Safety, Liveness, and Consistency

Ellis Michael

How Do We Specify Distributed Systems?

- **Execution:** Sequence of events (i.e., steps taken by the system), potentially infinite.
- **Property:** A predicate on executions.
- **Safety property:** Specifies the "bad things" that shouldn't happen in any execution.
- **Liveness property:** Specifies the "good things" that should happen in every execution.

(See paper for formal definitions.)

THEOREM: EVERY PROPERTY IS EXPRESSIBLE AS THE CONJUNCTION OF A SAFETY PROPERTY AND A LIVENESS PROPERTY.

[Alpern and Schneider. 1987]

THEOREM: EVERY PROPERTY IS EXPRESSIBLE AS THE CONJUNCTION OF A SAFETY PROPERTY AND A LIVENESS PROPERTY.

Neat automata theory!

[Alpern and Schneider. 1987]

Some Properties

- The system never deadlocks.
- Every client that sends a request eventually gets a reply.
- Both generals attack simultaneously.

More Properties: Consensus

n processes, all of which have an input value from some domain.

Processes output a value by calling *decide*(v). Non-faulty processes continue correctly executing protocol steps forever. We usually denote the number of faulty processes f.

- **Agreement:** No two correct processes decide different values.
- **Integrity:** Every correct process decides at most one value, and if a correct process decides a value v, some process had v as its input.
- **Termination:** Every correct process eventually decides a value.

Consistency is Key!

Consistency: *the allowed semantics (return values) of a set of operations to a data store or shared object.*

Consistency properties specify the **interface**, not the **implementation**. The data might be replicated, cached, disaggregated, etc. "Weird" consistency semantics happen all over the stack!

Anomaly: violation of the consistency semantics

Terminology: Strength and Weakness

- **Strong consistency:** the system behaves as if there's just a single copy of the data (or almost behaves that way).

  The intuition is that things like caching and sharding are implementation decisions and shouldn't be visible to clients.

- **Weak consistency:** allows behaviors significantly different from the single store model.

- **Eventual consistency:** the aberrant behaviors are only temporary.

Why the Difference?

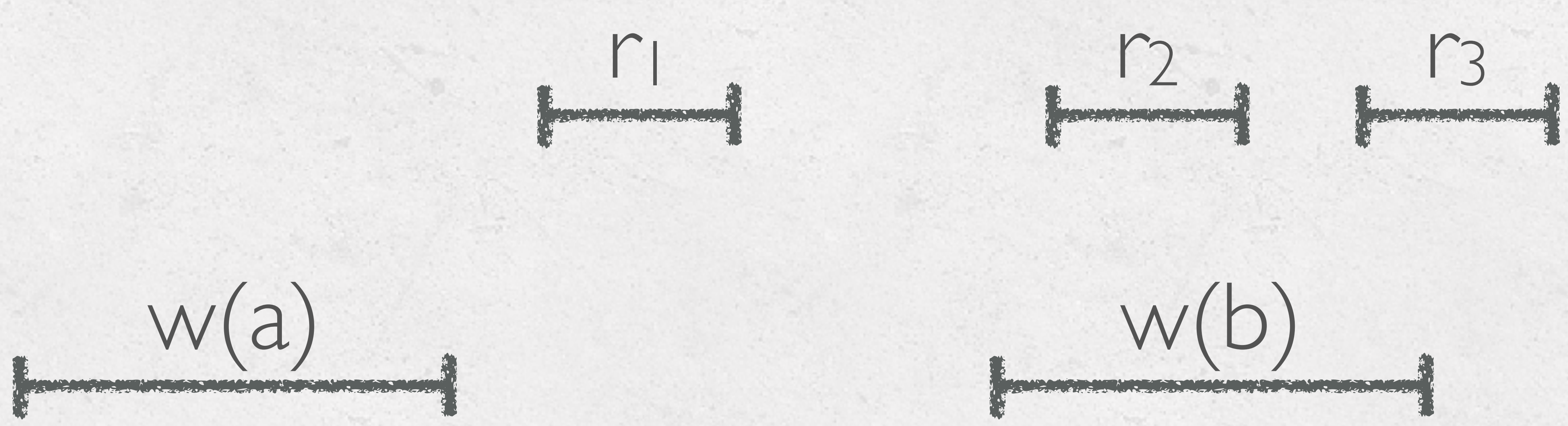
- **Performance**
  - Consistency requires synchronization/coordination when data is replicated
  - Often slower to make sure you always return right answer
- **Availability**
  - What if client is offline, or network is not working?
  - Weak/eventual consistency may be only option
- **Programmability**
  - Weaker models are harder to reason against

Lamport's Register Semantics

Registers hold a single value. Here, we consider single-writer registers only supporting `write` and `read`.

Semantics defined in terms of the *real-time* beginnings and ends of operations to the object.

- **safe:** a read not concurrent with any write obtains the previously written value
- **regular:** safe + a read that overlaps a write obtains either the old or new value
- **atomic:** safe + reads and writes behave as if they occur in some definite order

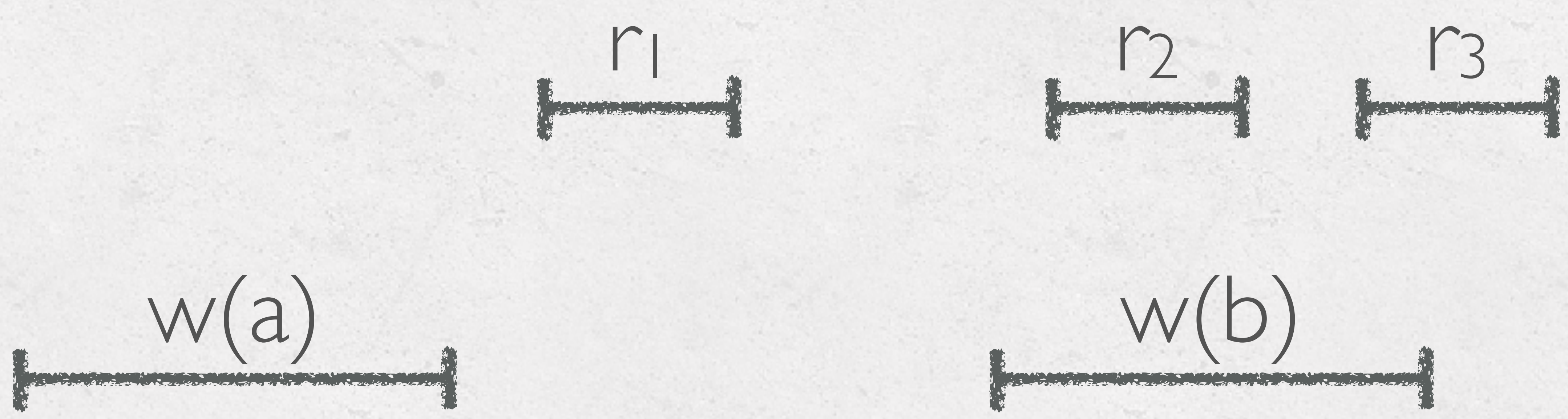


Lamport's Register Semantics

Registers hold a single value. Here, we consider single-writer registers only supporting `write` and `read`.

Semantics defined in terms of the *real-time* beginnings and ends of operations to the object.

- **safe:** a read not concurrent with any write obtains the previously written value
- **regular:** safe + a read that overlaps a write obtains either the old or new value
- **atomic:** safe + reads and writes behave as if they occur in some definite order



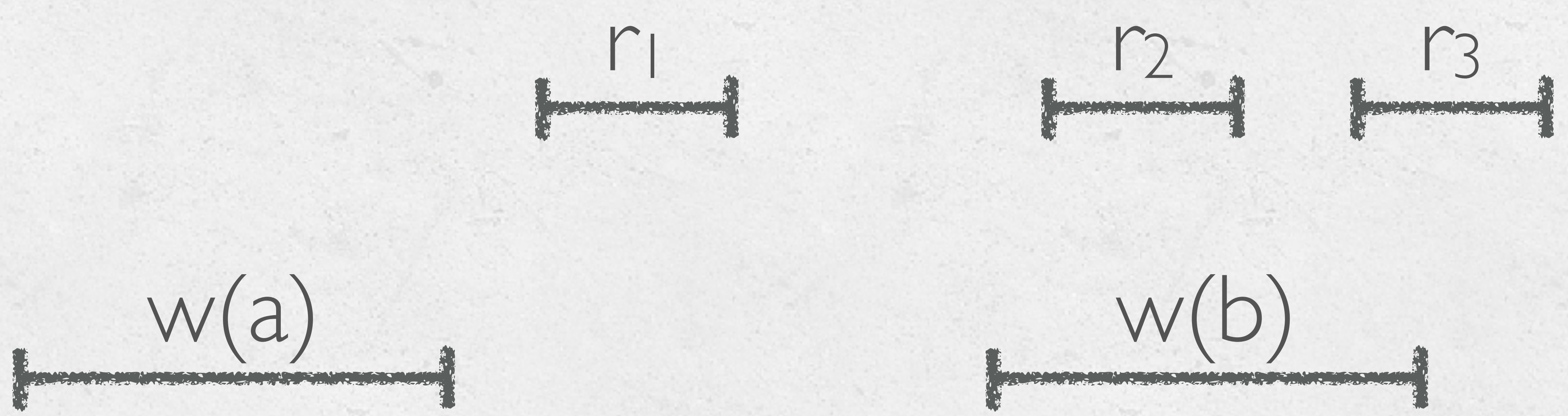
safe ⇒ $r_1 \rightarrow a$

Lamport's Register Semantics

Registers hold a single value. Here, we consider single-writer registers only supporting `write` and `read`.

Semantics defined in terms of the *real-time* beginnings and ends of operations to the object.

- **safe:** a read not concurrent with any write obtains the previously written value
- **regular:** safe + a read that overlaps a write obtains either the old or new value
- **atomic:** safe + reads and writes behave as if they occur in some definite order



safe $\Rightarrow r_1 \rightarrow a$

regular $\Rightarrow r_1 \rightarrow a \wedge (r_2 \rightarrow a \vee r_2 \rightarrow b) \wedge$

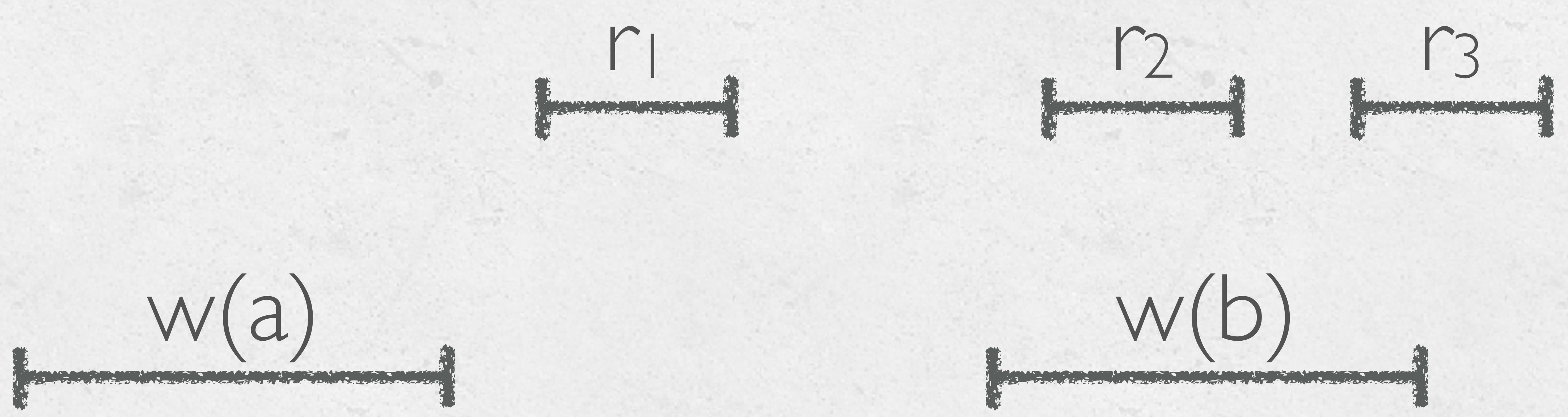
$(r_3 \rightarrow a \vee r_3 \rightarrow b)$

Lamport's Register Semantics

Registers hold a single value. Here, we consider single-writer registers only supporting `write` and `read`.

