Distributed Systems 1

CUCS Course 4113
https://systems.cs.columbia.edu/ds1-class/

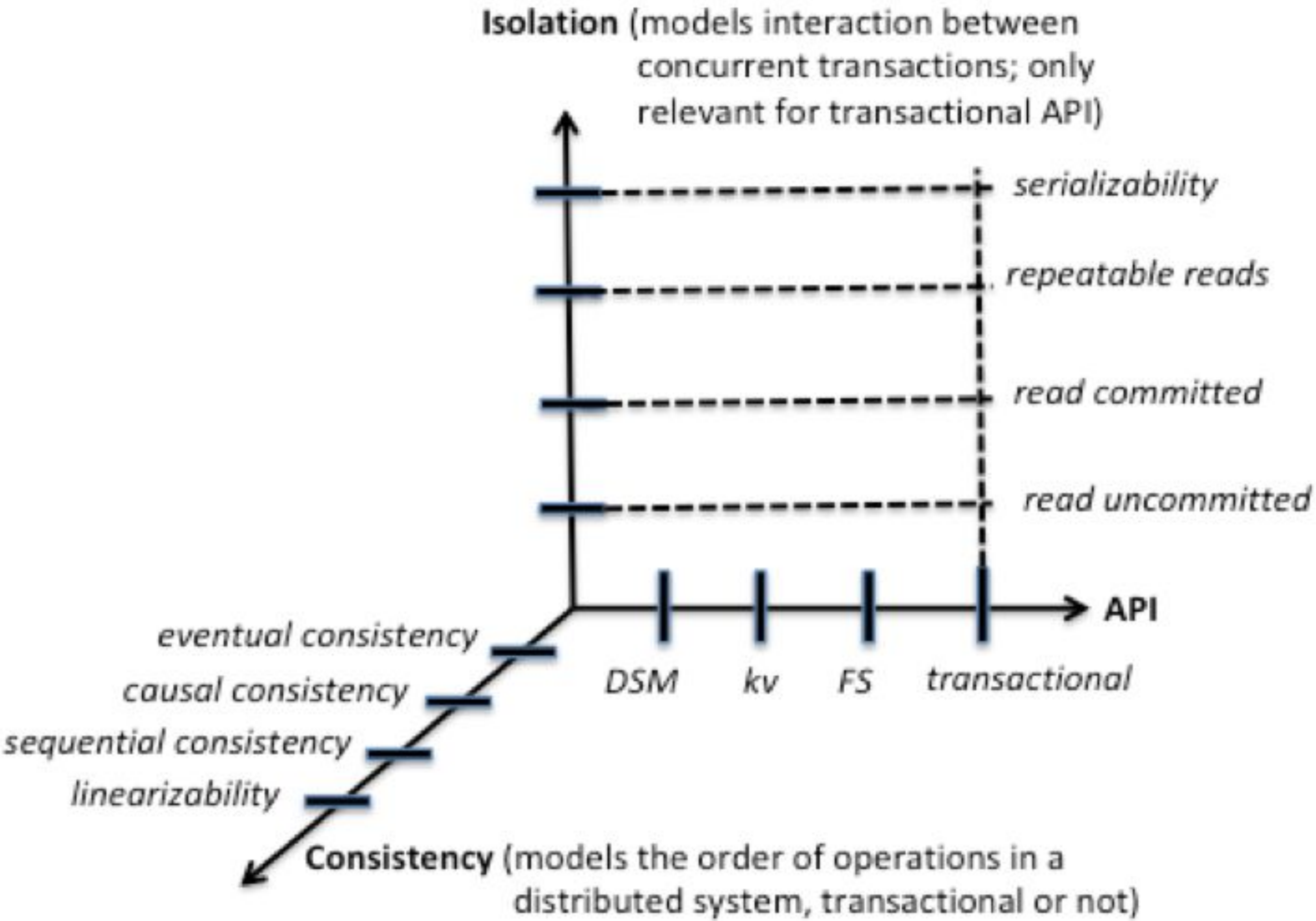
Instructor: Roxana Geambasu

Broader View of Isolation and Consistency Semantics

Context

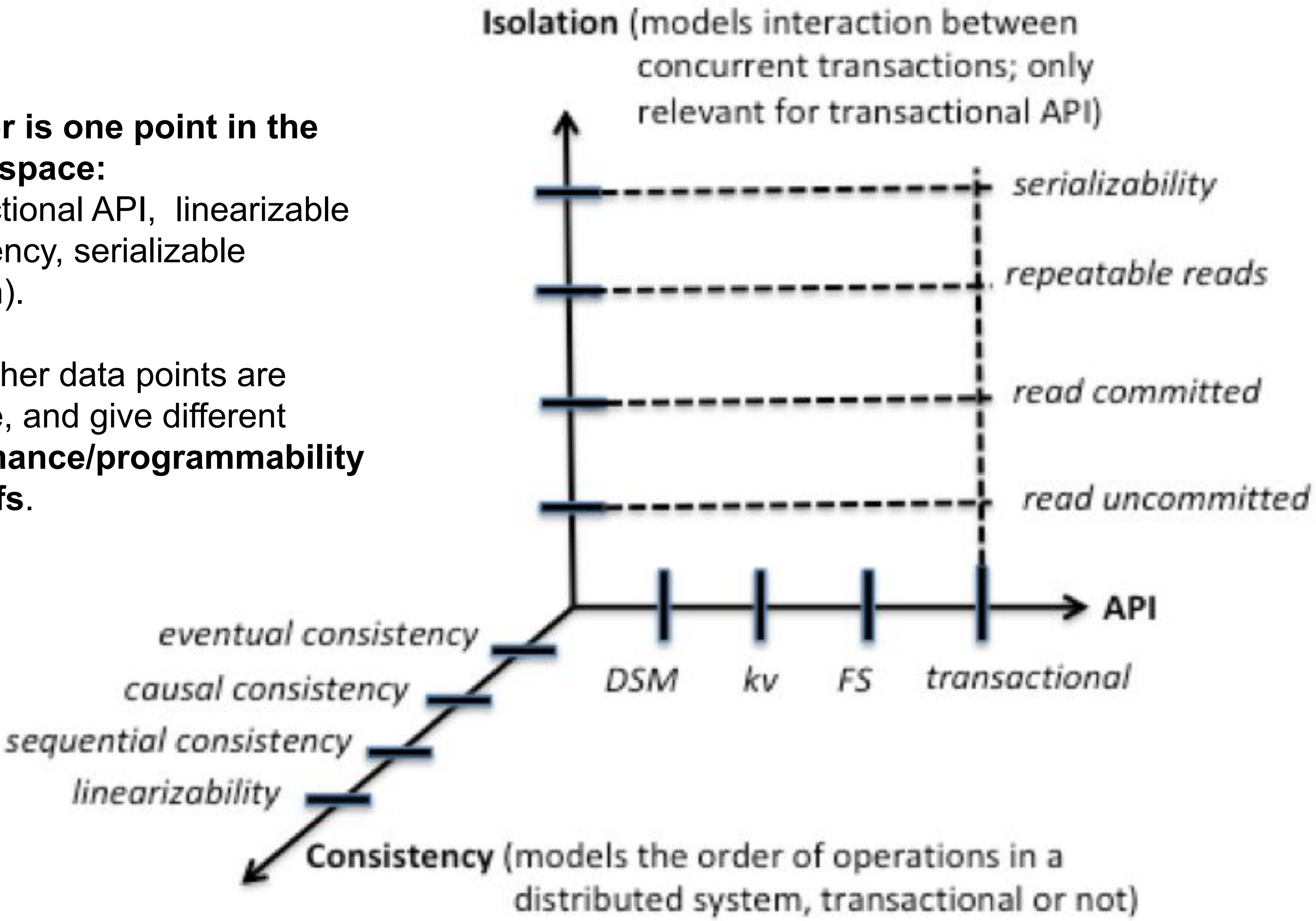
- We talked about protocols for building DSes with “strong semantics” that mimic the gold standard behavior of a non-distributed, non-concurrent system.
  - **Serializable isolation**: transactions appear to be executed sequentially, i.e., their individual ops don’t interfere.
  - **Linearizable consistency**: across all replicas, transactions appear to be executed in an order consistent with the real time ordering of their *commit* and *begin* operations.

- Spanner satisfies strong semantics at expense of performance.
- This tradeoff is not always suitable, so today we discuss a broader spectrum of well-understood, useful semantics.


Isolation (models interaction between concurrent transactions; only relevant for transactional API)
serializability
repeatable reads
read committed
read uncommitted
API
DSM
kv
FS
transactional
eventual consistency
causal consistency
sequential consistency
linearizability
Consistency (models the order of operations in a distributed system, transactional or not)

Spanner is one point in the design space:
(transactional API, linearizable consistency, serializable isolation).

Many other data points are possible, and give different **performance/programmability tradeoffs**.


