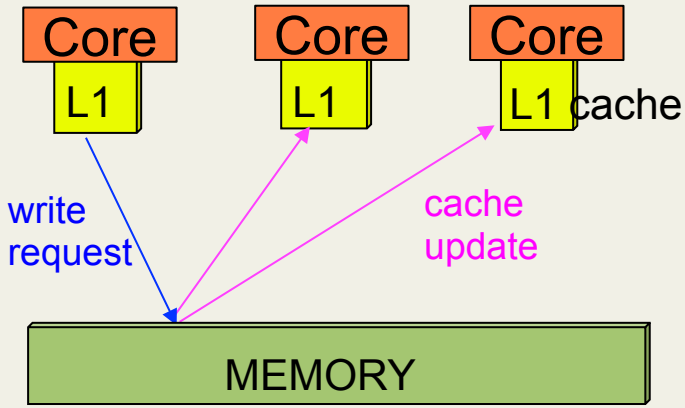
Projet ERODS (LIG)

Message Fundamentals

- Today's Lecture
  - The problem:
    - Sharing data between multiple processes (read and write access)
    - When accessing something means acquiring a copy through messages
    - When updating a local copy requires sending notification messages
  - Discussing consistency
    - How are the different processes seeing each other reads and writes?

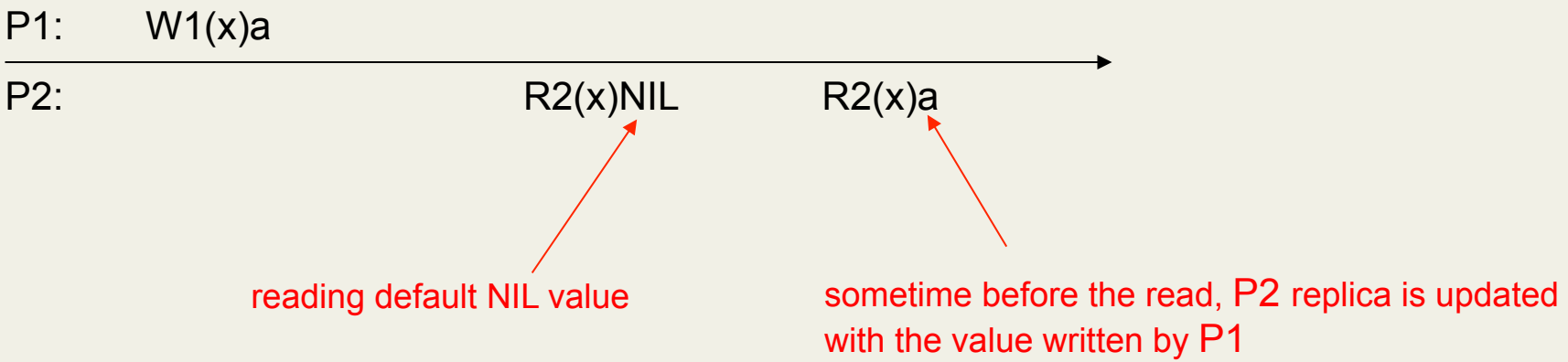
- **Multi-cores – memory** Illustration
- Could be **JVM – object store**





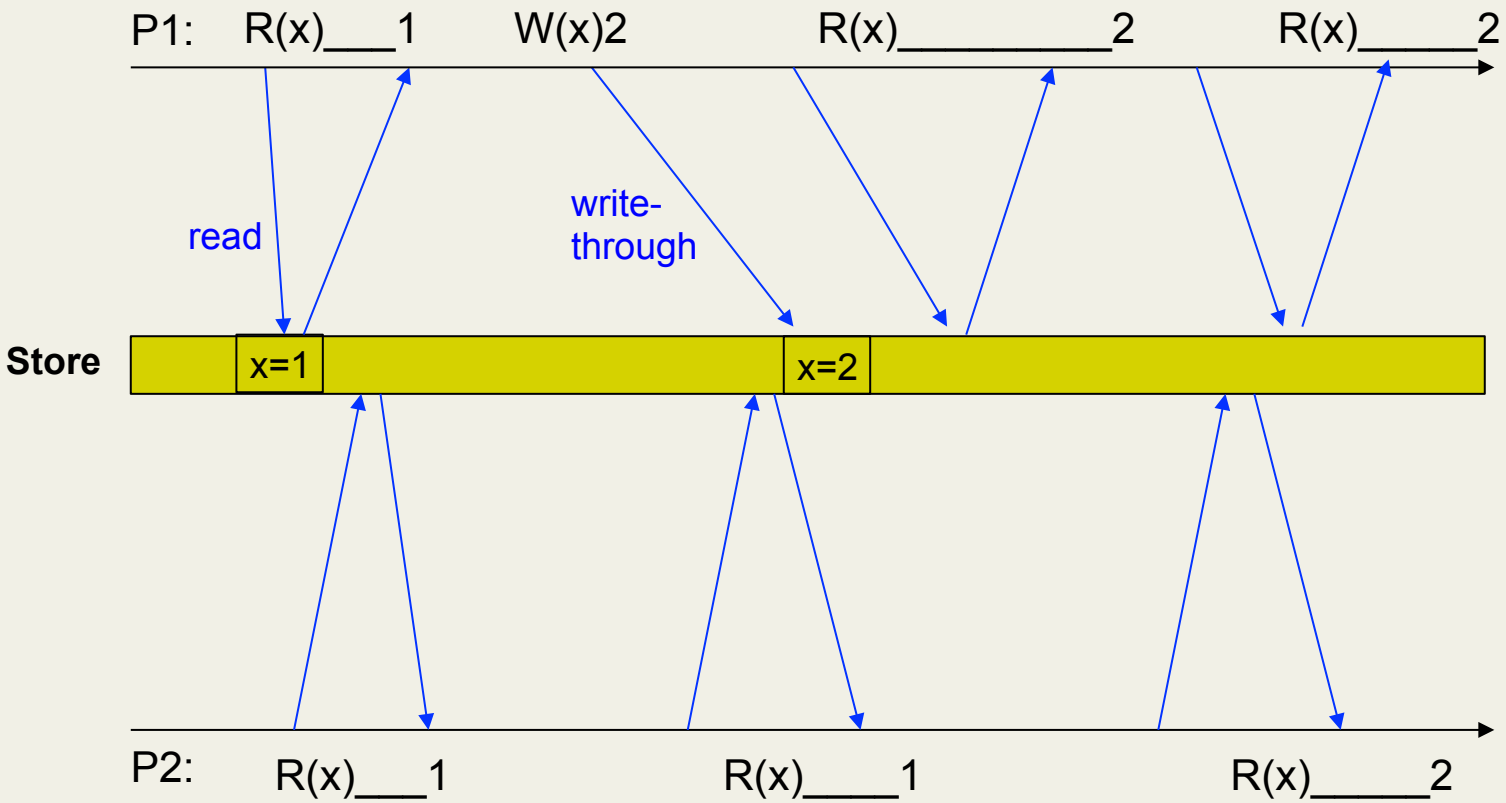
Useful Notations

- Notations
  - Read operation at process ***Pi*** on data ***x*** returning the value ***a***
    - Ri(x)a
  - Write operation at process ***Pi*** on data ***x*** writing the value ***b***
    - Wi(x)b
  - Time flows from left to write
  - All data items are initialized to *NIL*



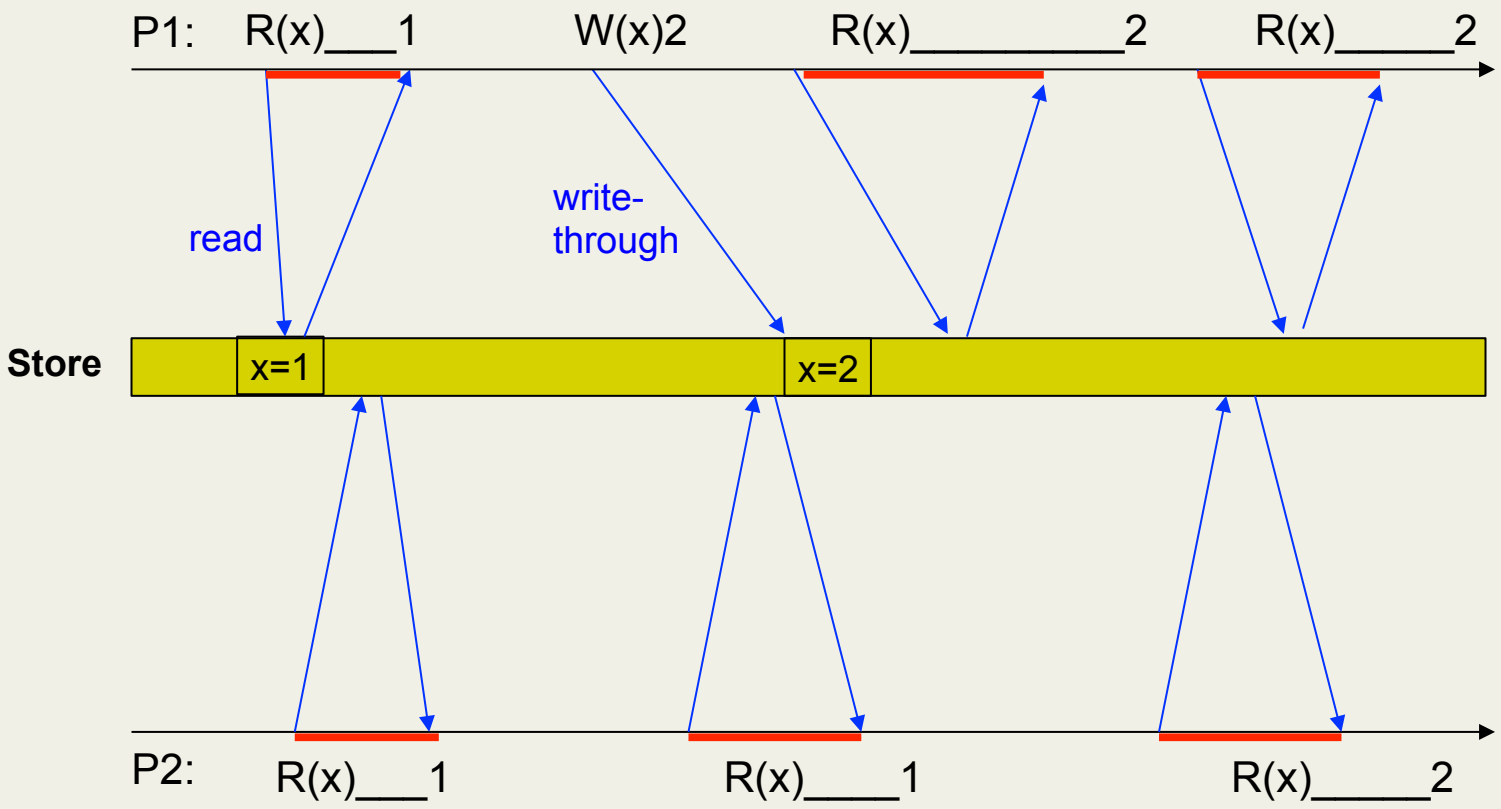
Shared Store

- Processes
  - We have N processes
  - We have a shared data store with data items