Semantics defined in terms of the *real-time* beginnings and ends of operations to the object.

- **safe:** a read not concurrent with any write obtains the previously written value
- **regular:** safe + a read that overlaps a write obtains either the old or new value
- **atomic:** safe + reads and writes behave as if they occur in some definite order



safe $\Rightarrow r_1 \to a$

regular $\Rightarrow r_1 \to a \wedge (r_2 \to a \vee r_2 \to b) \wedge$

$(r_3 \to a \vee r_3 \to b)$

atomic $\Rightarrow r_1 \to a \wedge (r_2 \to a \vee r_2 \to b) \wedge$

$(r_3 \to a \vee r_3 \to b) \wedge$

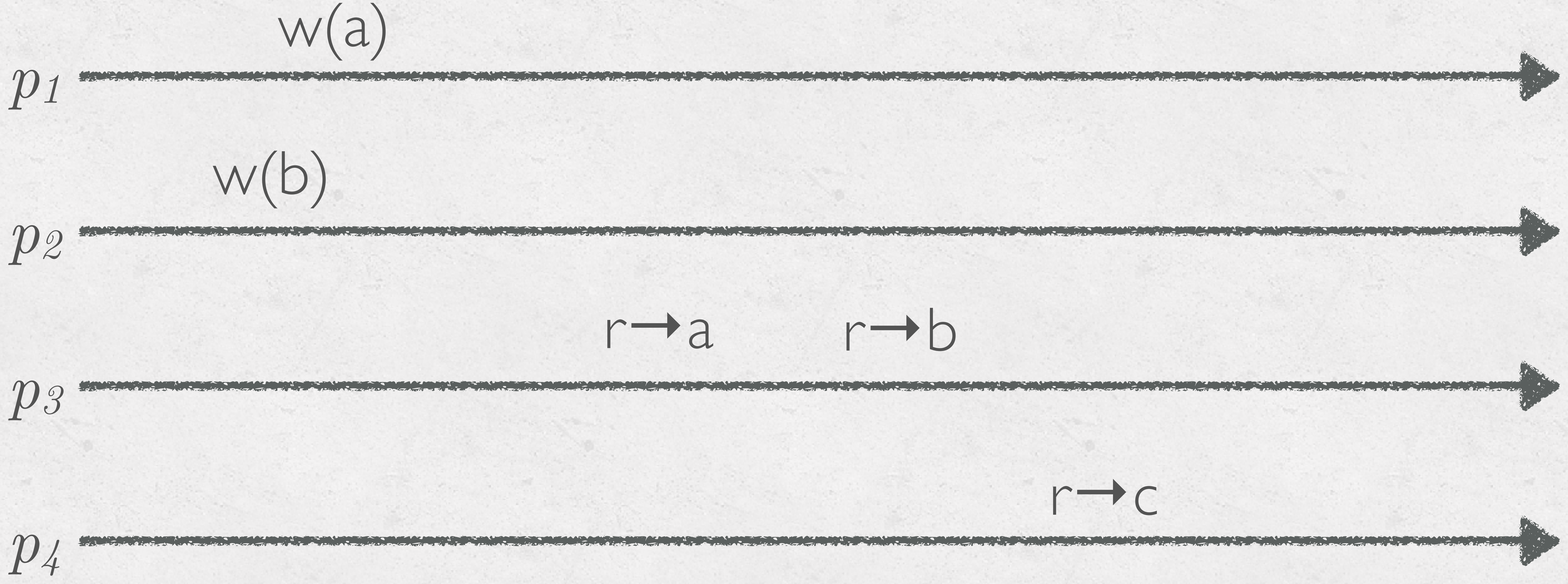
$(r_2 \to b \Rightarrow r_3 \to b)$

Sequential Consistency

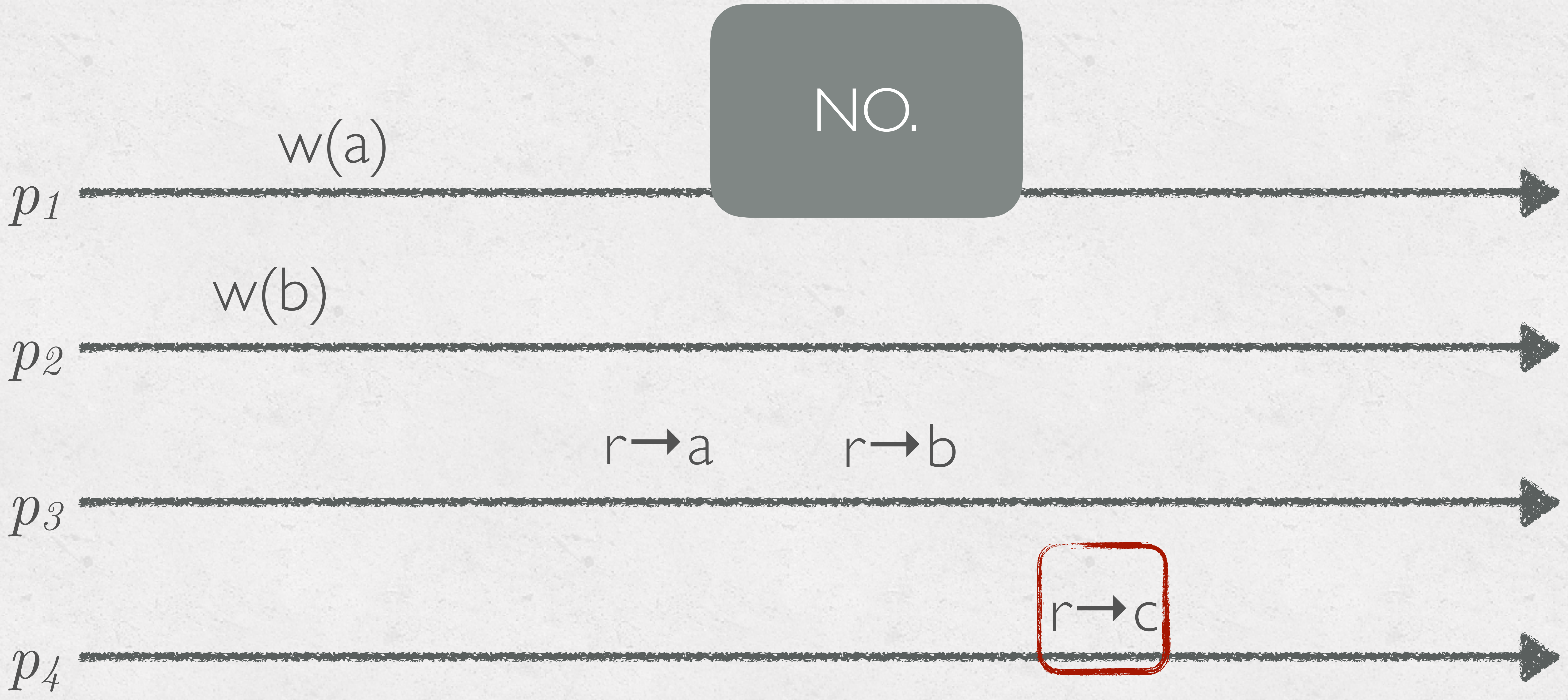
- Applies to arbitrary shared objects.
- Requires that a history of operations be *equivalent to a legal sequential history*, where a legal sequential history is one that respects the local ordering at each node.
- Called **serializability** when applied to transactions

Is It Sequential?

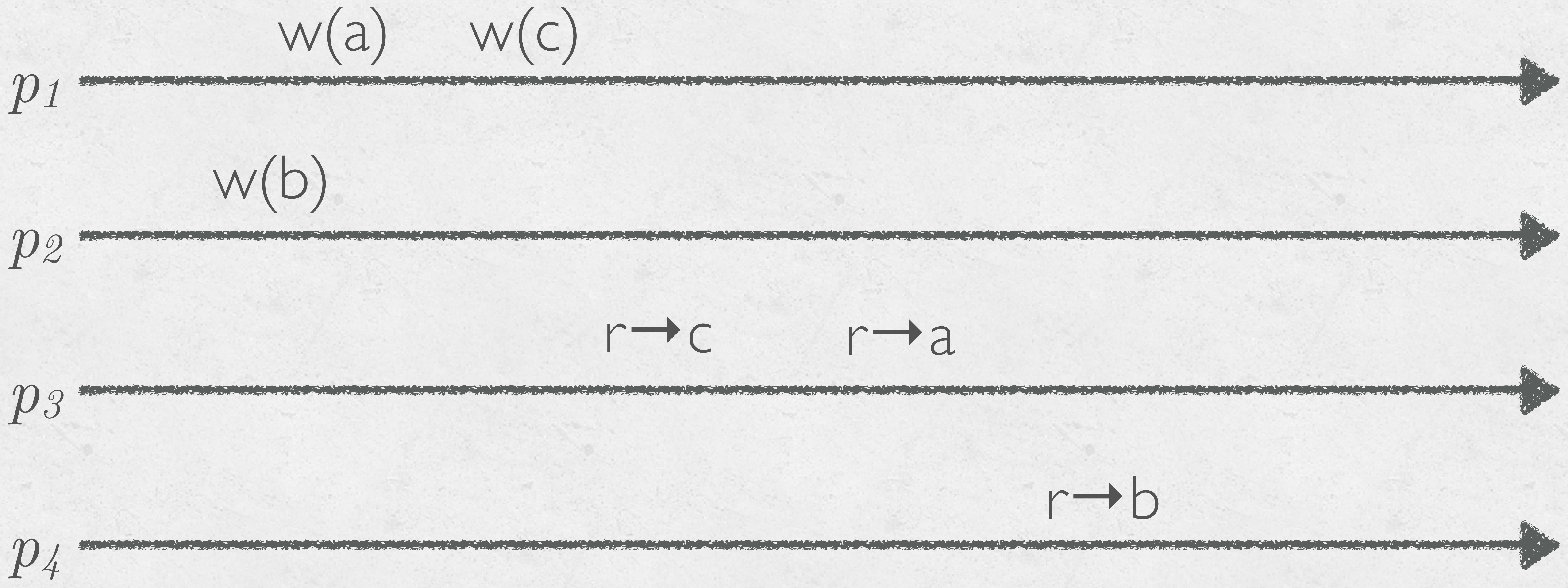
Is It Sequential?