Isolation (models interaction between concurrent transactions; only relevant for transactional API)
serializability
repeatable reads
read committed
read uncommitted
API
DSM
kv
FS
transactional
eventual consistency
causal consistency
sequential consistency
linearizability
Consistency (models the order of operations in a distributed system, transactional or not)

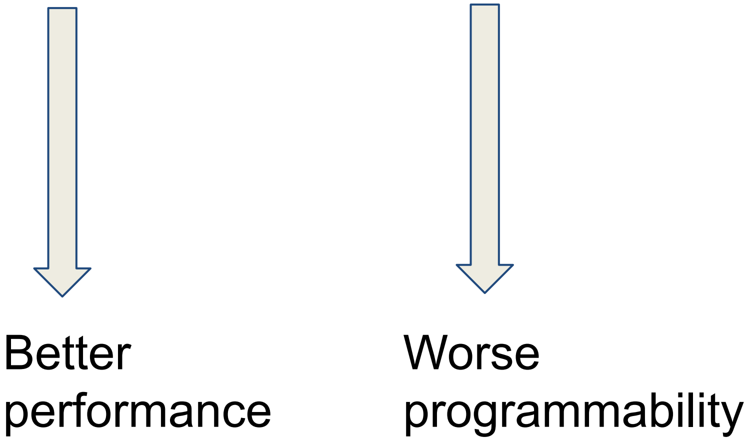
Isolation Semantics (a.k.a., Isolation Levels)

Isolation Semantics

- Relevant only for transactional APIs.
- Define how concurrent transactions interact with each other, i.e., whether individual effects of ongoing transactions can be witnessed by other transactions or not.
- Gold standard isolation is **serializability**: transactions are completely isolated from each other. For this, the DB engine must serialize conflicting transactions, which is expensive.
- Other isolation levels exist that offer weaker semantics (and hence more corner cases to consider when programming against them) but better performance.

Best Known Isolation Levels

- Serializability
- Repeatable reads
- Read committed
- Read uncommitted



Better performance

Worse programmability

- Easiest to remember them by thinking about locks that can be avoided/taken for shorter periods of time to gradually improve performance. But lockless implementations also exist.

Brief Descriptions

- **Serializability:**
  - Take r/w row-level and r range locks; keep them for entire transaction.
  - Ensures all conflicting, concurrent transactions are isolated from each other.
- **Repeatable reads:**
  - Take r/w row-level locks, keep them for entire transaction. Do not take r range locks at all.
  - Ensures that all row-level reads are repeatable.
  - Anomalies: **phantom reads** (concurrent Tx adds/removes row relevant to another transaction’s range query).

Brief Descriptions

- **Read committed:**
  - Take w row-level locks, keep them for entire transaction. Take r row-level locks, keep them only while row is read. No range locks.
  - Ensures that only committed updates are read.
  - Anomalies: phantom reads + **non-repeatable reads** (you may read a row that's being updated by another concurrent transaction, so depending on when you read that, the output may be different).
- **Read uncommitted:**
  - Take w row-level locks, keep them for entire transaction. No r locks, row-level or range-level.
  - Ensures that rows are atomically written.
  - Anomalies: phantom reads + non-repeatable reads + **dirty reads** (you may read a write of an in-process transaction that may ultimately be aborted).

Comparisons

- Anomalies make it harder and harder for programmers to reason about behavior of DB.
- But less synchronization leads to better performance (this is true even in lockless implementations).
- Typically, default in commercial databases (e.g., Oracle, SQL Server, PostgreSQL, MySQL) is read committed.

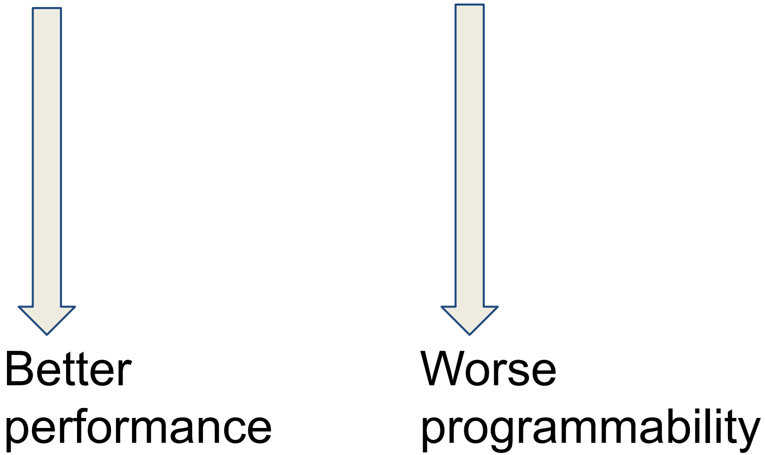
Consistency Semantics (a.k.a. Consistency Models)

Consistency Semantics

- Relevant for both Tx and non-Tx APIs. We'll focus on non-Tx API.
- Constrain the order in which individual operations (or individual transactions for a Tx API) are witnessed by different readers.
- Gold standard is **linearizability**: operations are seen in the real time order in which they are "committed" (finished). For this, the storage system must coordinate among replicas/shards, wait out clock uncertainty, etc. -- all of which can be very expensive.
- Other consistency models exist that offer weaker semantics (and hence more corner cases to consider when programming against them) but better performance, scalability, and sometimes availability.