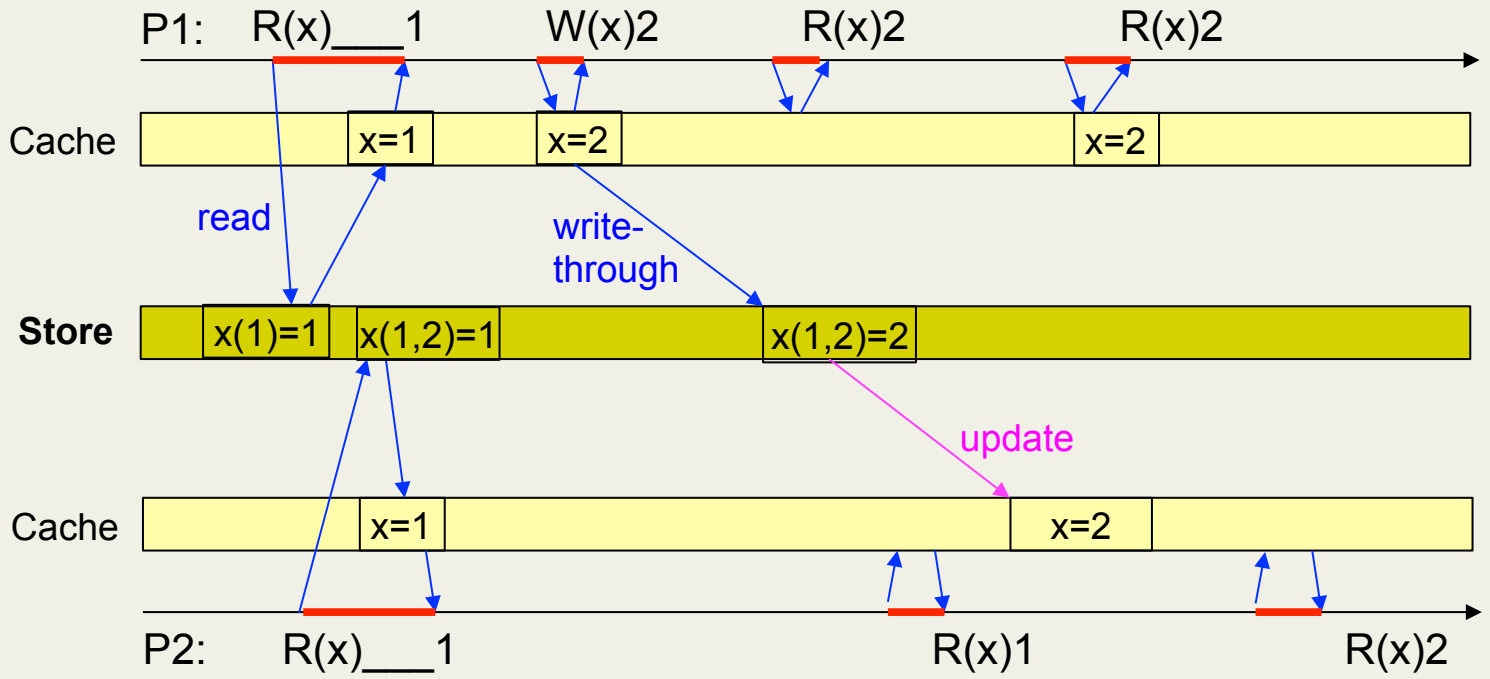
Shared Store

- Discussion
  - Like a normal shared memory, requires the use of synchronization
  - Poor performance (latencies)



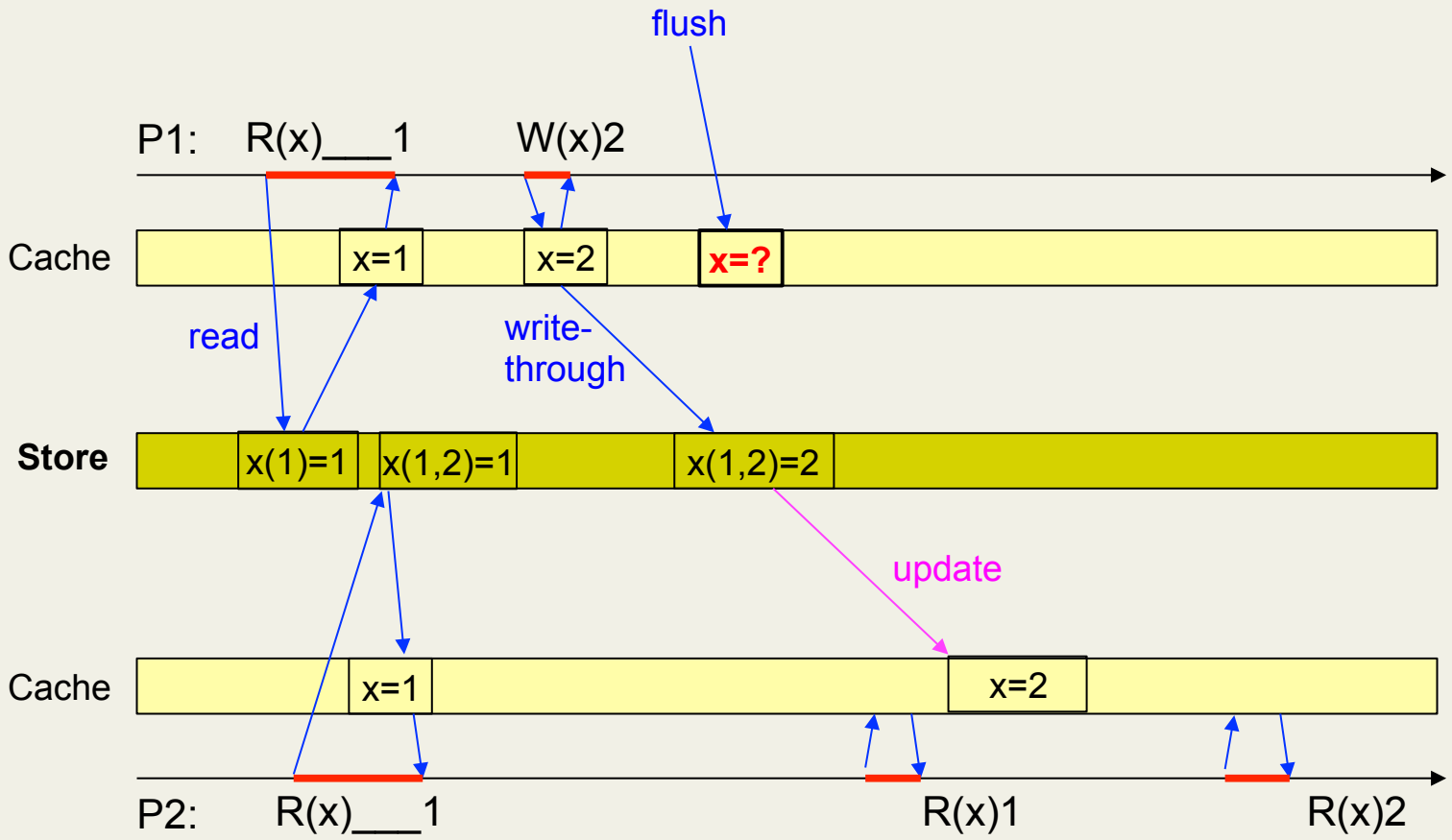
Shared Store

- Introducing local caches
  - One local cache per process
  - Shorter latencies, but we need a consistency protocol
    - Store must remember who has a cached copy of each data item
    - Must be able to callback caches to install updates



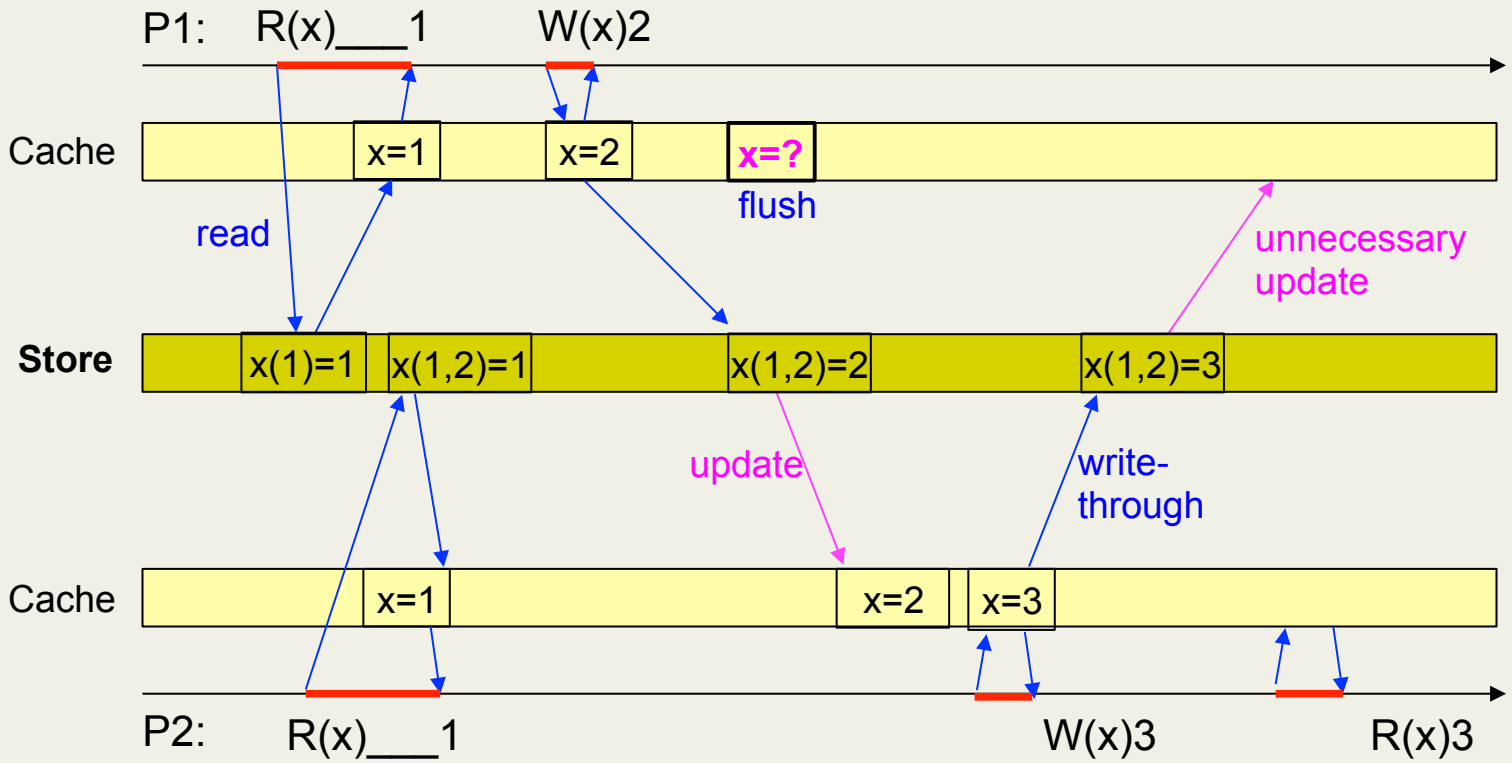
Shared Store

- Introducing cache flushes
  - Caches may overflow, we need to flush some cached data items



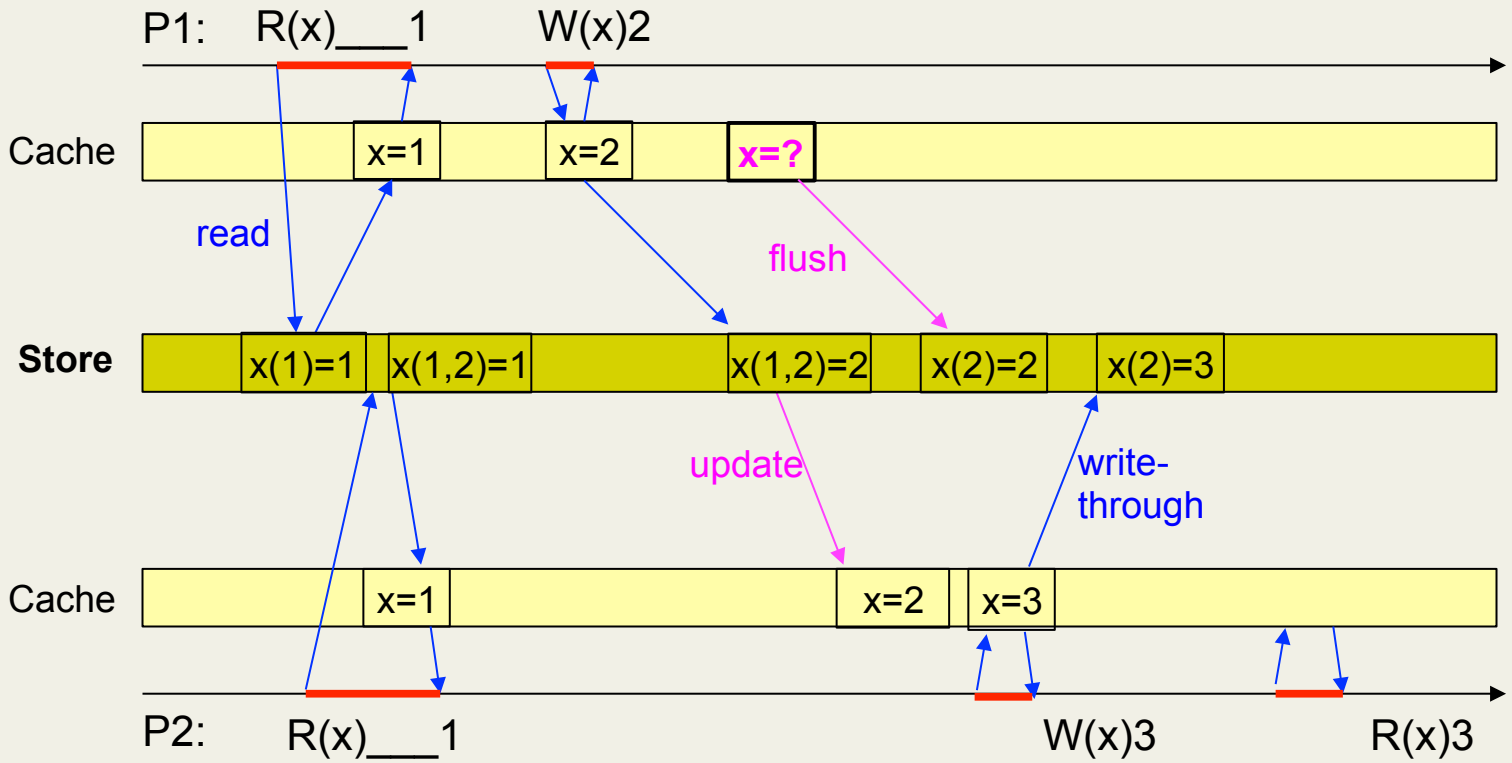
Shared Store

- Cache Design
  - How do we maintain the store copy lists?
  - So to avoid unnecessary update messages