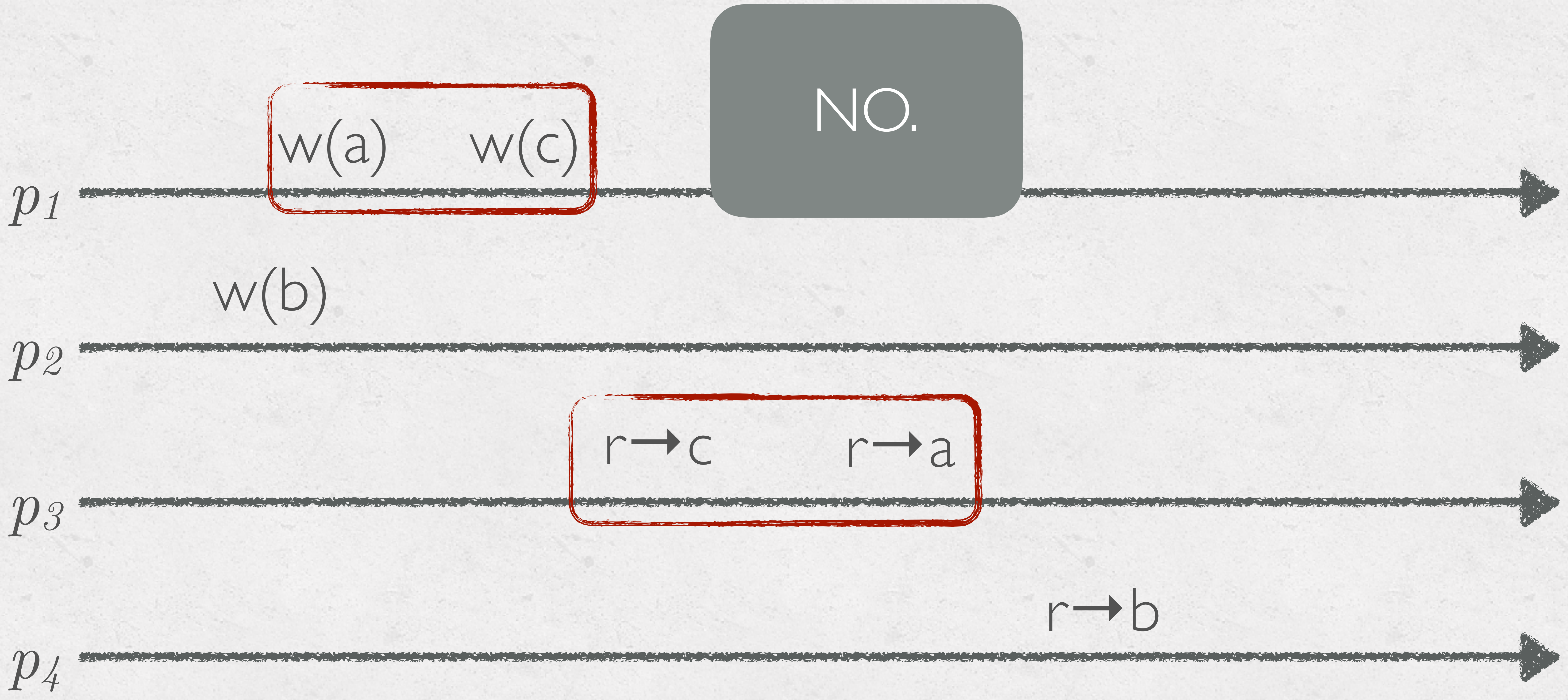
Is It Sequential?



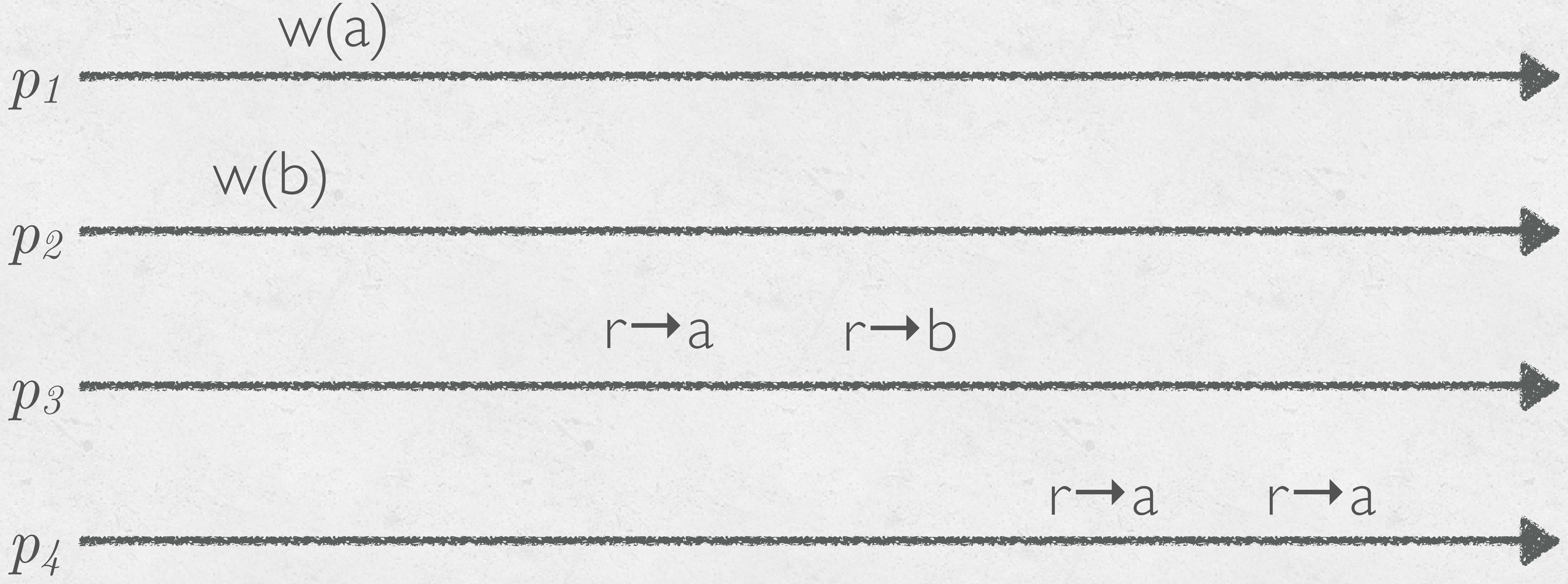
Is It Sequential?



Is It Sequential?



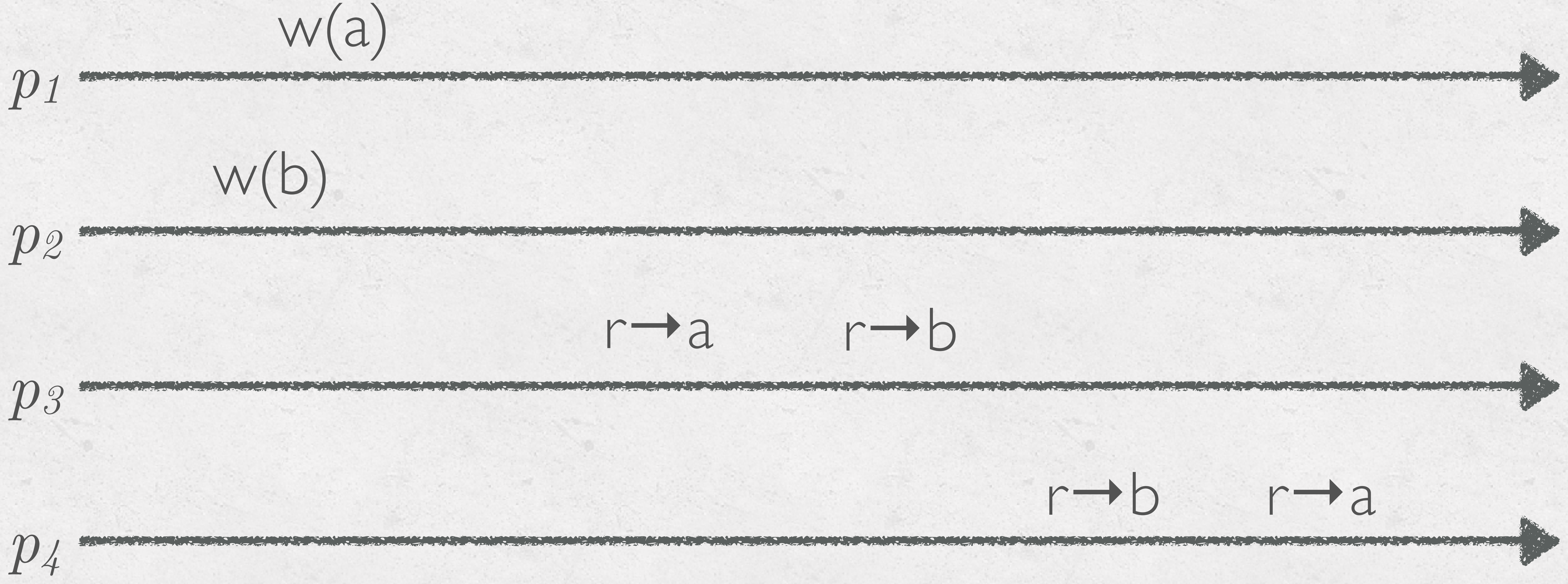
Is It Sequential?



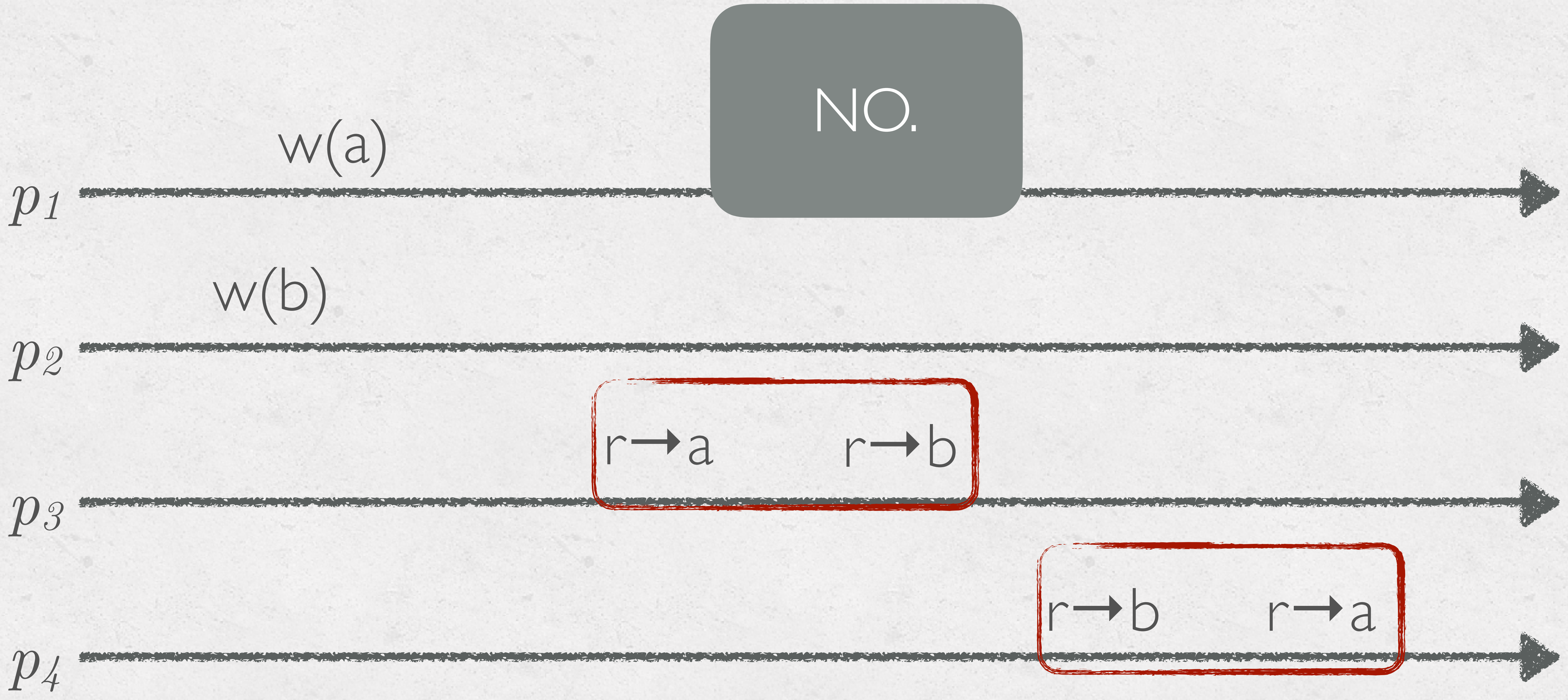
Is It Sequential?

w(a) r→a r→a r→a w(b) r→b

Is It Sequential?