Best Known Consistency Models

- Strict consistency
- Linearizability
- Sequential consistency
- Causal consistency
- Eventual consistency

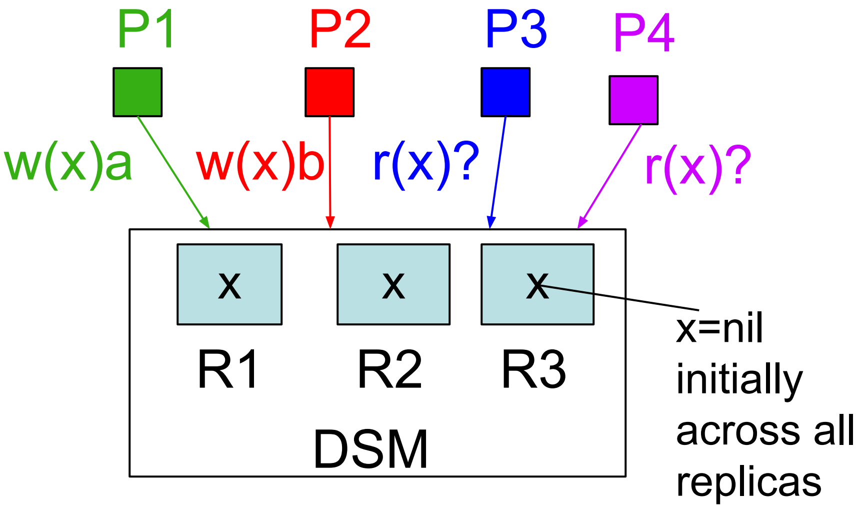


Better performance

Worse programmability

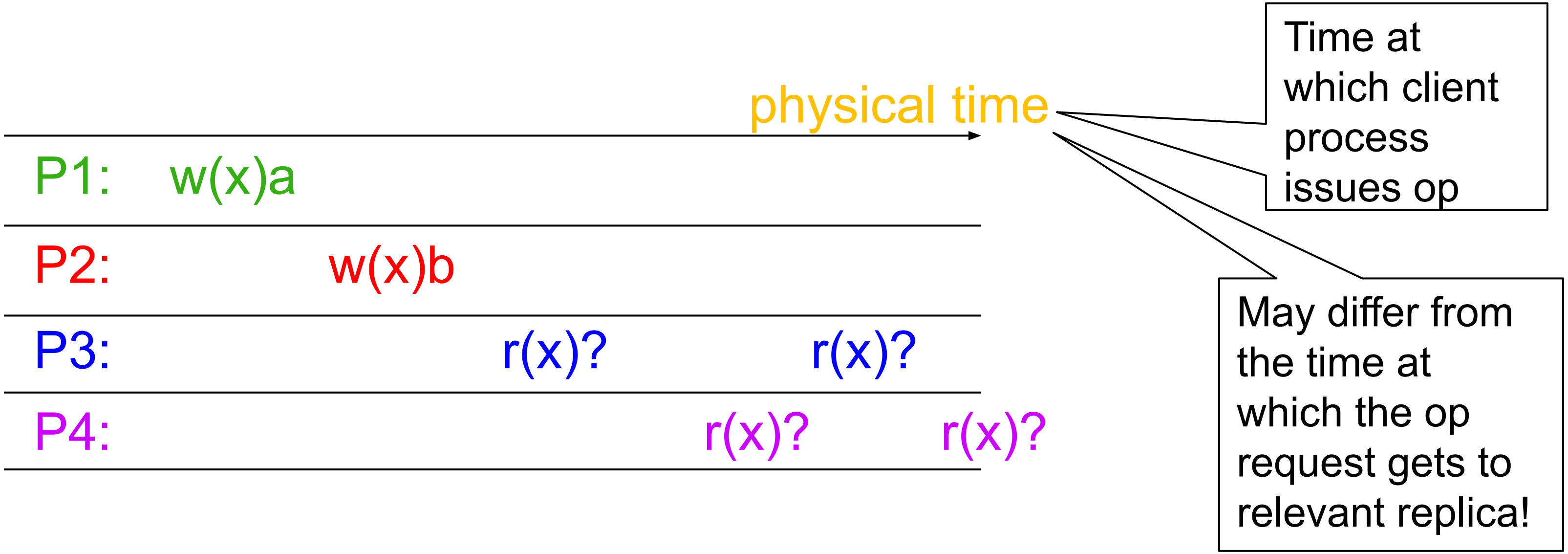
- Variations boil down to: (1) the allowable staleness of reads and (2) the ordering of writes across all replicas.