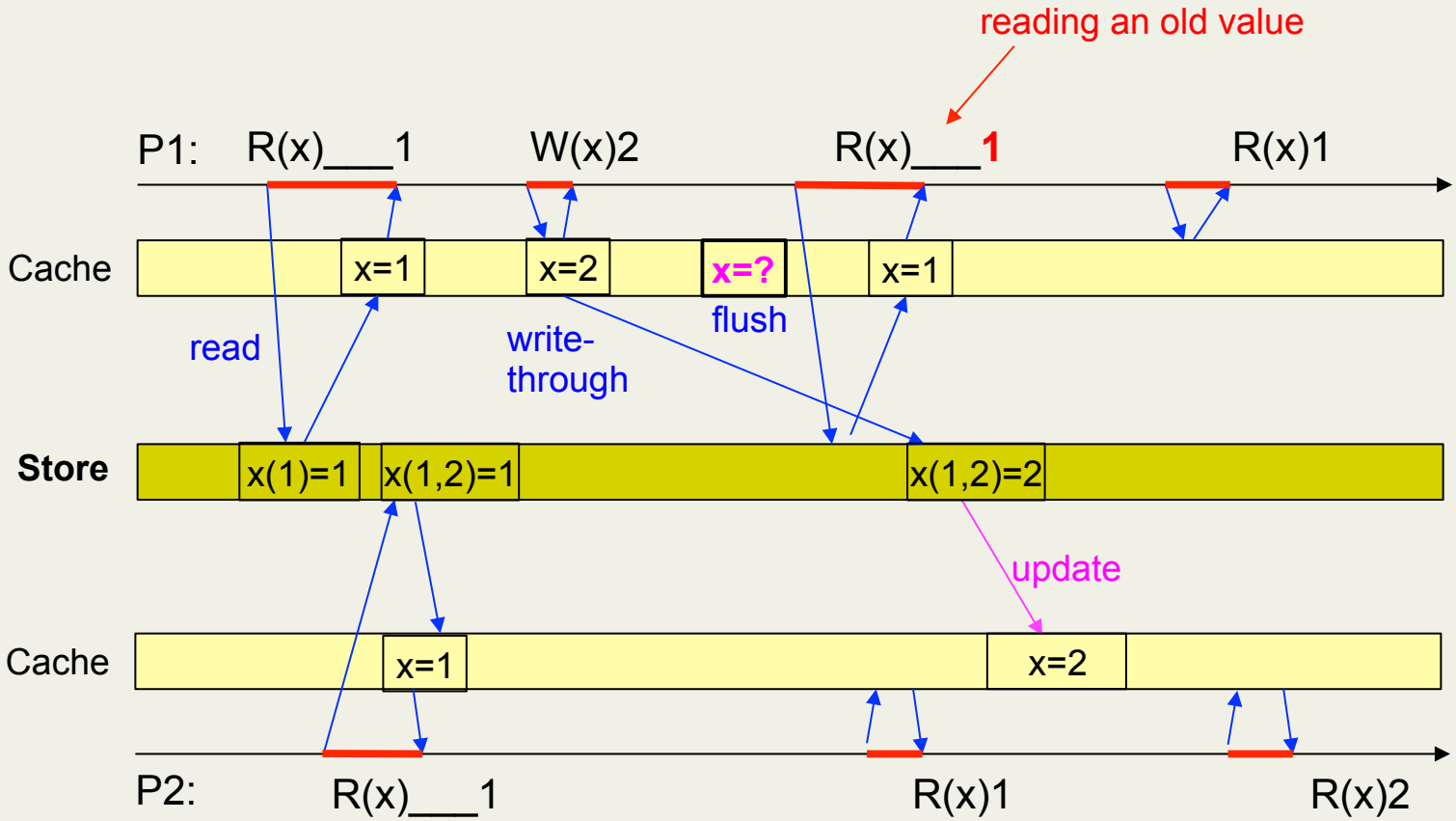
Shared Store

- Cache Design
  - Maintaining the store copy lists
    - Background messages
    - Time-To-Live caches



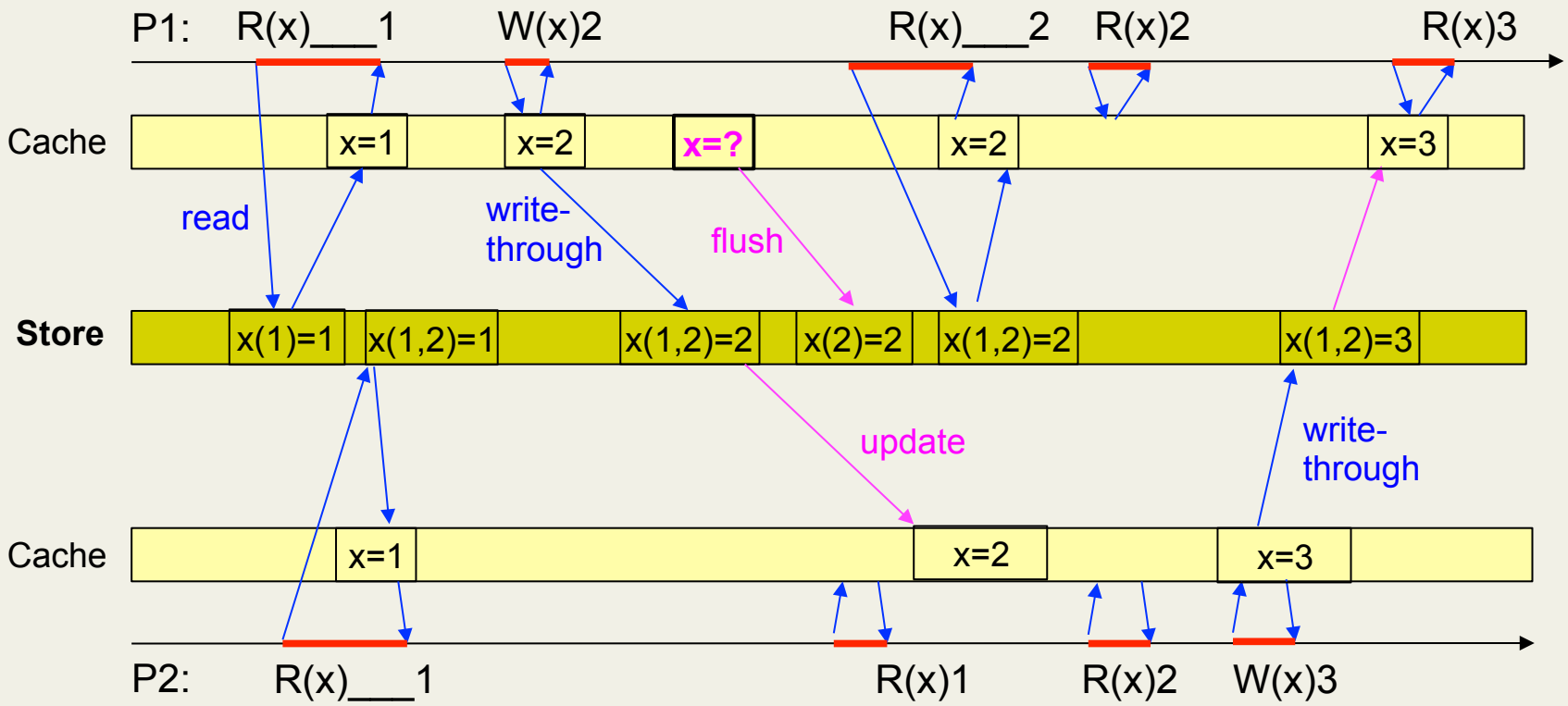
Shared Store

- Cache Design
  - Use **FIFO** communication channels!
    - With TCP/IP, requires to keep sockets open
    - With UDP, you need to implement FIFO/lossless

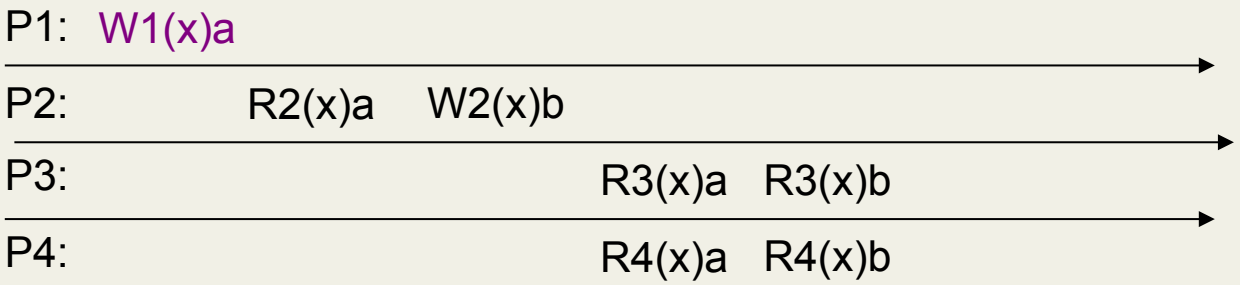


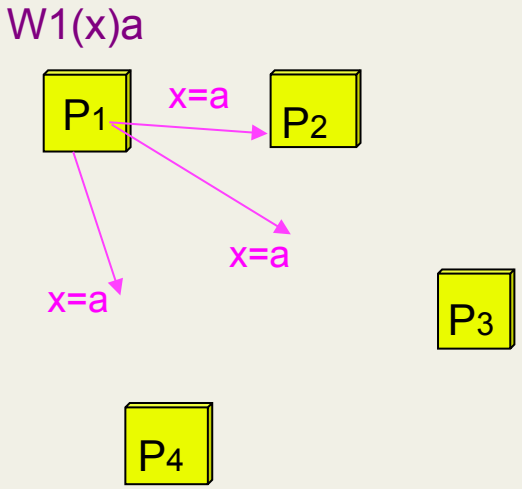
Shared Store

- Overview
  - Shared data store
  - Cache with flush and consistency protocol over FIFO channels



The problem





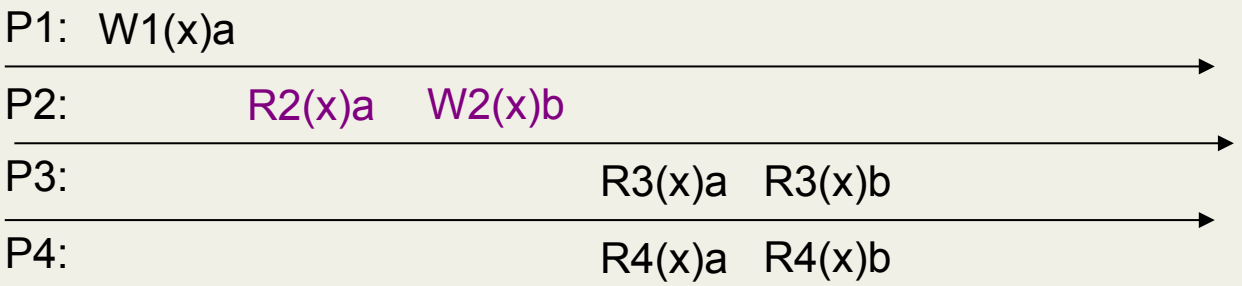
P1
x=a;

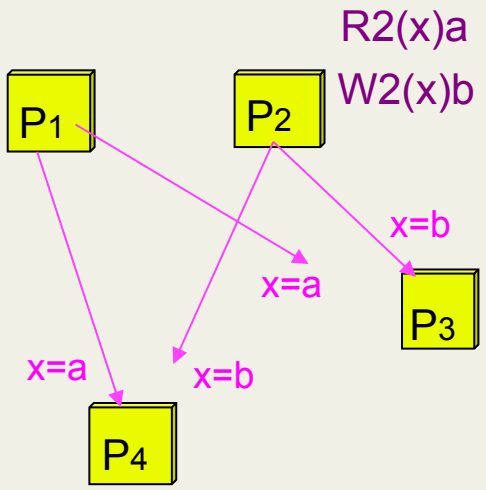
P2
if (x==a)
  x=b;

P3
switch(x)
case a:
  …
case b:
  ...