Is It Sequential?



Linearizability

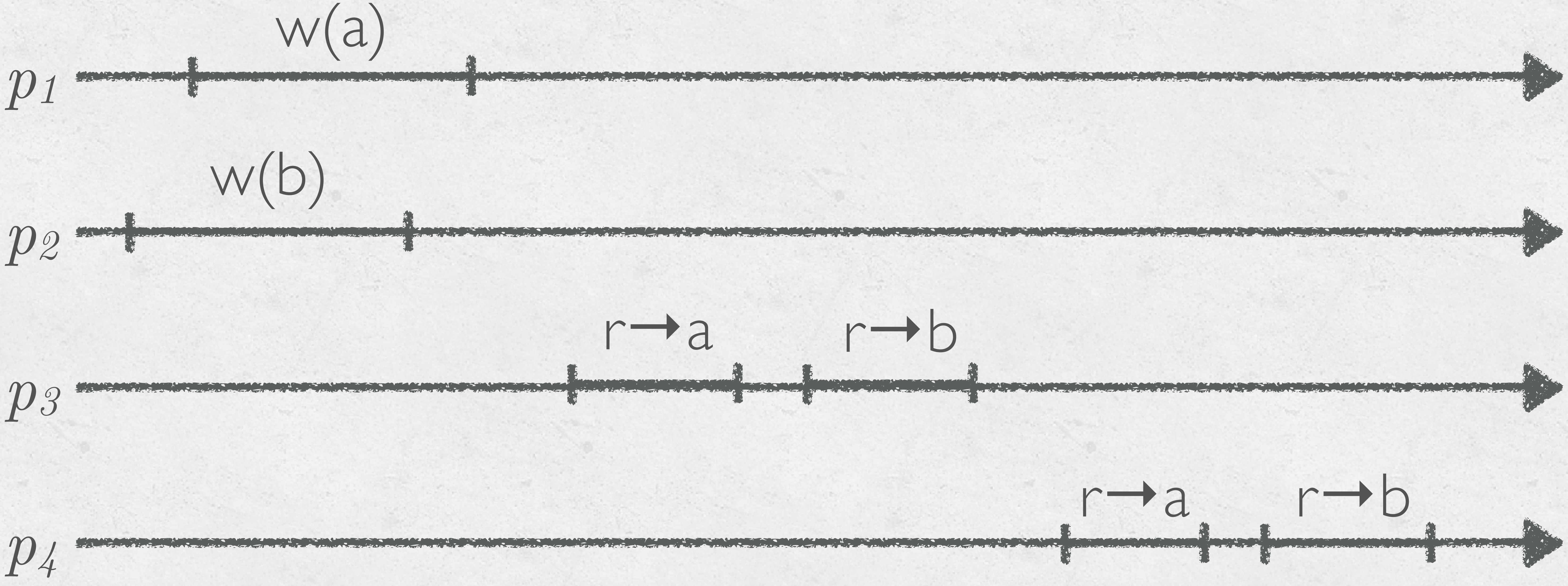
Linearizability = sequential consistency + respects real-time ordering.

If e_1 **ends** before e_2 **begins**, *then* e_1 *appears before* e_2 *in the sequential history.*

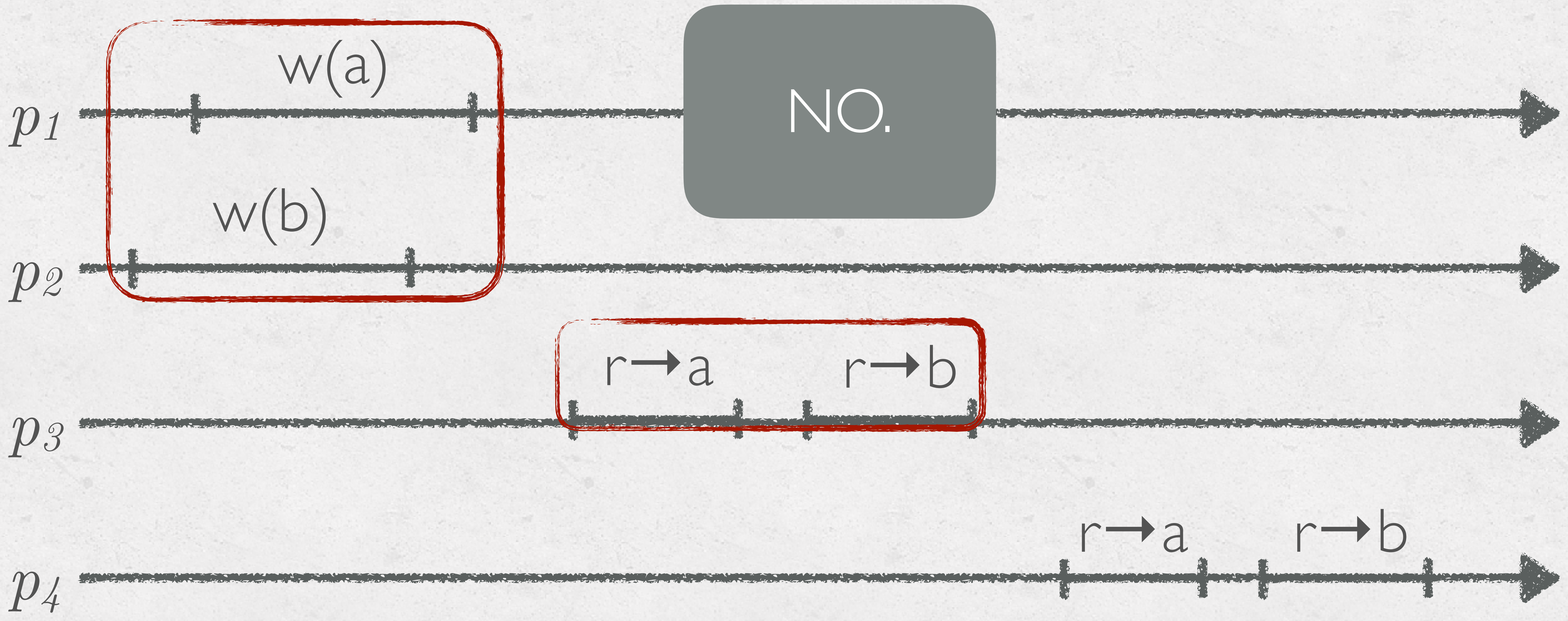
Linearizable data structures behave as if there's a single, correct copy.

Atomic registers are linearizable.

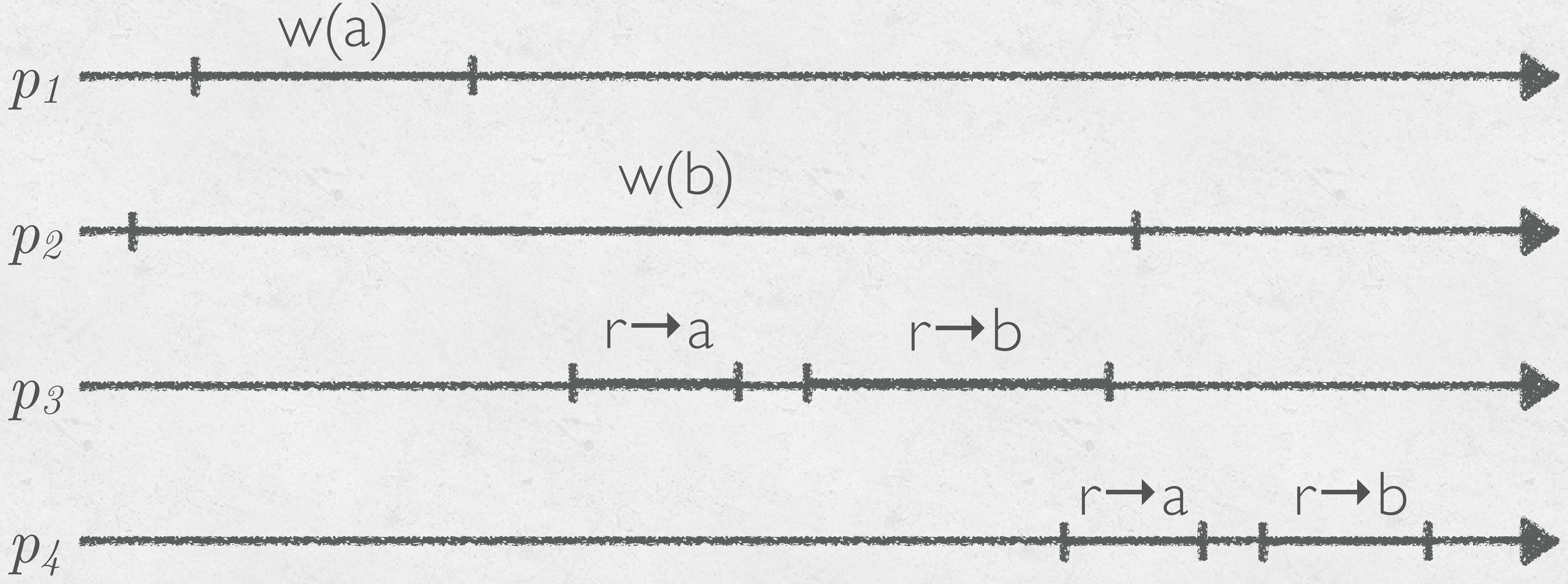
Is It Linearizable?



Is It Linearizable?