Examples with Replicated Distributed Shared Memory (DSM)



- To define consistency, we must define what values of reads are admissible by the DSM.
- The semantic restricts *all* possible executions.
- In the slides, we will use individual examples to show what's admissible vs. not for a given semantic.

Structure of an Example



physical time

P1: w(x)a

P2: w(x)b

P3: r(x)? r(x)?

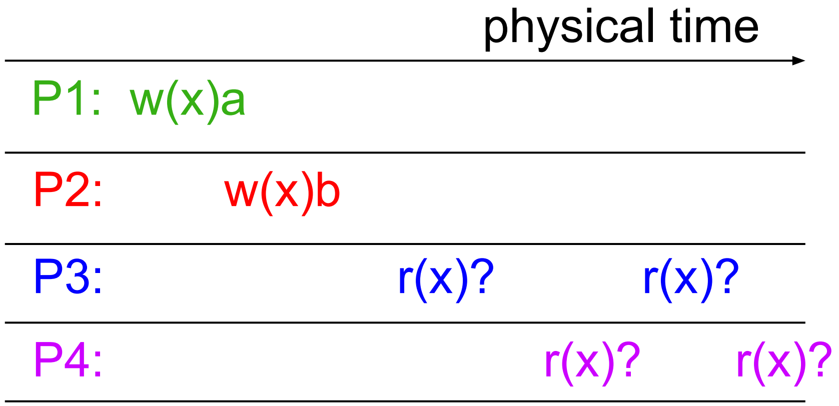
P4: r(x)? r(x)?

Time at which client process issues op

May differ from the time at which the op request gets to relevant replica!

Strict Consistency

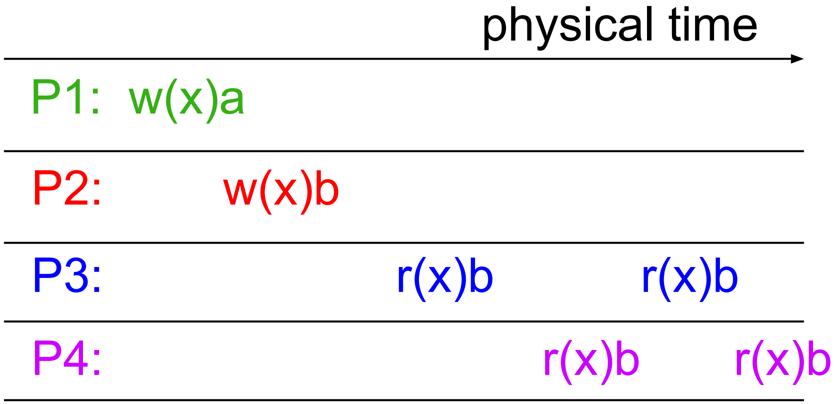
- **Defn:** Any execution is the same as if all read/write ops were executed in order of physical time at which they were **issued**.
- **Therefore:** (1) Reads are never stale; (2) all replicas enforce physical-time ordering for all writes.



if DSM is strictly consistent, what can these reads return?

Strict Consistency

- **Defn:** Any execution is the same as if all read/write ops were executed in order of physical time at which they were **issued**.
- **Therefore:** (1) Reads are never stale; (2) all replicas enforce physical-time ordering for all writes.