P4
switch(x)
case a:
  …
case b:
  ...

The problem



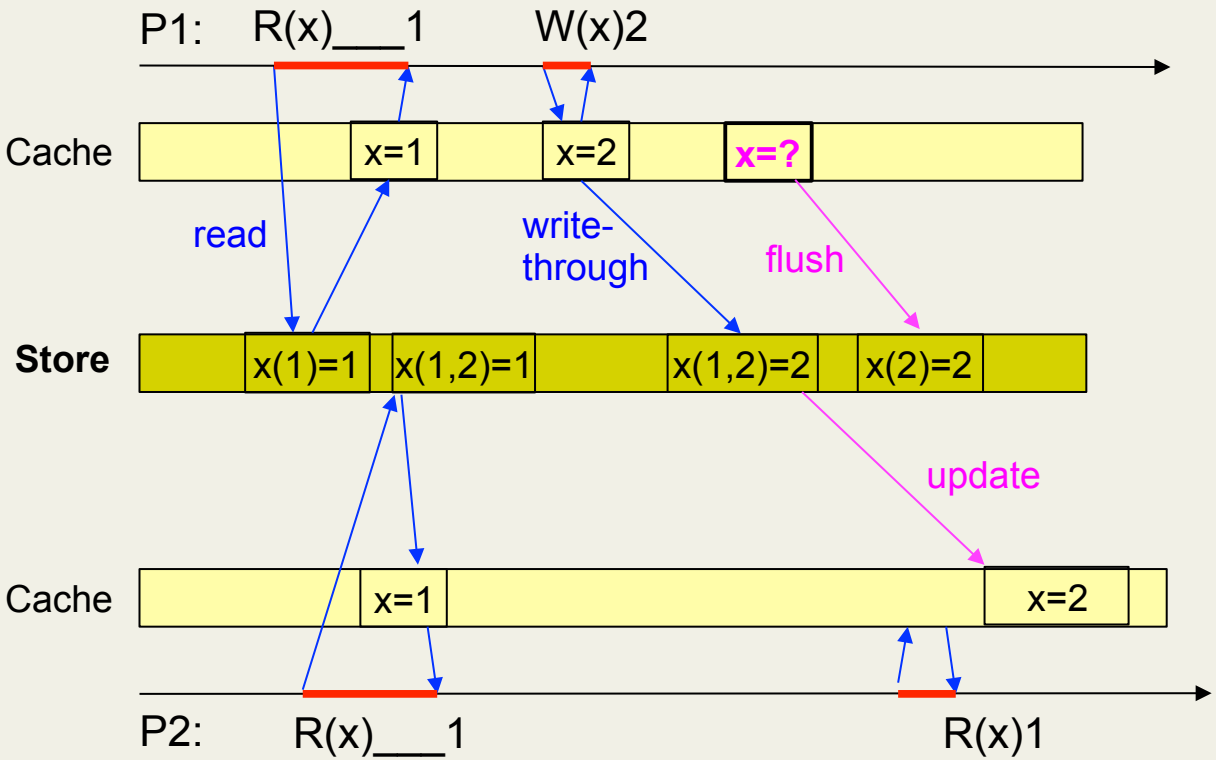


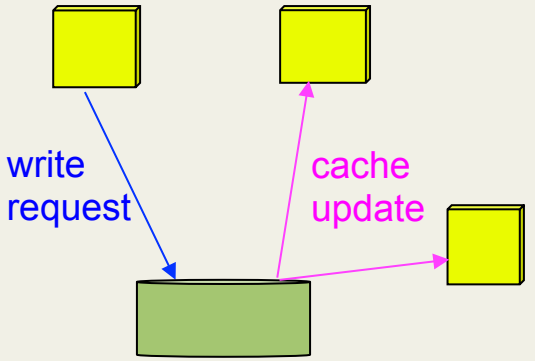
P1	**P2**	**P3**	**P4**
x=a;	if (x==a)	switch(x)	switch(x)
	x=b;	case a:	case a:
		…	…
		case b:	case b:
		...	...

write
request

Consistency Models

- Consistency definition
  - Essentially a contract between processes and a data store
- Different models are possible
  - ***How would we describe the behavior of our shared store?***

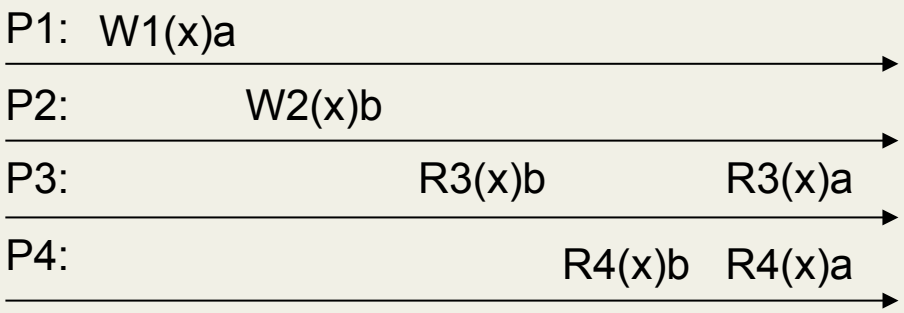




Sequential consistency

- Defined by Lamport (1979)
  - Memory works as expected with multiple processes

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.

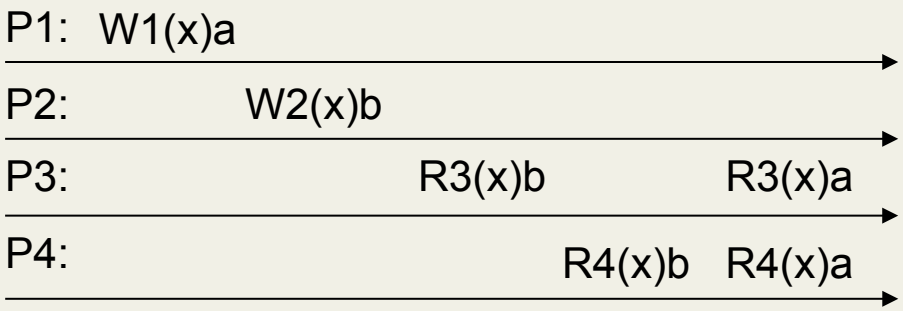


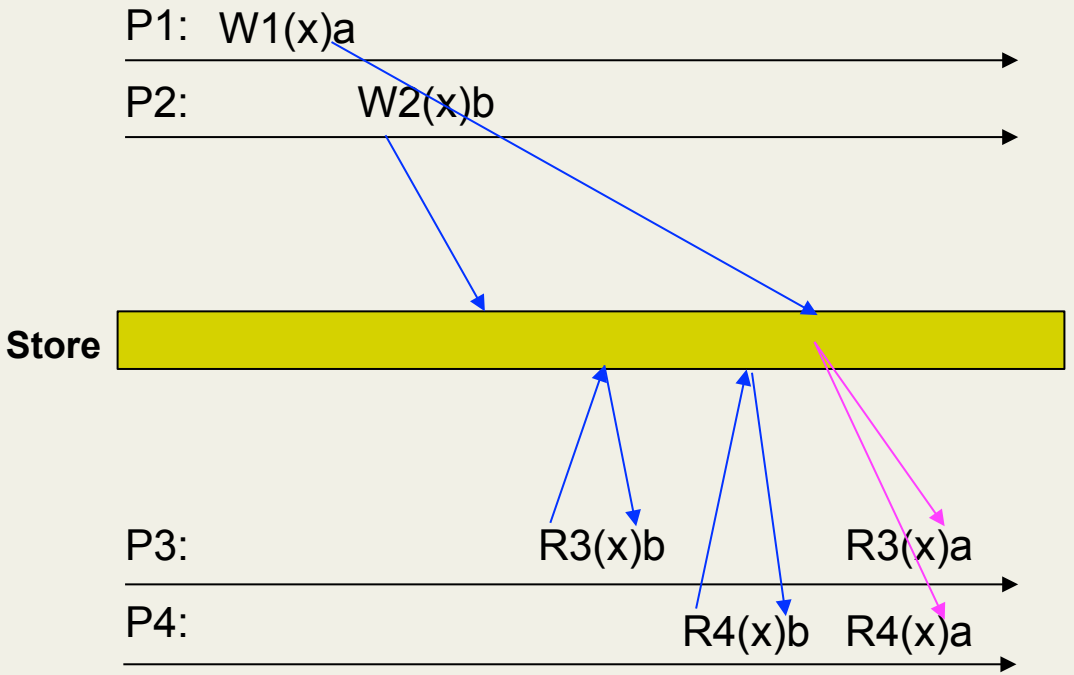
sequentially consistent

possible equivalent sequential order:

W2(x)b **R**3(x)b **R**4(x)b **W**1(x)a **R**3(x)a **R**4(x)a

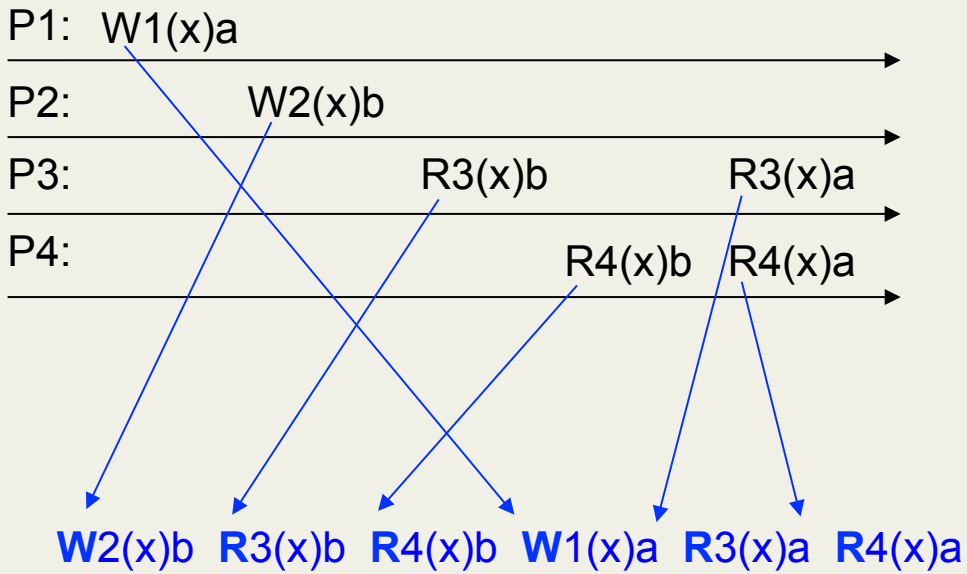
Sequential Consistency





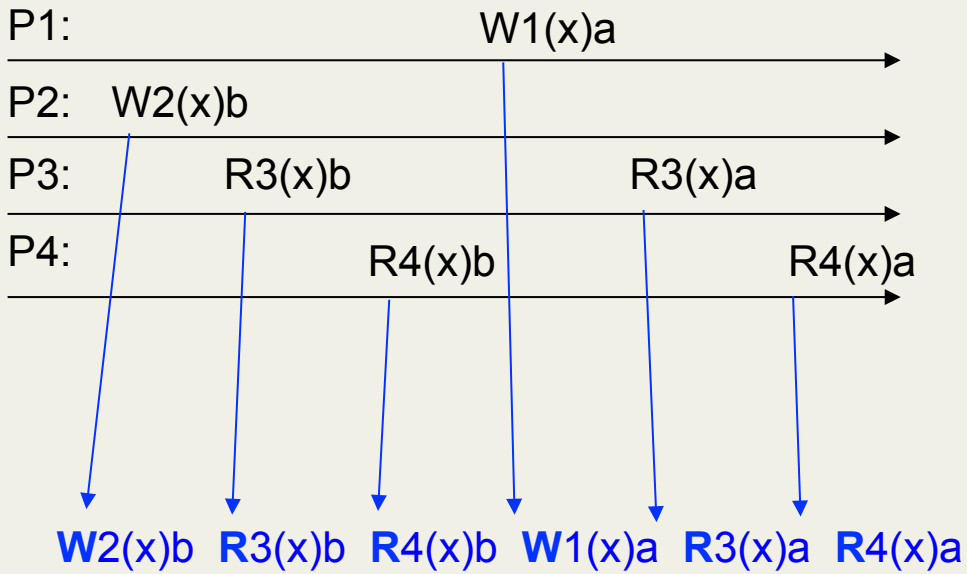
Sequential Consistency

Different possible equivalent sequential orders,
as if executed by a single processor,
on the same memory content