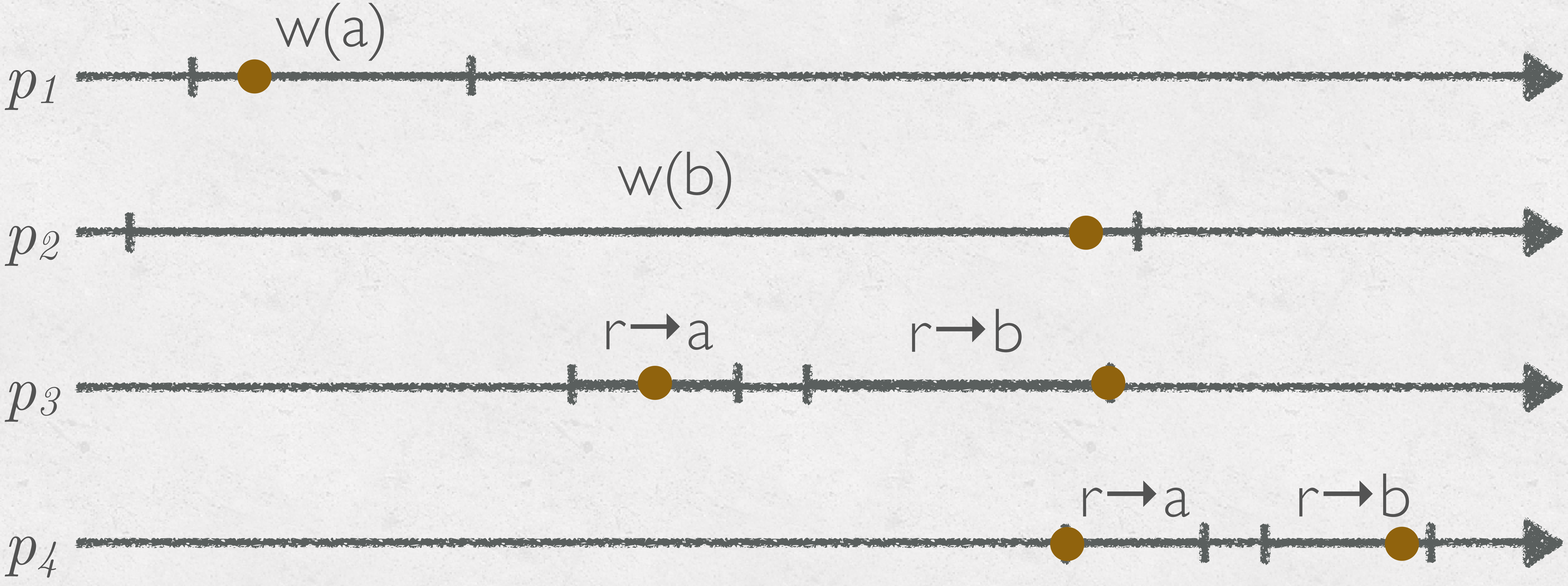


Is It Linearizable?



Is It Linearizable?

YES!



Linearizability vs. Sequential Consistency

- Sequential consistency allows operations to appear out of real-time order. How could that happen in reality?

Linearizability vs. Sequential Consistency

- Sequential consistency allows operations to appear out of real-time order. How could that happen in reality?
- The most common way systems are sequentially consistency but not linearizability is that they allow read-only operations to return **stale data**.