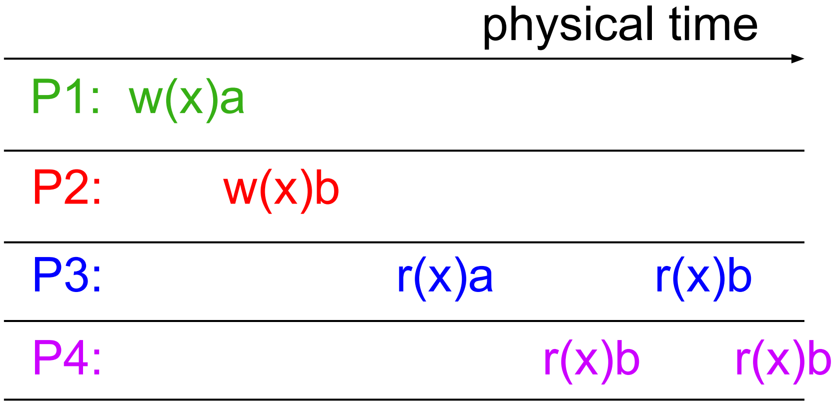
physical time

P1: w(x)a

P2: w(x)b

P3: r(x)b r(x)b

P4: r(x)b r(x)b



physical time

P1: w(x)a

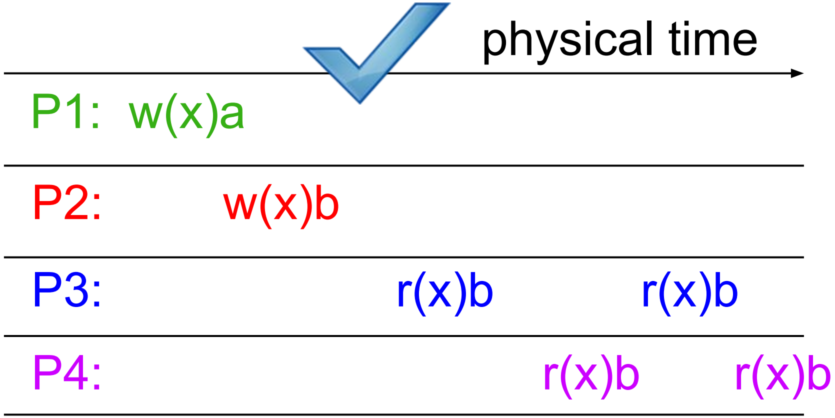
P2: w(x)b

P3: r(x)a r(x)b

P4: r(x)b r(x)b

Strict Consistency

- **Defn:** Any execution is the same as if all read/write ops were executed in order of physical time at which they were **issued**.
- **Therefore:** (1) Reads are never stale; (2) all replicas enforce physical-time ordering for all writes.



✓

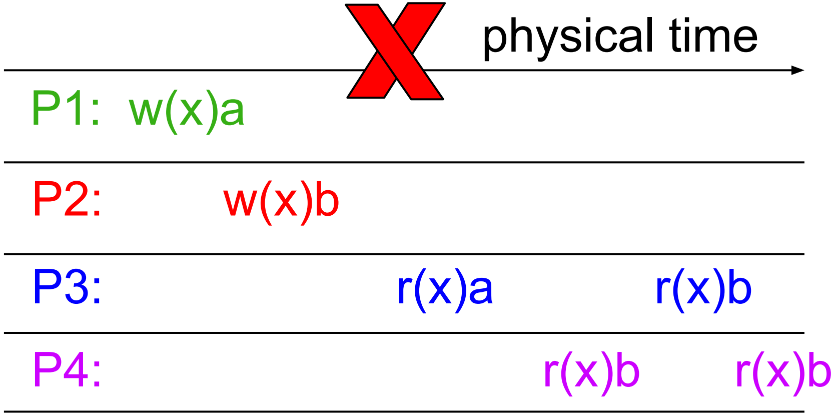
physical time

P1: w(x)a

P2: w(x)b

P3: r(x)b r(x)b

P4: r(x)b r(x)b



✗

physical time

P1: w(x)a

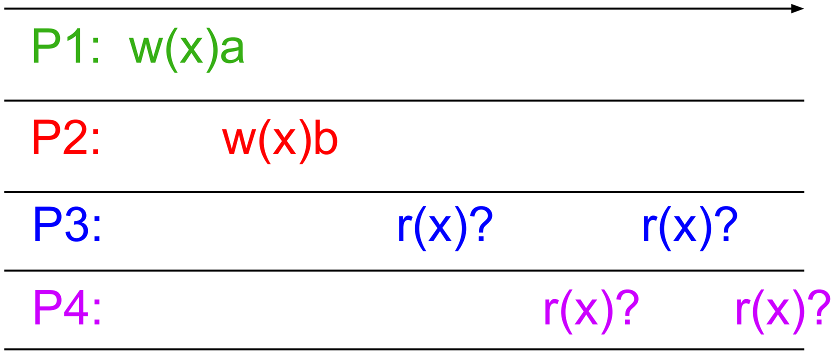
P2: w(x)b

P3: r(x)a r(x)b

P4: r(x)b r(x)b

Linearizability

- **Defn:** Any execution is the same as if all read/write ops were executed in **some global order** s.t. any read returns the value of the **most recent completed write** at that location.
- **Therefore**: (1) Once a write completes, all later reads return the value of that write or of a later write. (2) Once a read returns a value, all later reads return that value or value of a later write.