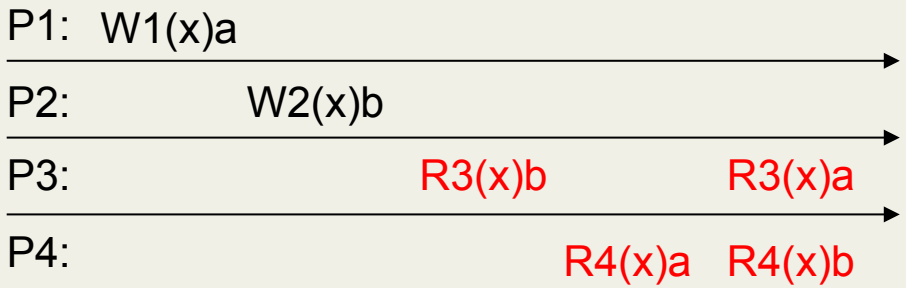
Sequential Consistency

Different possible equivalent sequential orders,
as if executed by a single processor,
on the same memory content



Sequential Consistency

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.



not sequentially consistent

no possible equivalent sequential order:

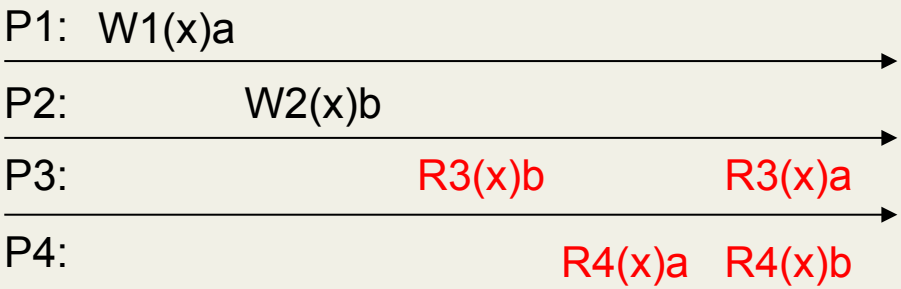
W2(x)b **R**3(x)b **W**1(x)a **R**3(x)a **R**4(x)a **R**4(x)b