Stale Reads

Stale Reads

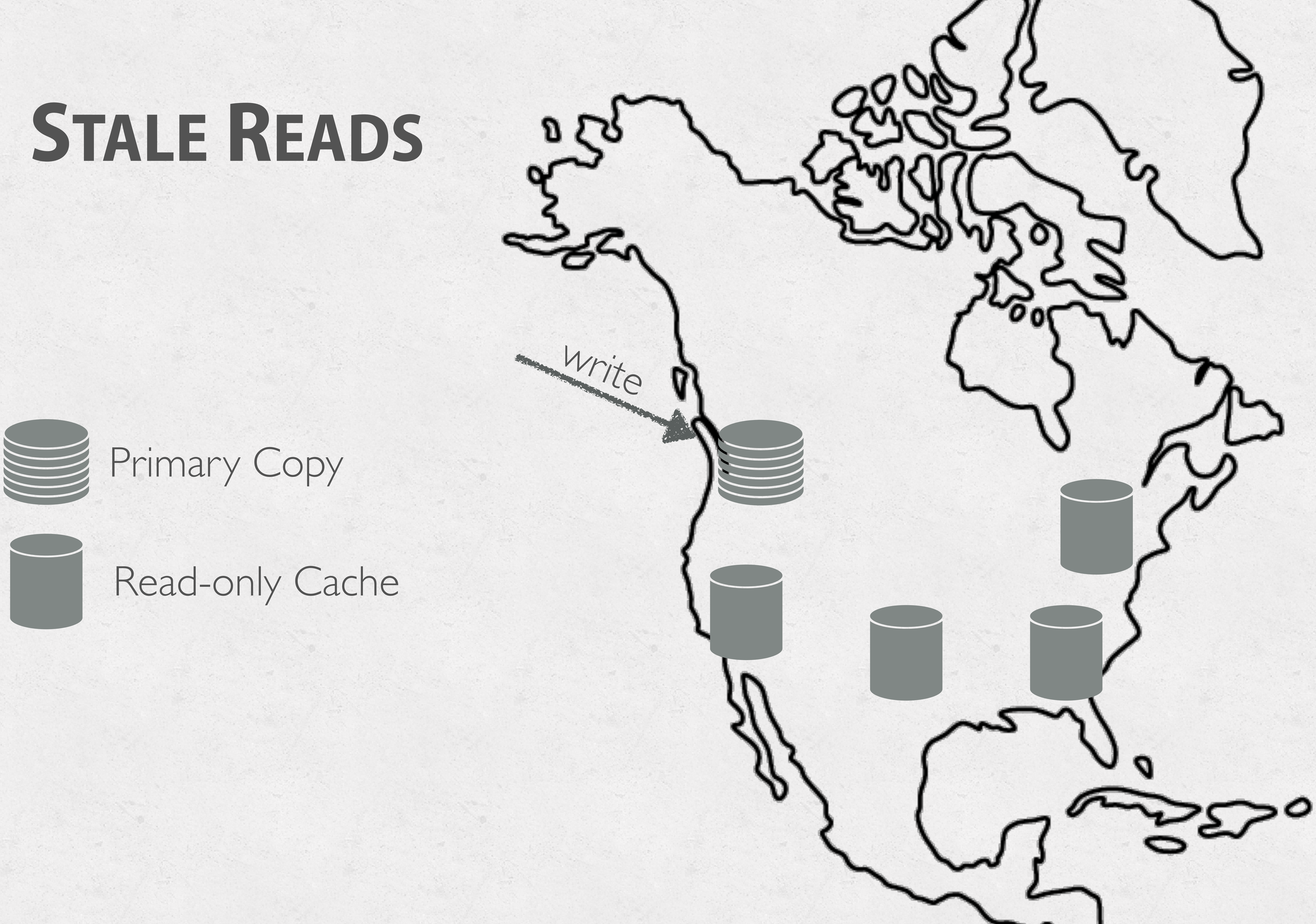
Primary Copy

Stale Reads

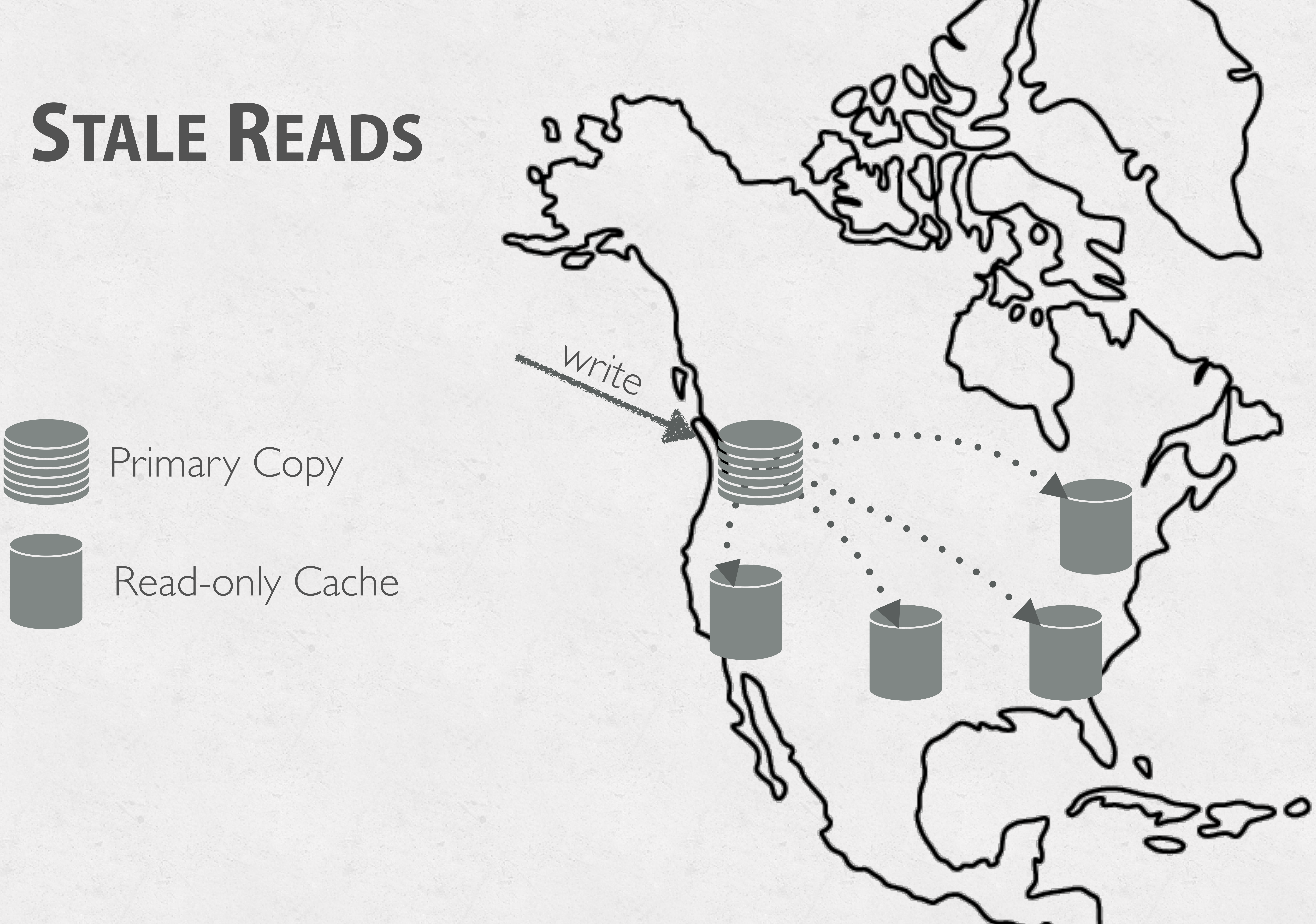
Primary Copy

Read-only Cache

Stale Reads



Stale Reads



Causal Consistency

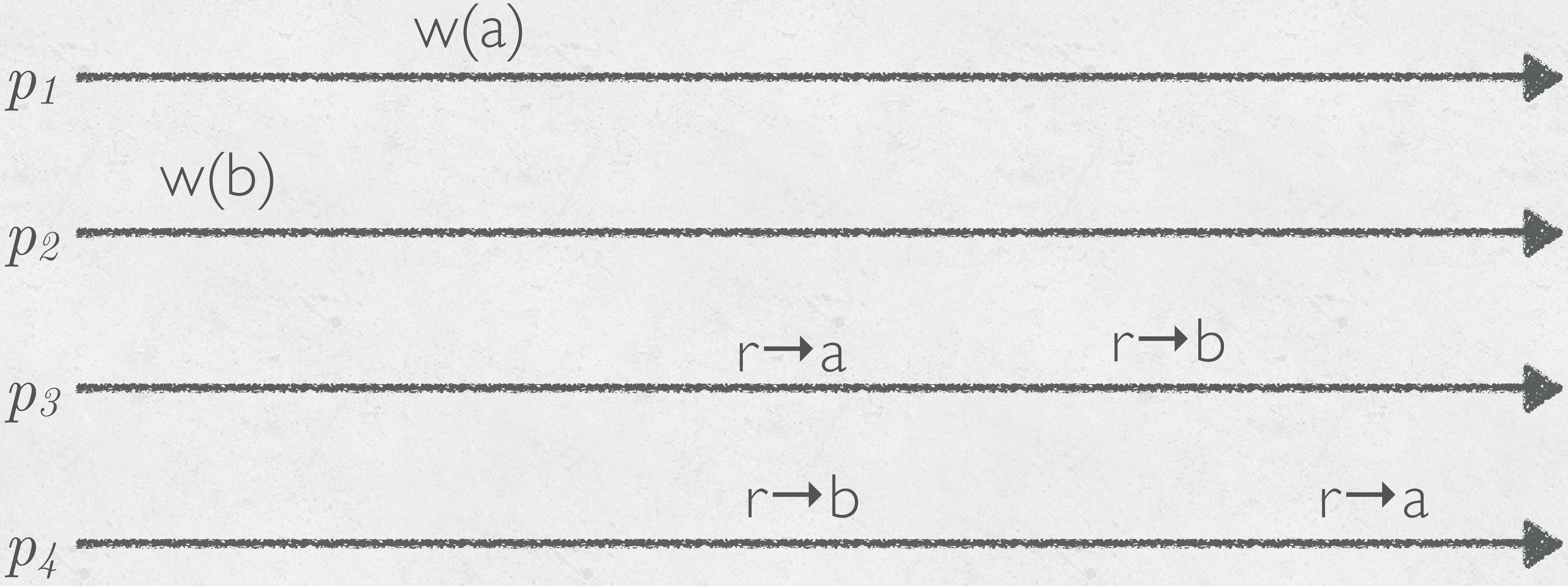
- Writes that are not concurrent (i.e., writes related by the happens-before relation) must be seen in that order. Concurrent writes can be seen in different orders on different nodes.

Causal Consistency

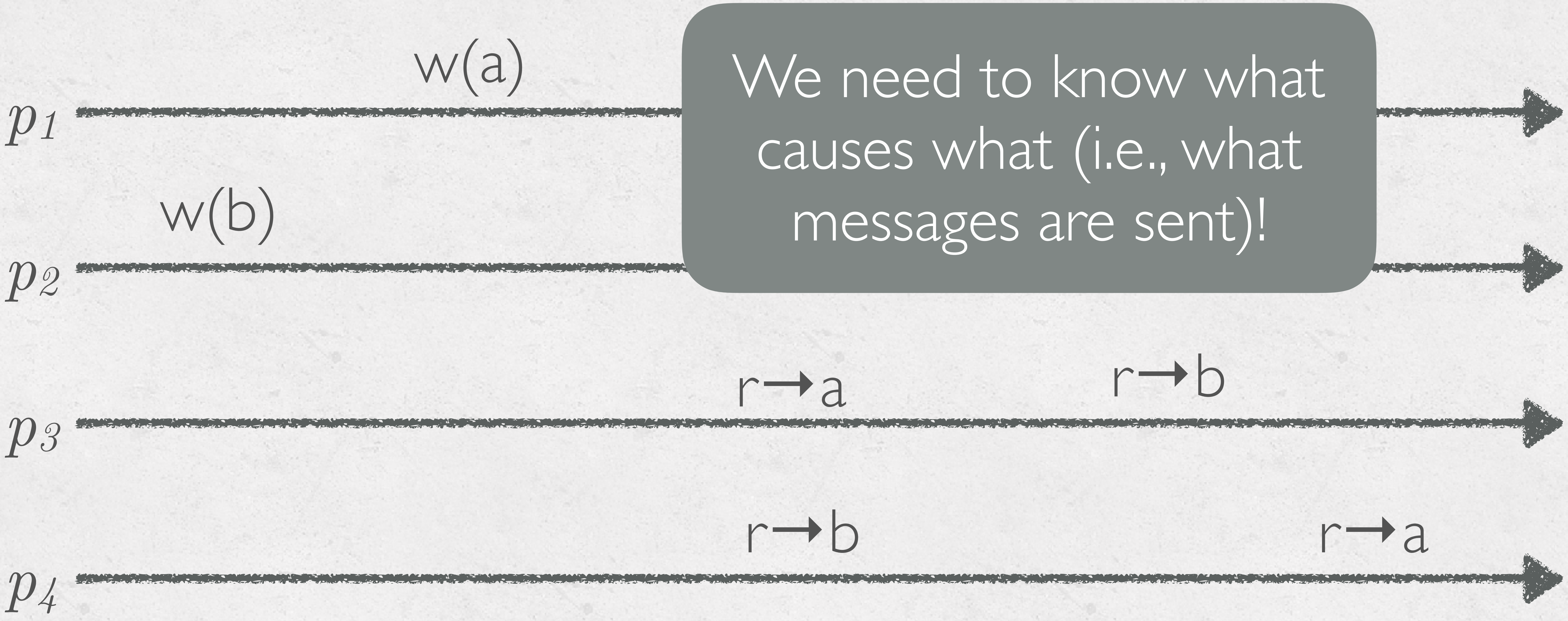
- Writes that are not concurrent (i.e., writes related by the happens-before relation) must be seen in that order. Concurrent writes can be seen in different orders on different nodes.

- Linearizability implies causal consistency.

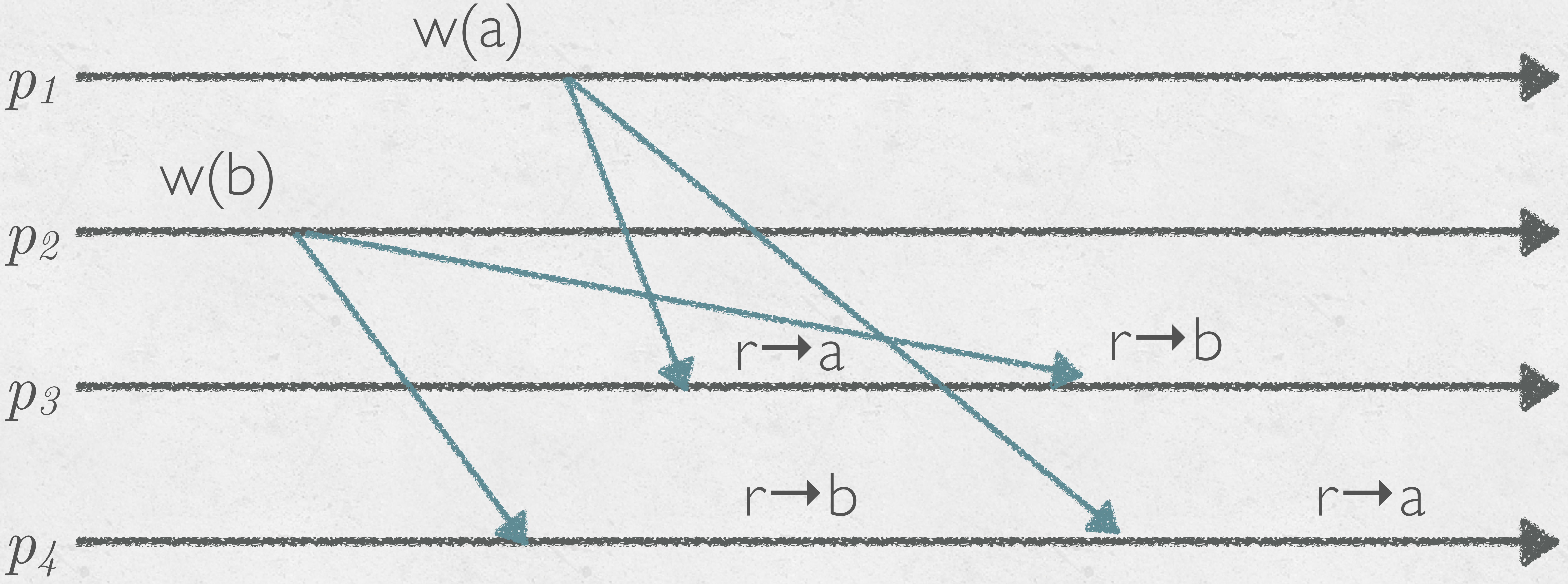
Is It Causal?



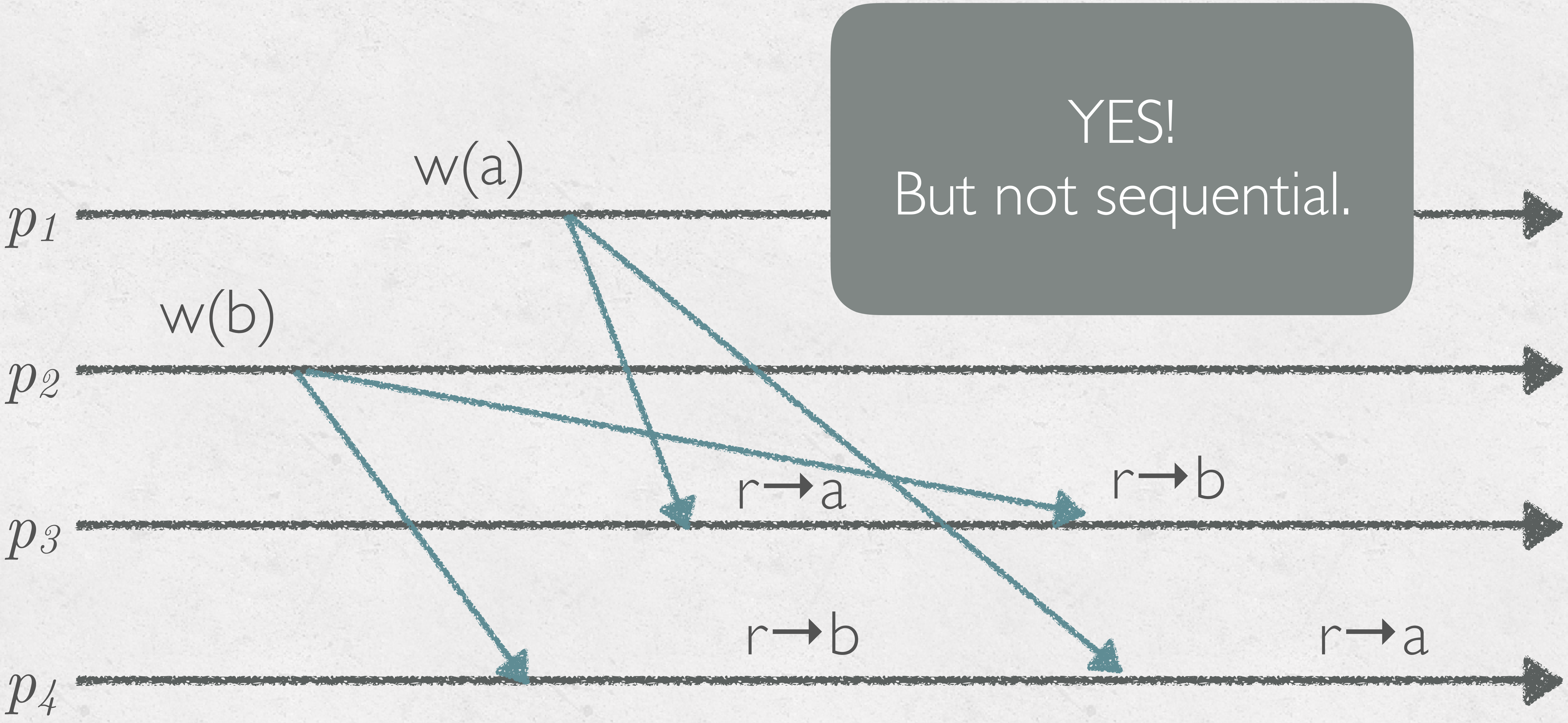
Is It Causal?