P1: w(x)a

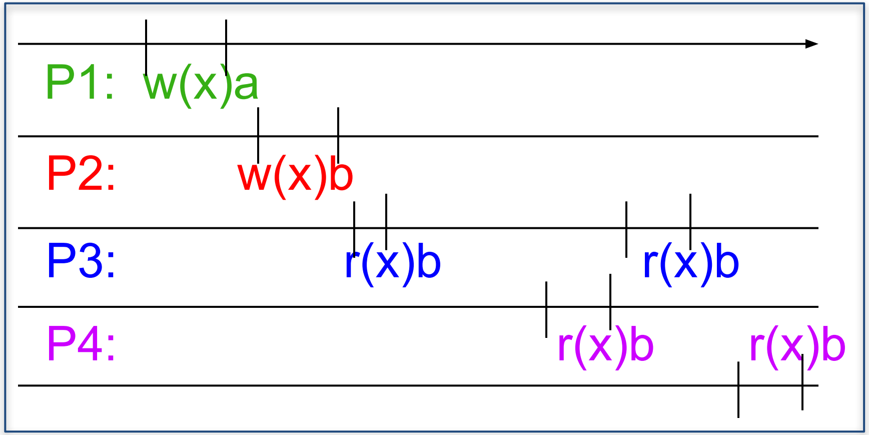
P2: w(x)b

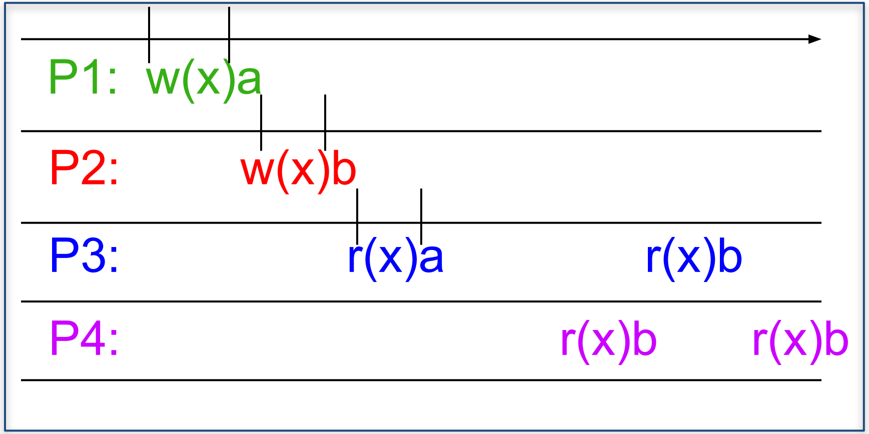
P3: r(x)? r(x)?

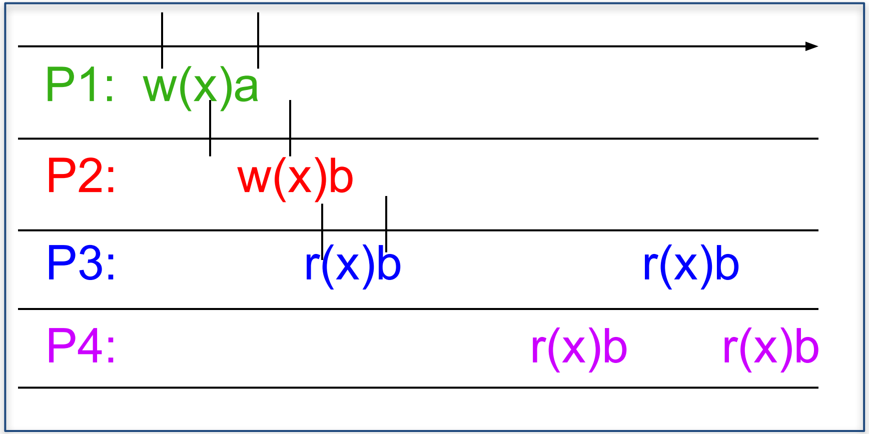
P4: r(x)? r(x)?