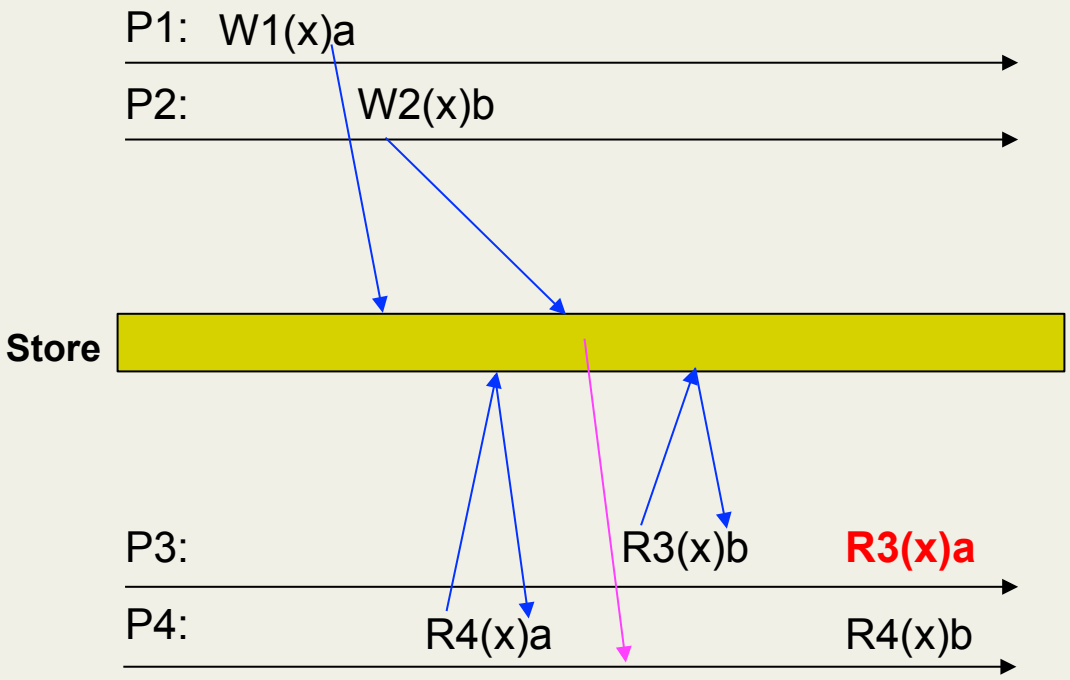
can't read the value b from x

W1(x)a **R**3(x)a **R**4(x)a **W**2(x)b **R3(x)b** **R**4(x)b

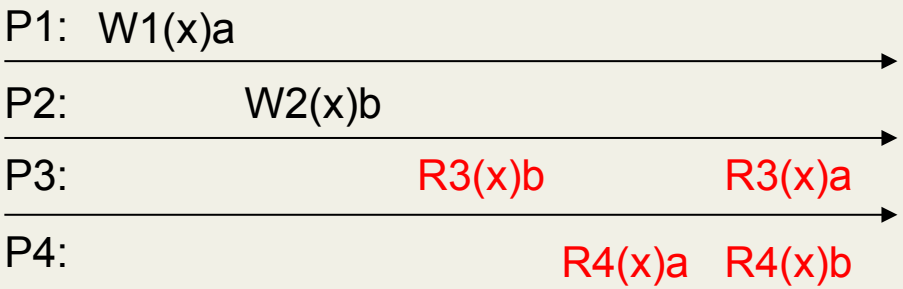
violate P3 local sequential order

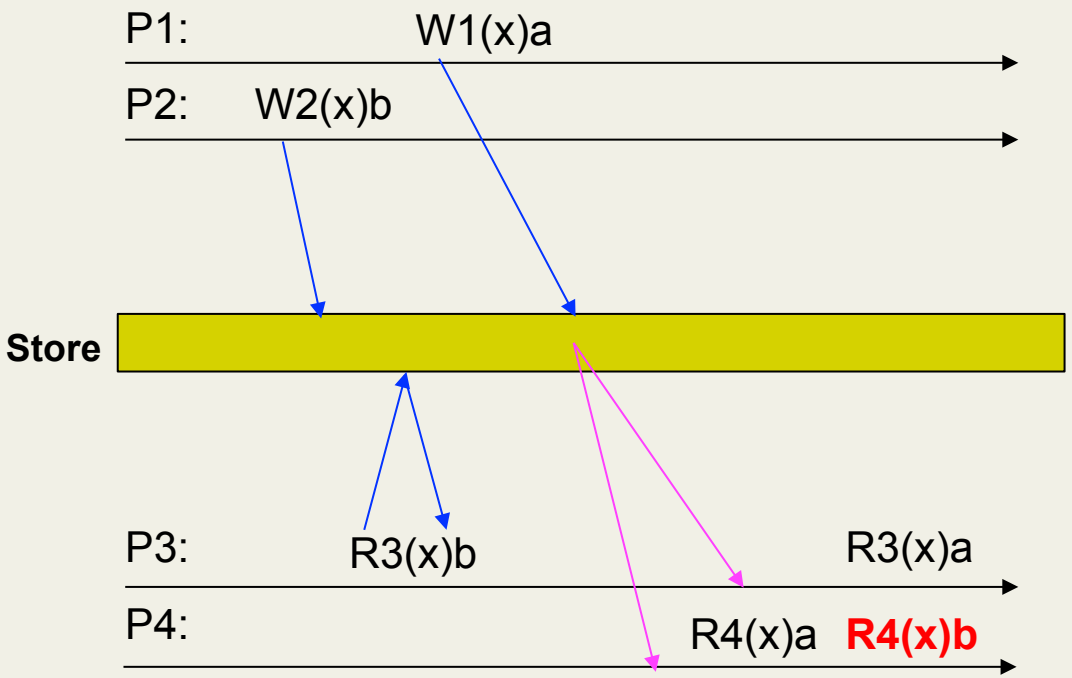
Sequential Consistency





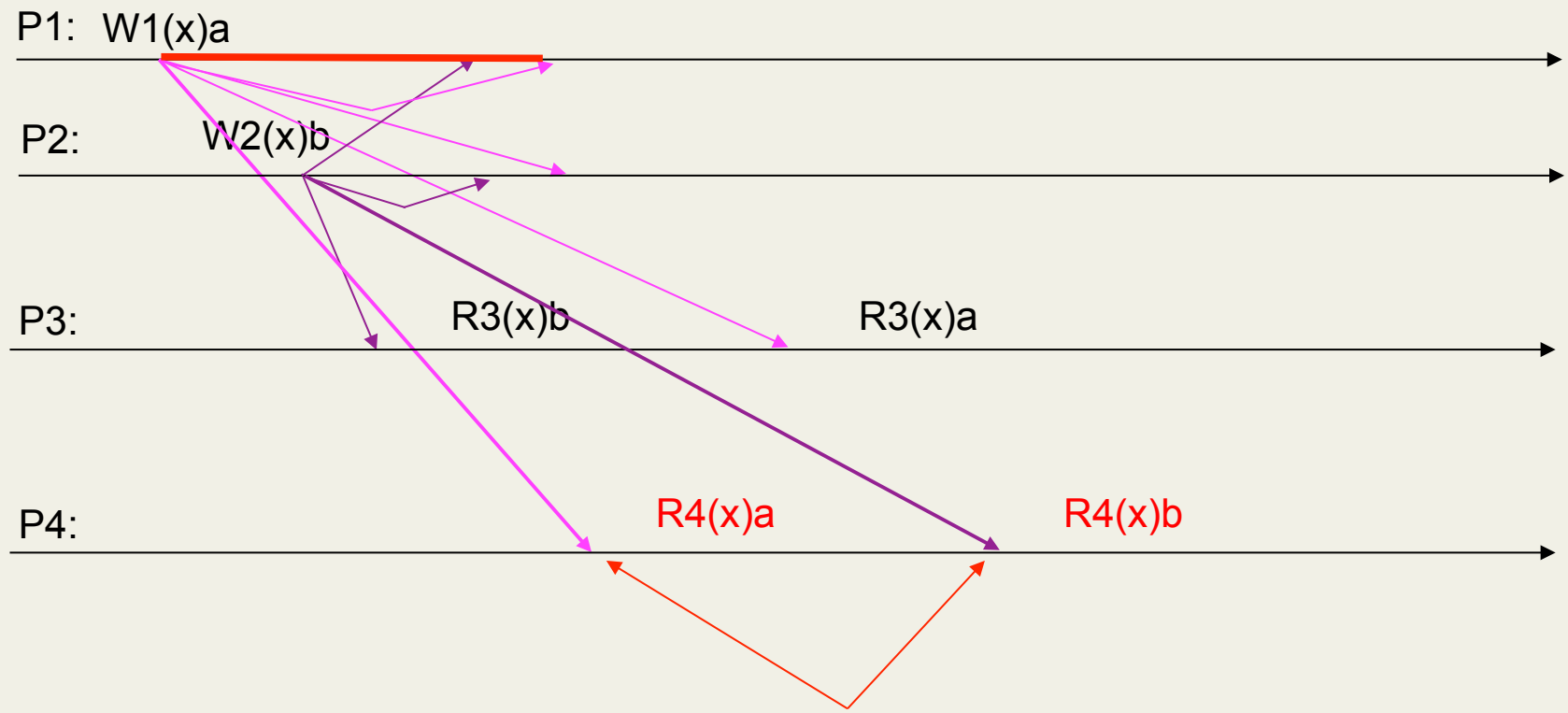
Sequential Consistency





Sequential Consistency

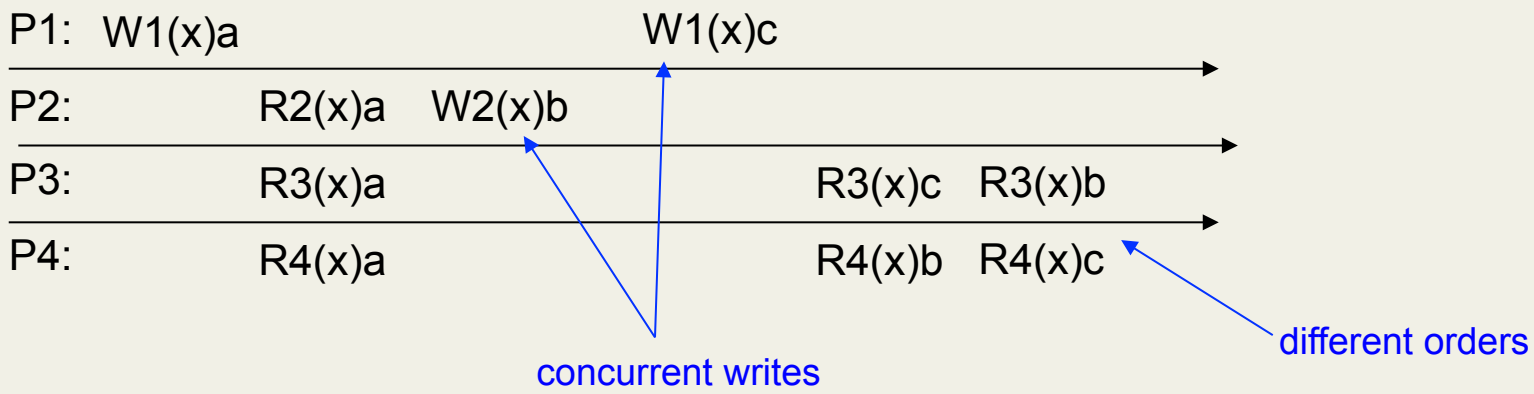
- Non-Sequential Executions
  - Impossible when a totally-ordered multicast is used to multicast updates
  - In other words, the system we just built only permit sequential executions



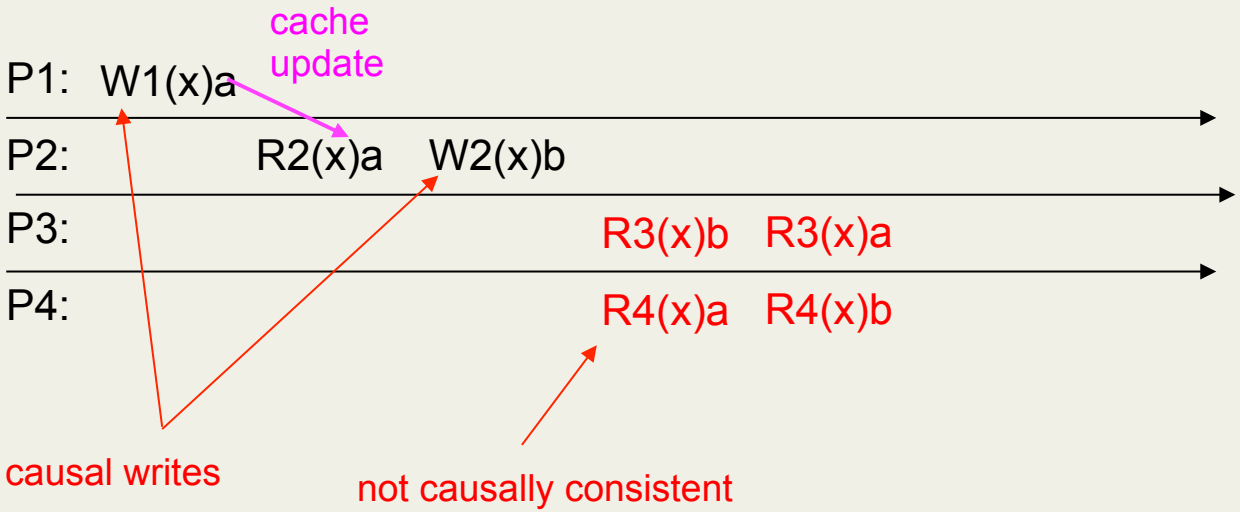
Causal Consistency

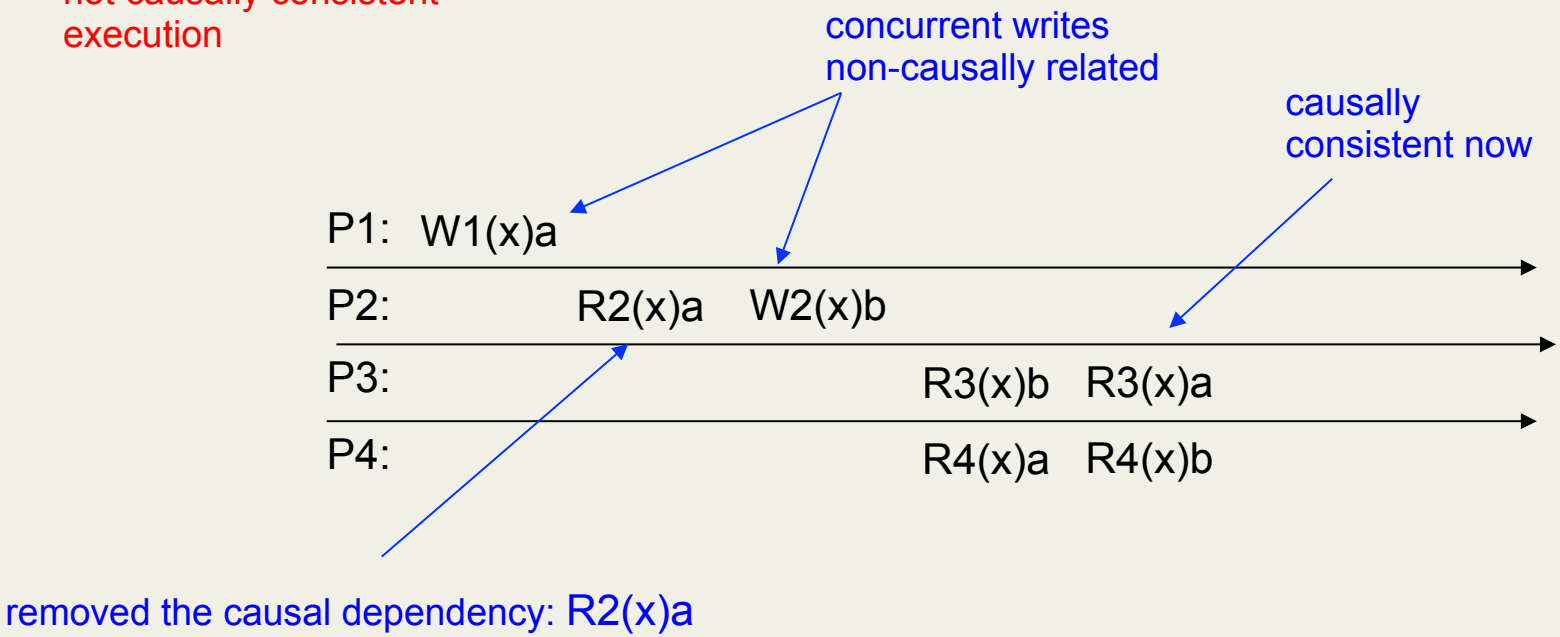
- Weaker Consistency
  - Harder to use, potentially more parallelism
  - Causality
    - If an event E2 is caused or may be influenced by an event E1
    - Causality requires that everyone sees the event E1 before the event E2

*Writes that are **potentially** causally related must be seen by all processes in the same order. Concurrent writes may be seen in a different order on different machines.*

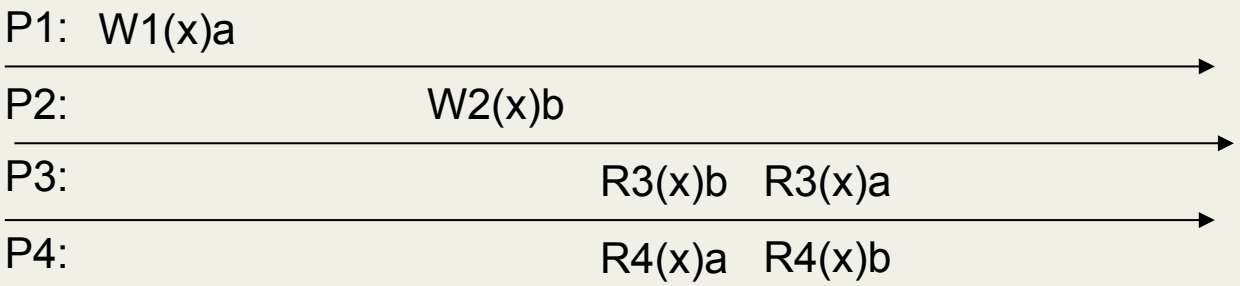


Causal Consistency





Causal Consistency



P1	**P2**	**P3**	**P4**
x=a;	x=b;	switch(x)	switch(x)
		case a:	case a:
		…	…
		case b:	case b:
		...	...

Sequential Consistency:

both **P3** and **P4** see (a,b)
both **P3** and **P4** see (b,a)

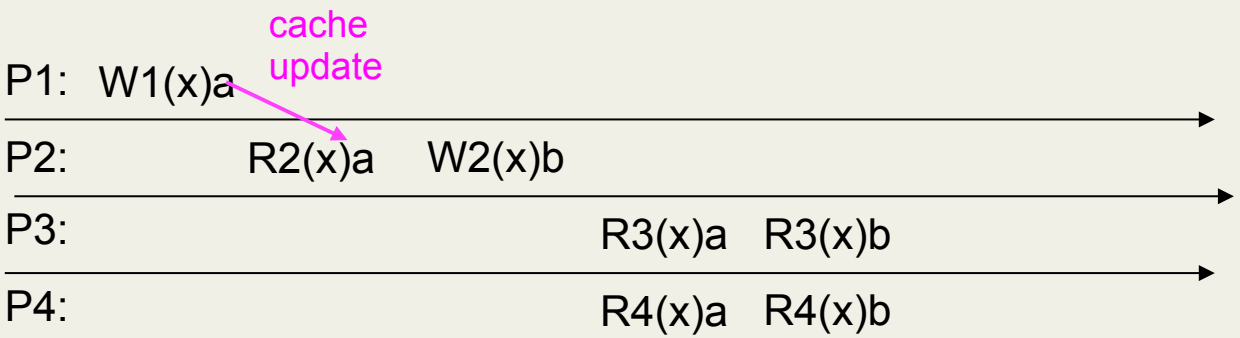
Causal Consistency:

P3 see either (a,b) or (b,a)
P4 see either (a,b) or (b,a)

Causal consistency covers 99% of the natural programming cases...

If the assignments of **P1** and **P2** are not ordere how could the rest of the program depends on that order?

Causal Consistency



P1	**P2**	**P3**	**P4**
x=a;	if (x==a)	switch(x)	switch(x)
	x=b;	case a:	case a:
		…	…
		case b:	case b:
		...	...

Sequential Consistency:

both **P3** and **P4** see (a,b)

Causal Consistency:

both **P3** and **P4** see (a,b)

Obviously, the two writes are causally depende
Hence they must be seen in the order...

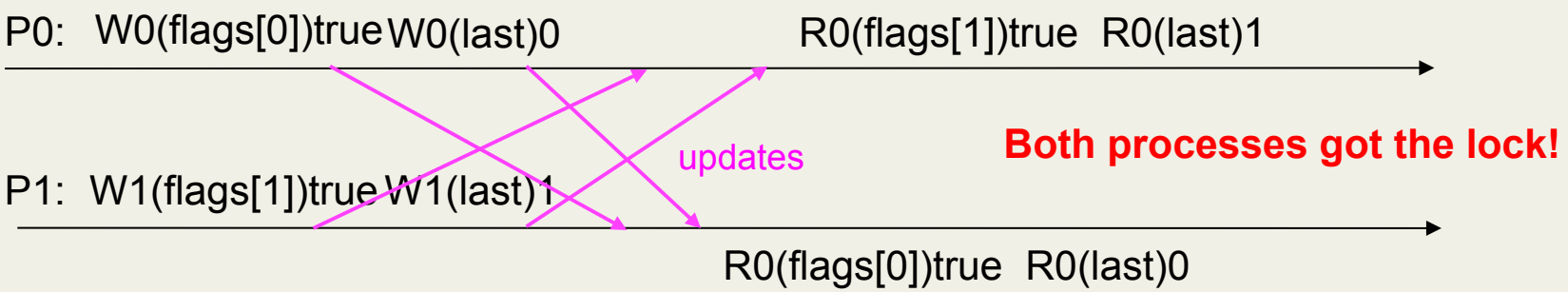
Since the assignments of **P1** and **P2** are ordere
it would be obviously wrong to see the order (b
on process **P3** and **P4**

Peterson's Algoritm

```
void lock(int i) {
        flags[i] = true;
        last = i;
        while (flags[1-i] && last==i);
}

void unlock(int i) {
        flags[i] = false;
}
```

An example of a code that does not work on a memory with causal consistency...