w(a)
w(b)
We need to know what causes what (i.e., what messages are sent)!
p_1
p_2
p_3
p_4
r→a
r→b
r→b
r→a

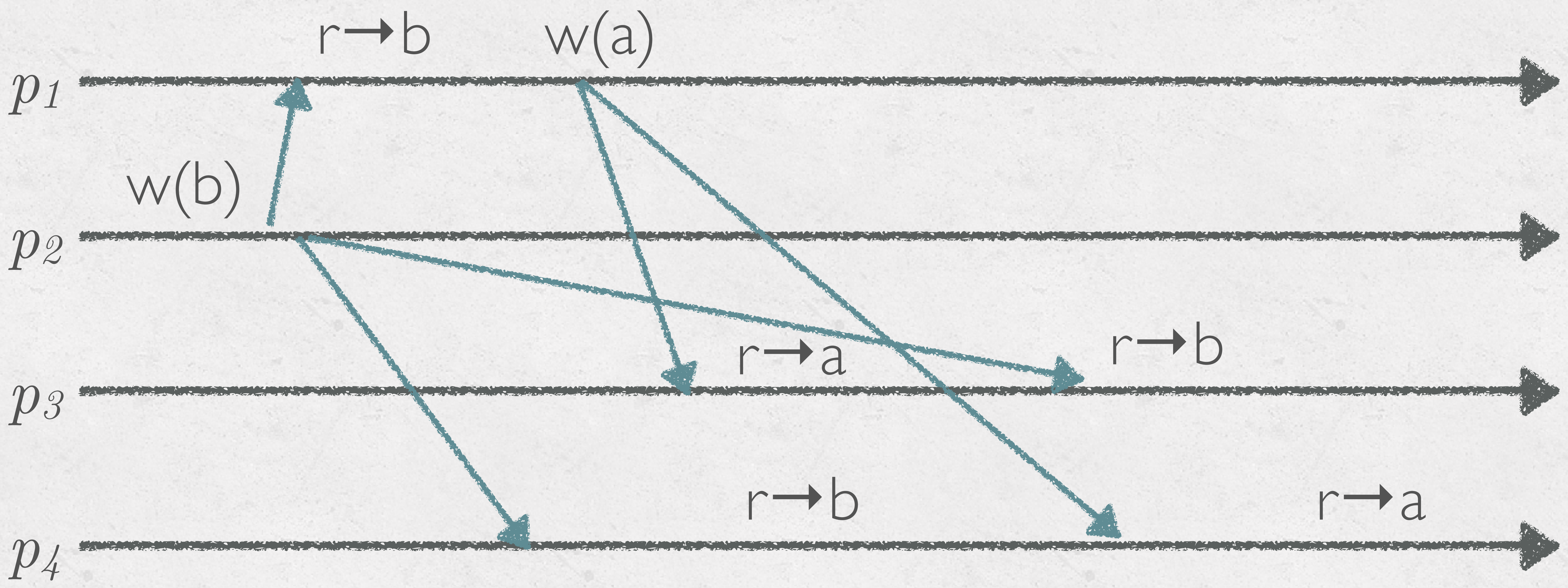
Is It Causal?



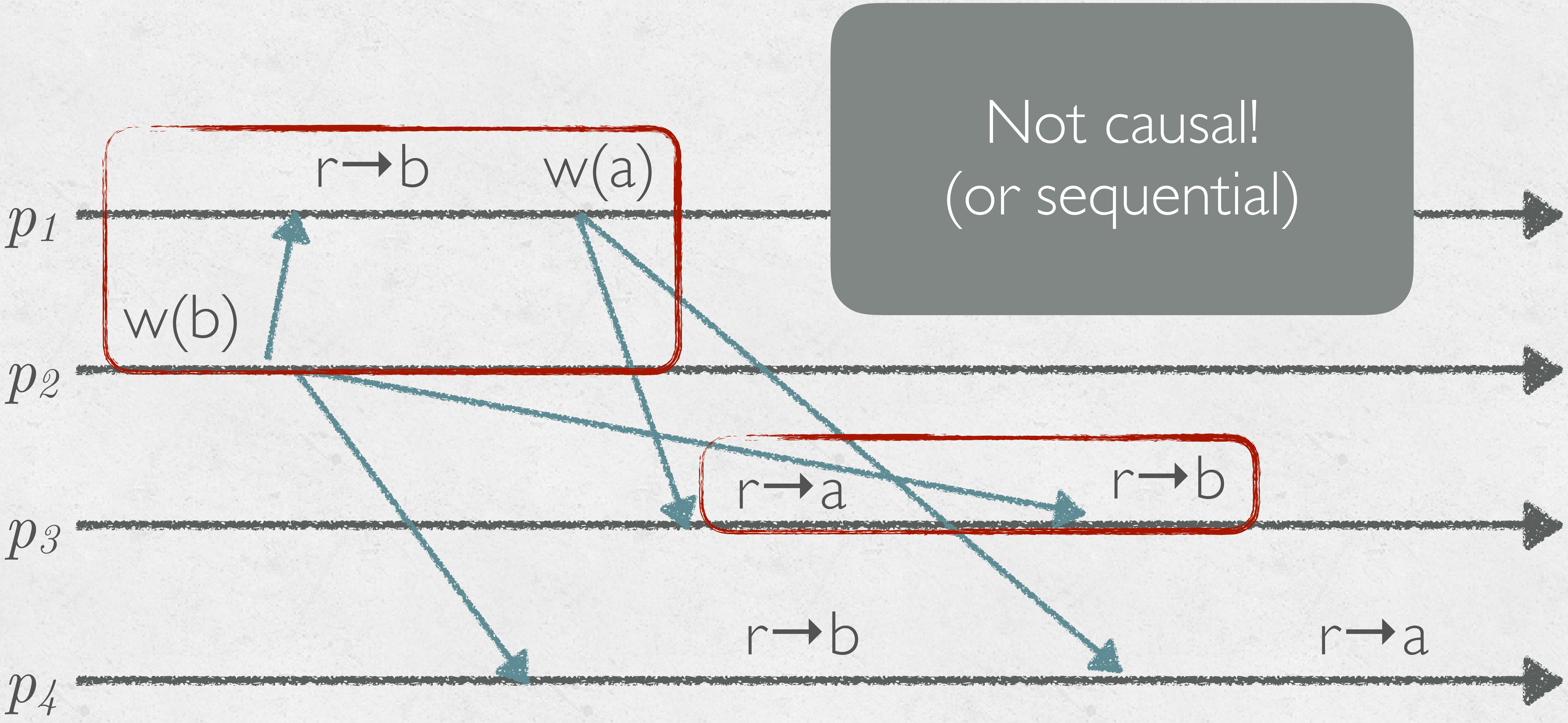
Is It Causal?



IS IT CAUSAL?



Is It Causal?



Cool Theorem: Causal consistency* is the strongest form of consistency that can be provided in an always-available convergent system.

Basically, if you want to process writes even in the presence of network partitions and failures, causal consistency is the best you can do.

[Mahajan et al. UTCS TR-11-22]

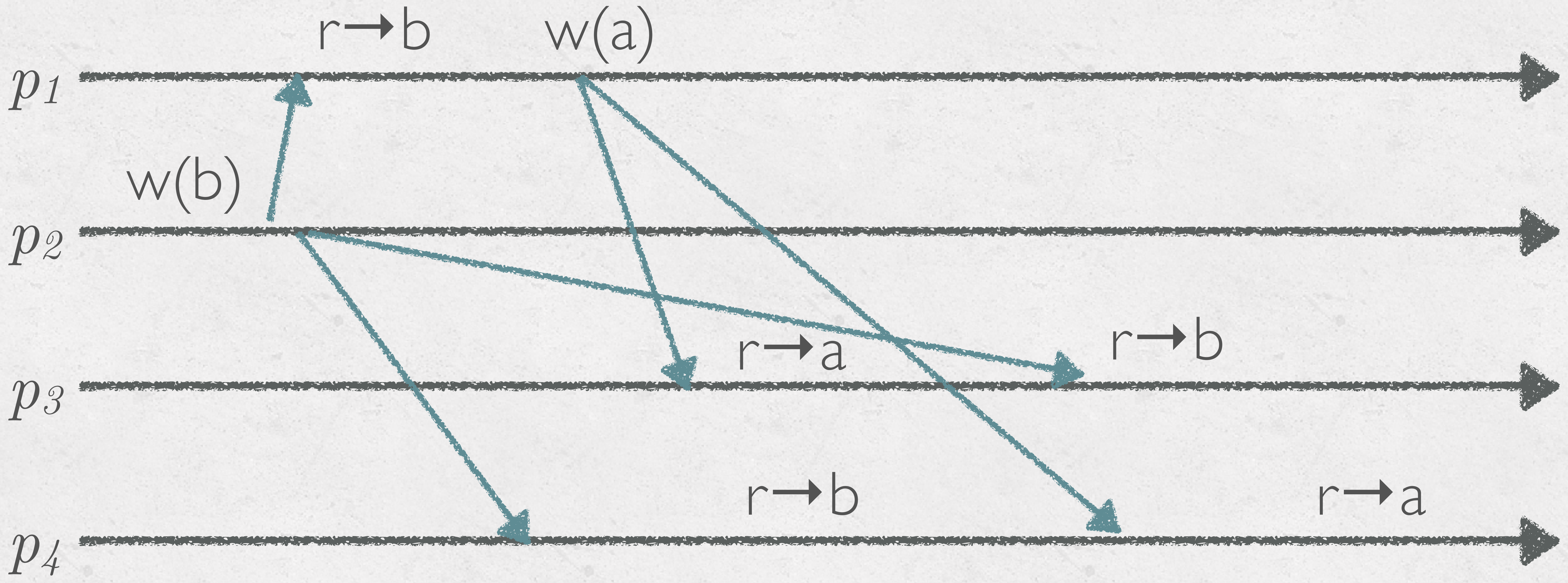
*real-time causal consistency

We Can Get Weaker!