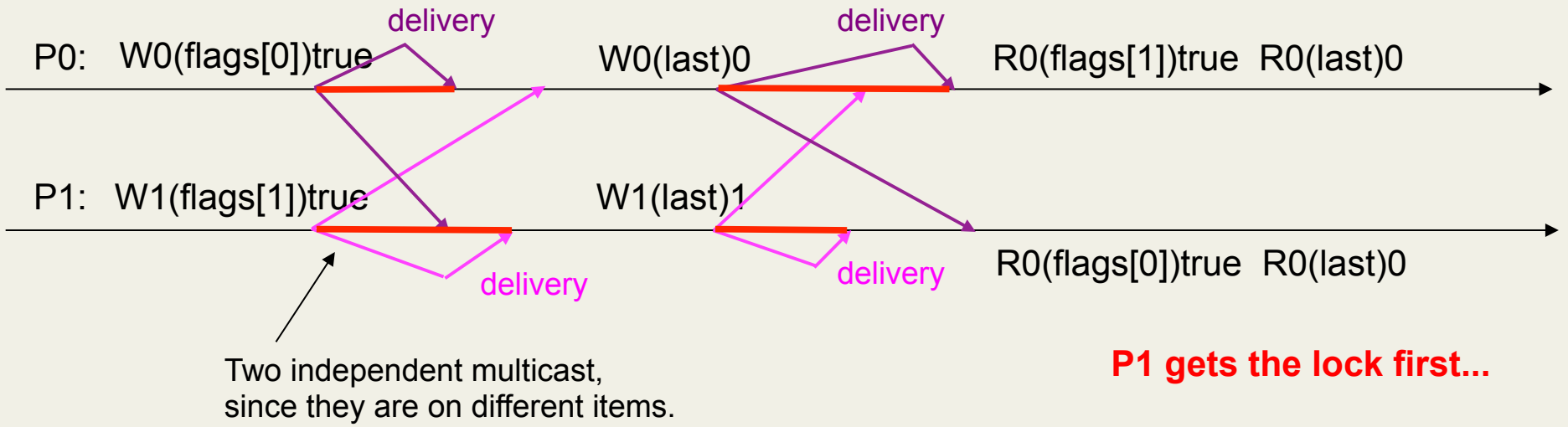
Because writes are causally independent, they can be seen in different orders by different processes



Peterson's Algorithm

```
void lock(int i) {
      flags[i] = true;
      last = i;
      while (flags[1-i] && last==i);
}

void unlock(int i) {
      flags[i] = false;
}
```

Using sequential consistency,
The writes are all totally ordered,
hence both processes see them in the same order.

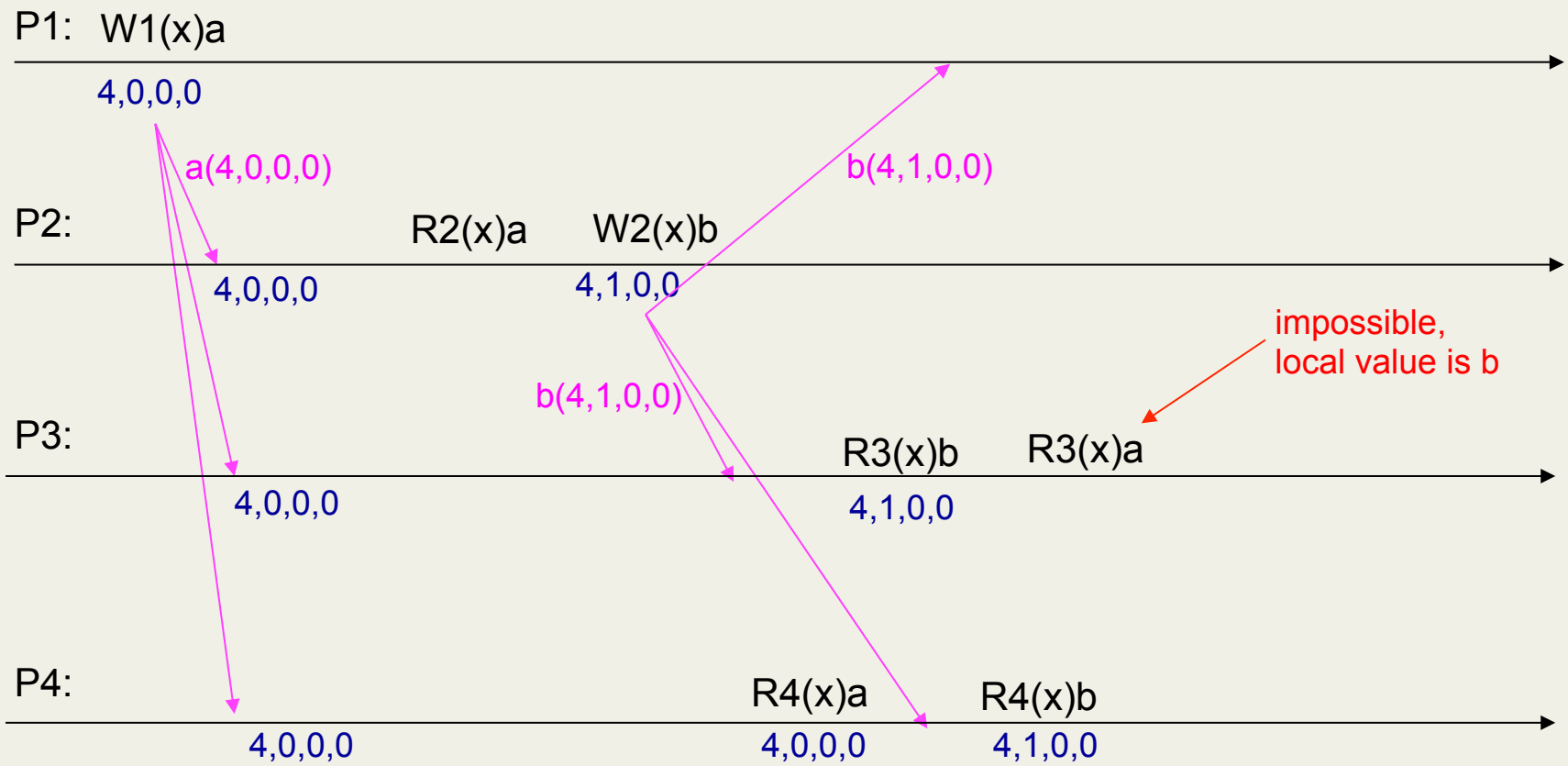


P1 gets the lock first...

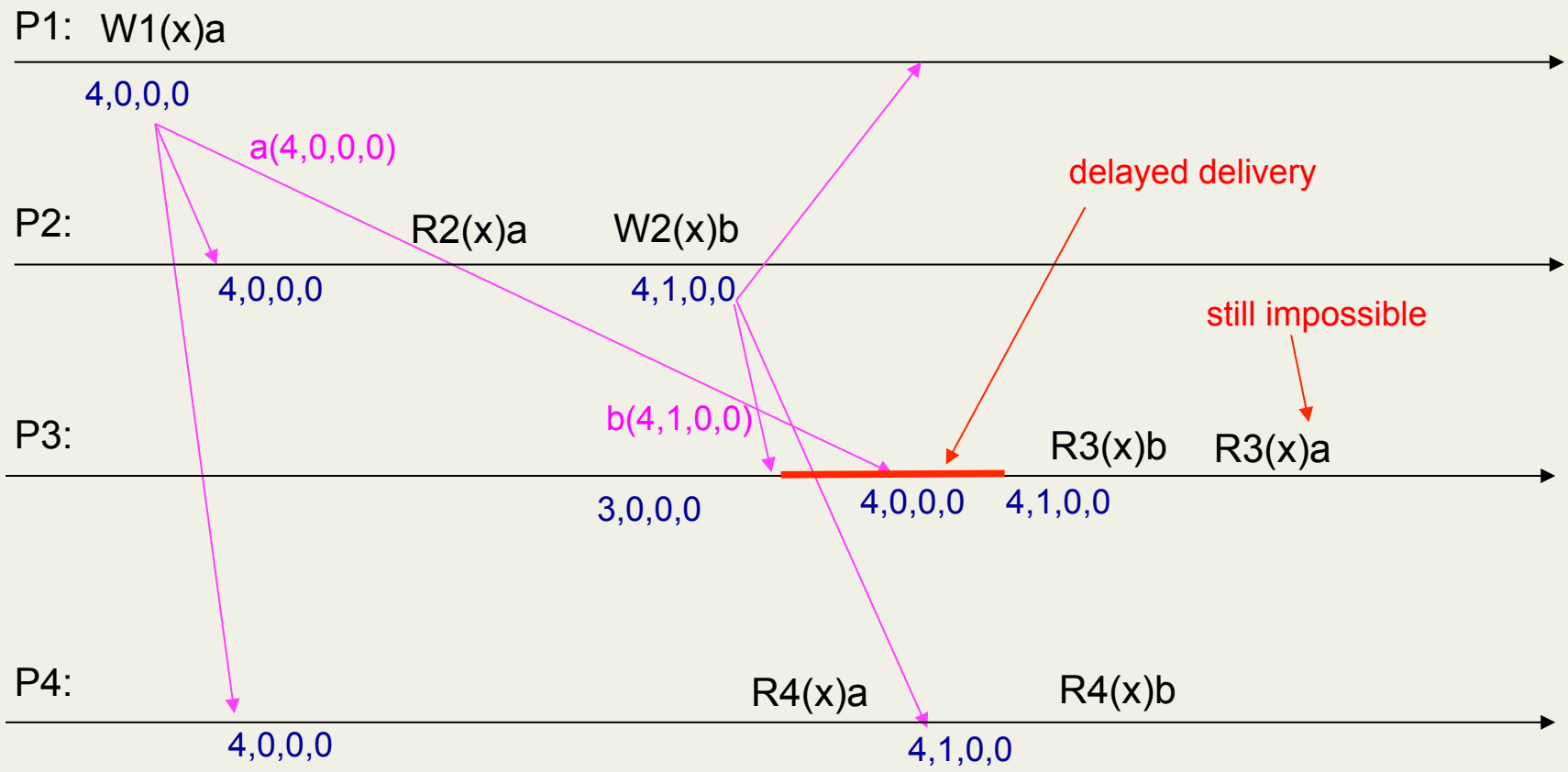
Causal Consistency

- Design
  - Requires that each process keeps tracks of which write operations it has seen
    - One may use vector clocks for this
  - Replica coherence
    - One **vector clock per data item**
      - Clock ticks on writes (as in causally-ordered multicast)
    - On local writes, multicast update messages
      - Causally-ordered multicast of the value
      - Timestamped with the vector clock