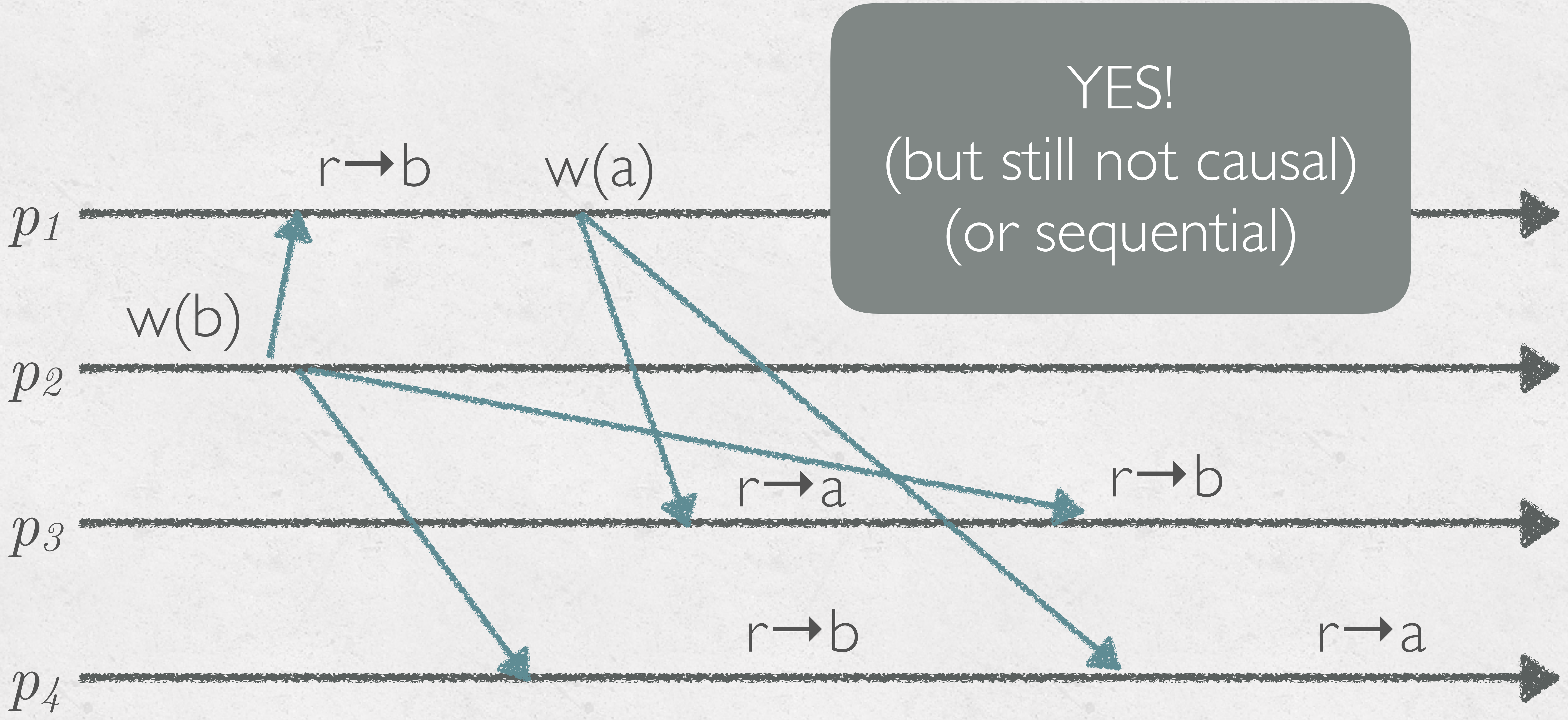
- **FIFO Consistency:** writes done by the same process are seen in that order; writes to different processes can be seen in different orders. Equivalent to the PRAM model.

- **Eventual Consistency** ≈ if all writes to an object stop, eventually all processes read the same value. (Not even a safety property! "Eventual consistency is no consistency.")

Is It FIFO?

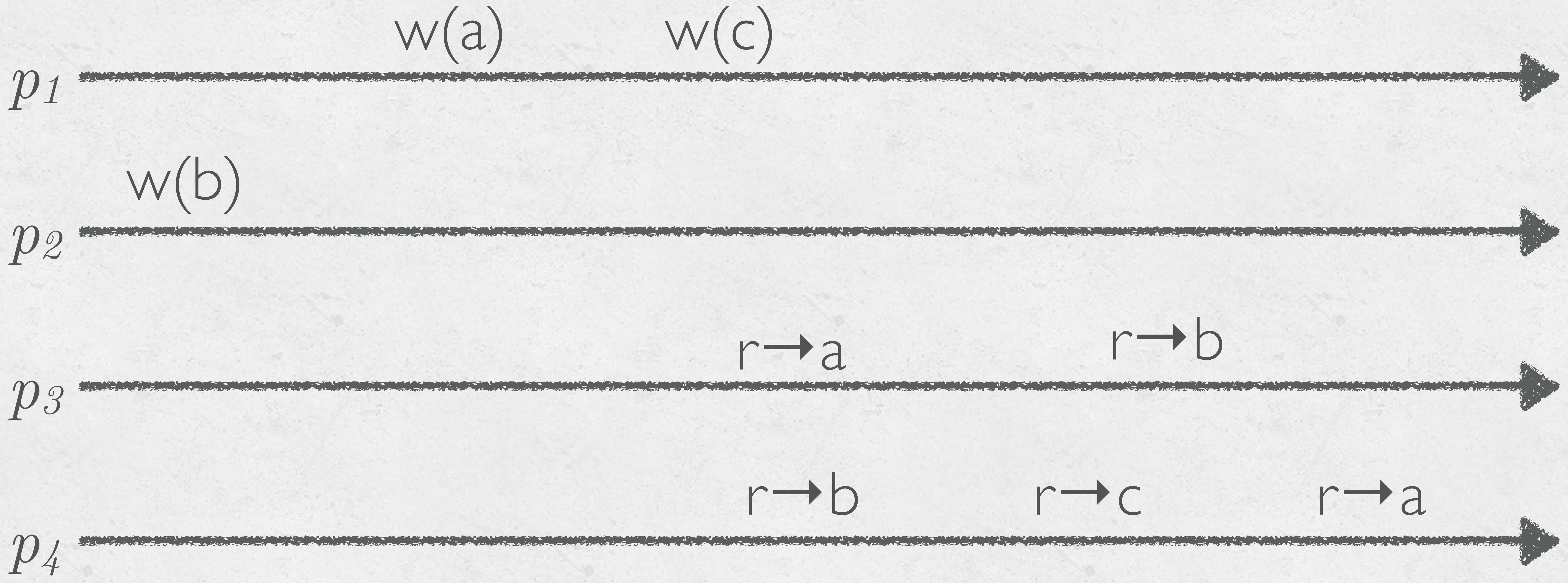


Is It FIFO?



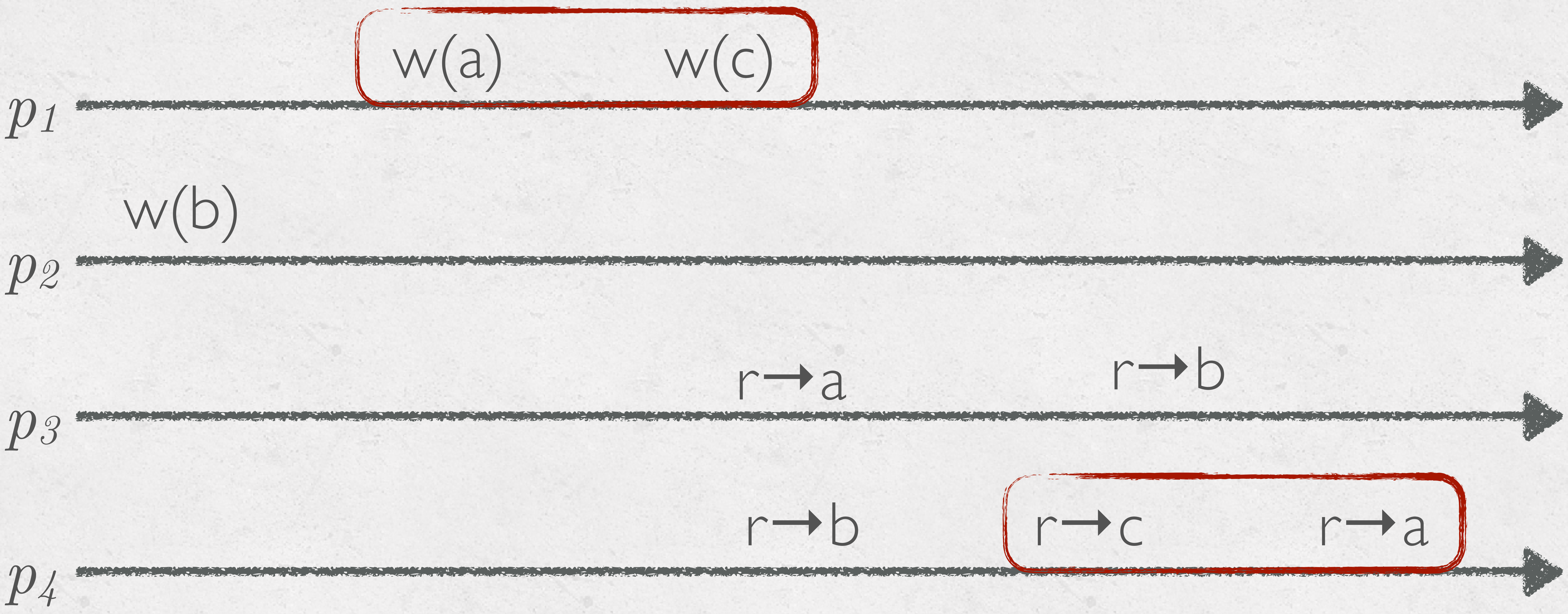
YES!
(but still not causal)
(or sequential)
p1
p2
p3
p4
w(b)
r→b
w(a)
r→a
r→b
r→b
r→a

Is It FIFO?



Is It FIFO?

Not FIFO!



Lamport's register semantics, sequential consistency, linearizability, and causal consistency, and FIFO consistency are all *safety properties*.

Using Consistency Guarantees

Thread 1

```
a = 1
print("b:" + b)
```

Thread 2

```
b = 1
print("a:" + a)
```

Initially, both a and b are 0.

What are the possible outputs of this program?

USING CONSISTENCY GUARANTEES

Thread 1

```
a = 1
print("b:" + b)
```

Thread 2

```
b = 1
print("a:" + a)
```

Depends on memory consistency!

Initially, both a and b are 0.

What are the possible outputs of this program?

Using Consistency Guarantees

Thread 1

```
a = 1
print("b:" + b)
```

Thread 2

```
b = 1
print("a:" + a)
```

Suppose both `prints` output 0.

Using Consistency Guarantees

Thread 1

```
a = 1
print("b:" + b)
```

Thread 2

```
b = 1
print("a:" + a)
```

Suppose both `prints` output 0.

Then there's a cycle in the happens-before graph.
Not sequential!

Aside: Java's Memory Model

- Java is **not** sequentially consistent!
- It guarantees sequential consistency only when the program is *data-race free*.
- A **data-race** occurs when two threads access the same memory location concurrently, one of the accesses is a write, and the accesses are not protected by locks (or monitors etc.).

A Common (incorrect) Idiom

```
class Foo {
  private Bar bar = null;

  public void baz() {
    if (bar == null) {
      synchronized(this) {
        if (bar == null) {
          bar = new Bar();
        }
      }
    }
    bar.doAThing();
  }
}
```

A Common (incorrect) Idiom, Corrected

```
class Foo {
  private volatile Bar bar = null;

  public void baz() {
    if (bar == null) {
      synchronized(this) {
        if (bar == null) {
          bar = new Bar();
        }
      }
    }
    bar.doAThing();
  }
}
```

volatile = accesses are sequentially consistent

A Common (incorrect) Idiom, Corrected

```
class Foo {
  private volatile Bar bar = null;

  public void baz()
    if (bar == nul
      synchronized
        if (bar ==
          bar = ne
        }
      }
    }
    bar.doAThing();
  }
}
```

e = accesses are
ially consistent

Reminder: you don't need to worry about multi-threaded access for the labs!

(except not grabbing locks in equals and hashCode)

How to Use Weak Consistency?

- Separate operations with stronger semantics, weak consistency (and high performance) by default
- Application-level protocols, either using separate communication, or extra synchronization variables in the data store (not always possible)

Main Takeaways

- The weaker the consistency model, the harder it is to program against (usually).
- The stronger the model, the harder it is to enforce (again, usually).