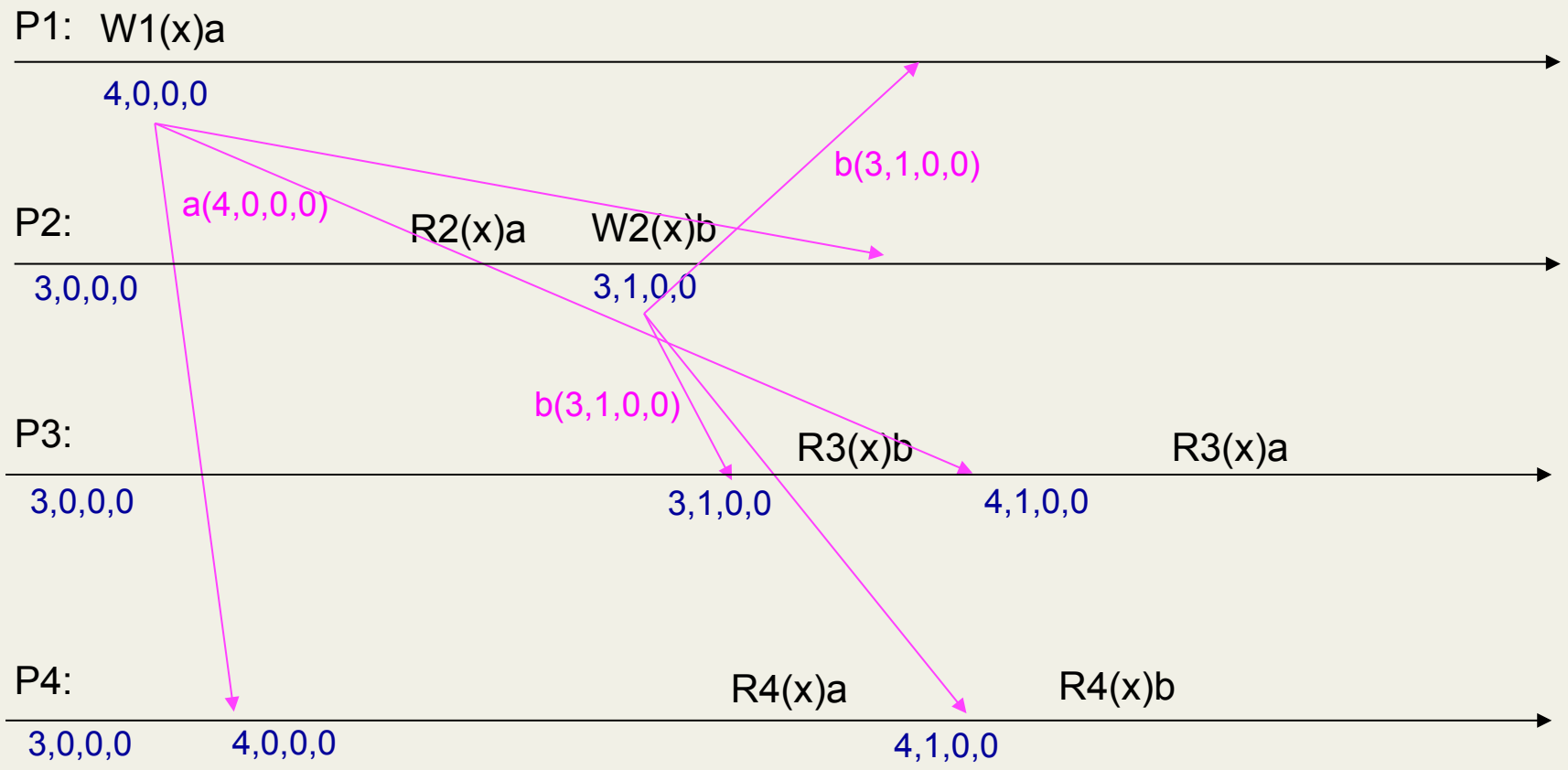
Causal Consistency



Causal Consistency



Causal Consistency



Eventual Consistency

- Eventual Consistency
  - Weaker consistency, but rather easy to use
    - Corresponds to a class of systems with simpler requirements
  - DNS example:
    - Everybody reads, only the domain owner updates DNS records
      - It is ok to read out of date records for a while
      - Use lazy background update messages
    - *Eventually*, copies will get consistent
  - Web example:
    - Same reality about the Web and web page updates
    - Even including in-network caching à la Akamai

Synchronization and Consistency

- Are they related?
  - Consistency defines the behavior of your “distributed memory”
  - Synchronization provides tools to control a “distributed execution”
- Absolutely
  - Even with a sequentially consistent memory, one needs mutual exclusion
  - Consistency protocols applied on every memory operations are costly
  - Considered memory consistency and synchronization is more efficient

Synchronization and Consistency

- Let's discuss the Compare-And-Swap instruction...
  - Used in multi-core environment to implement mutual exclusion and other locks
  - To illustrate the difference between concurrency control and memory consistency
- CAS principle
  - C-like code given below
  - Only works if the CAS instruction is atomic (hardware locking the memory bus)

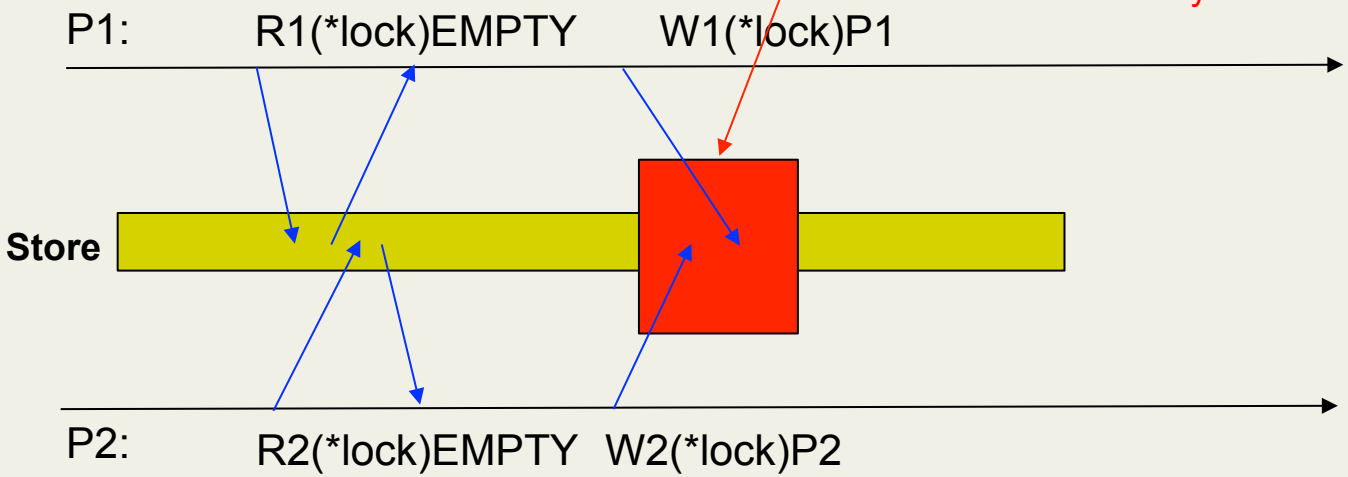
```
boolean CAS(int *lock, int value, int new_value) {
    if (*lock==value) {
      *lock = new_value;
      return true;
    }
  return false;
}


void lock(int *lock, int owner) {
  while(!CAS(*lock, EMPTY, owner);
}
```

Synchronization and Consistency

```
boolean CAS(int *lock, int value, int new_value) {
    if (*lock==value) {
      *lock = owner;
      return true;
    }
  return false;
}
```

DOES NOT WORK!

We need a locking procotol to properly acquire locks and we need a consistency protocol to properly read and write correct values
In a multi core machine, the CAS instruction locks the shared memory bus.



Synchronization and Consistency

- Basic Idea
  - Avoid double work... since we need both consistency and synchronization protocols
  - For instance, we have since that acquiring a lock and sequential consistency
  - both require something based on the same technique as our totally-ordered
  - multicast based on logical clocks
- Principles
  - Associate a monitor with one or more data items
    - Called *protected data items*
  - Coordinate consistency and synchronization protocols
    - Monitor operations must respect sequential consistency
      - Enter and leave critical sections are seen in the same order by all processes
    - Data consistency
      - All writes on protected items must be visible locally before one
      - enters the critical section
      - Accesses (reads or writes) on protected items outside the critical
      - section are undefined

Synchronisation and Consistency

Examples

p1	E(x)	W(x)a	W(x)b	L(x)											
p2								R(x)a	R(x)b	E(x)	R(x)b				
p3										R(x)a	E(x)	R(x)b			
p4										R(x)b	E(x)	R(x)b			

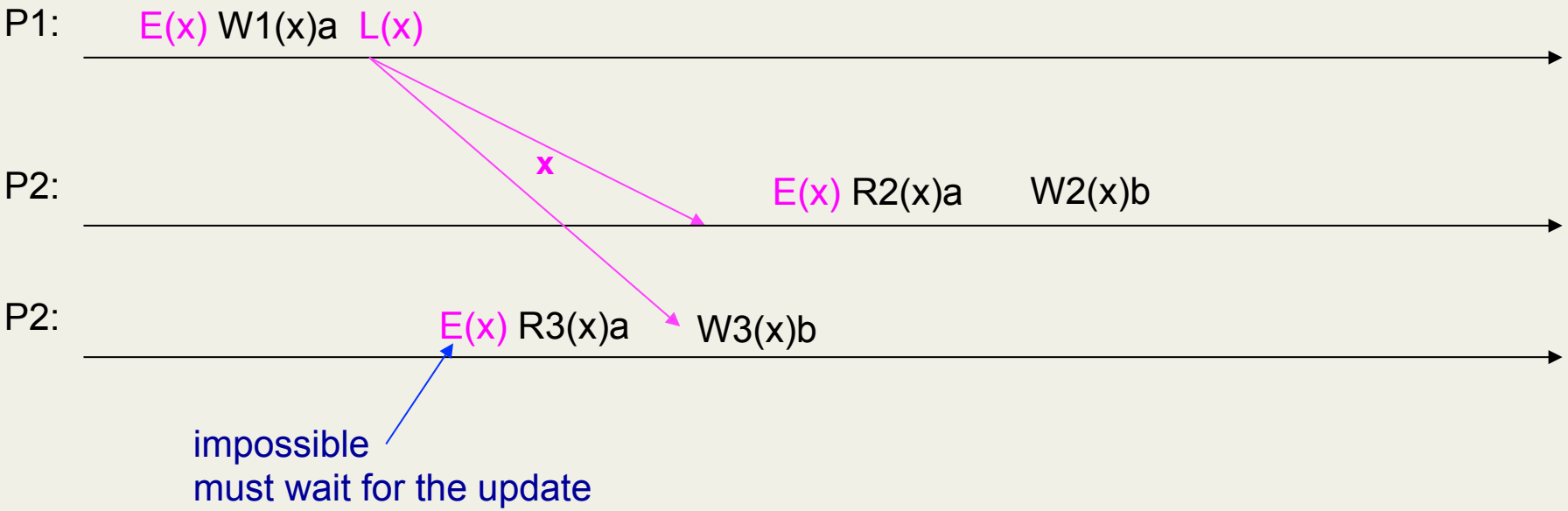
possible execution

p1	E(x)	W(x)a	W(x)b	L(x)											
p2													E(x)	R(x)a	

impossible to read the value a

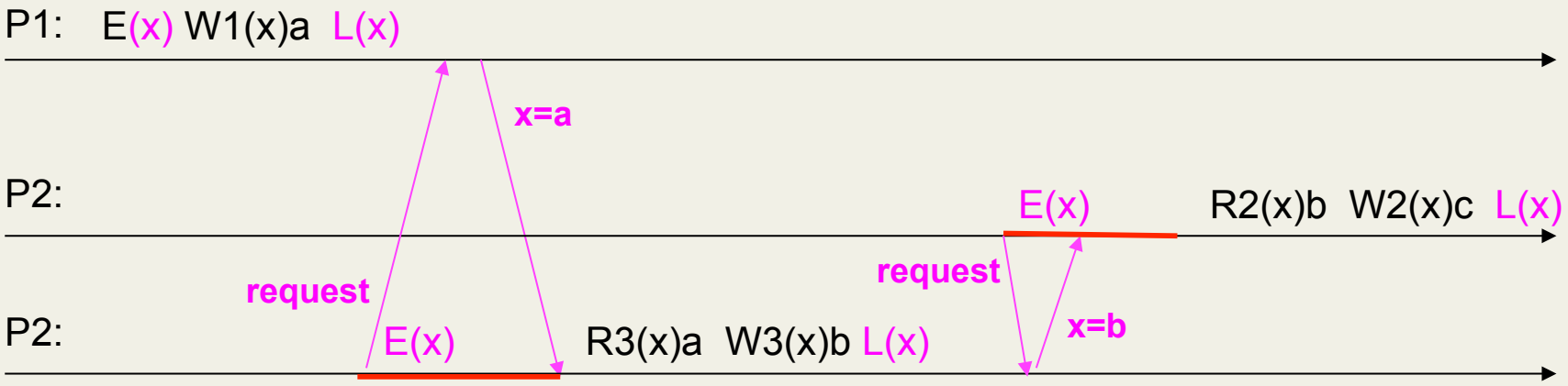
Release Consistency

- Eager Approach
  - Only enters the critical section once all local copies are up to date
    - Acquiring the critical section and receiving pending updates must be coordinated
  - **When leaving the critical section**
    - ***Eagerly* send local updates towards other replicas**



Entry Consistency

- Lazy Approach
  - **Upon entering the critical section**
    - **Pull missing updates on protected items**
    - Optimization: pigging back updates on granting access
  - When leaving the critical section
    - Nothing needs to be done



Discussing Consistency

- Undefined Semantics
  - Accessing protected data items outside critical sections