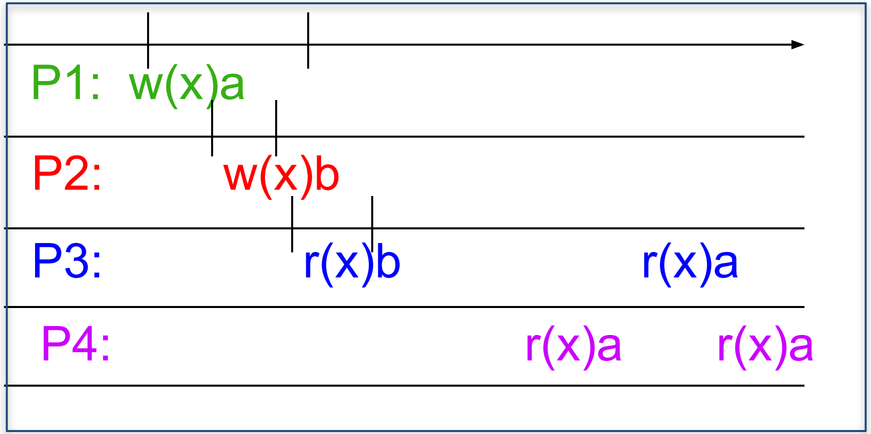
if DSM is linearizable, what can these reads return?

Linearizability

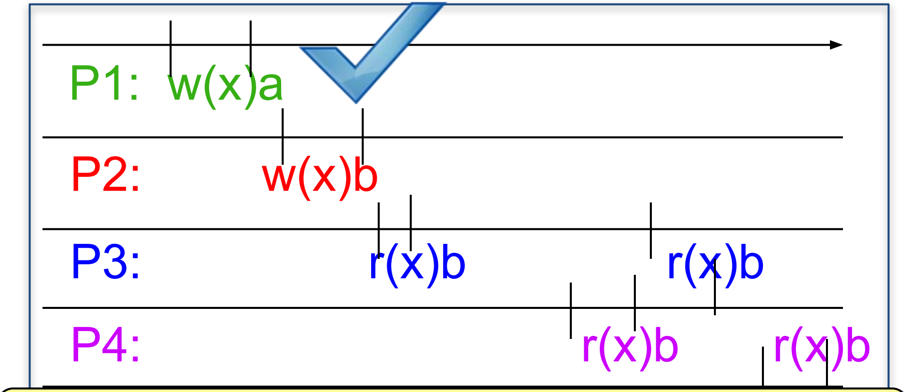




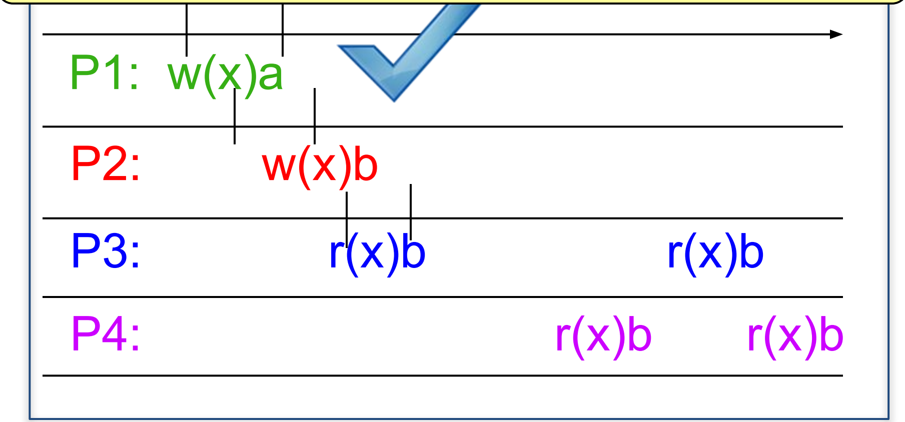


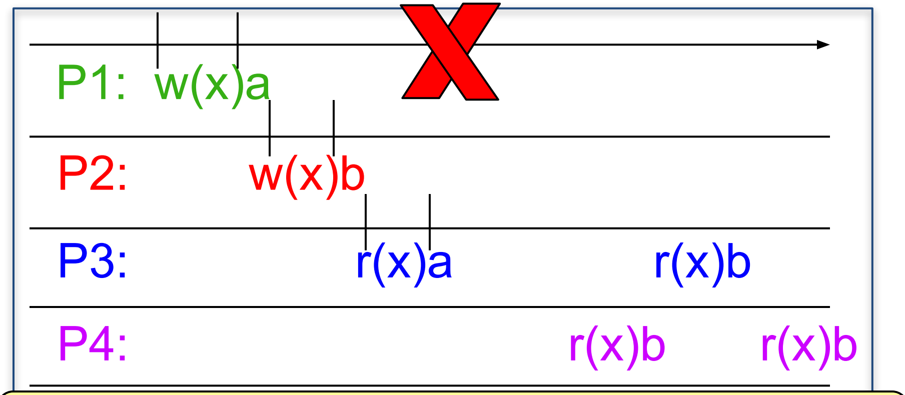


Linearizability

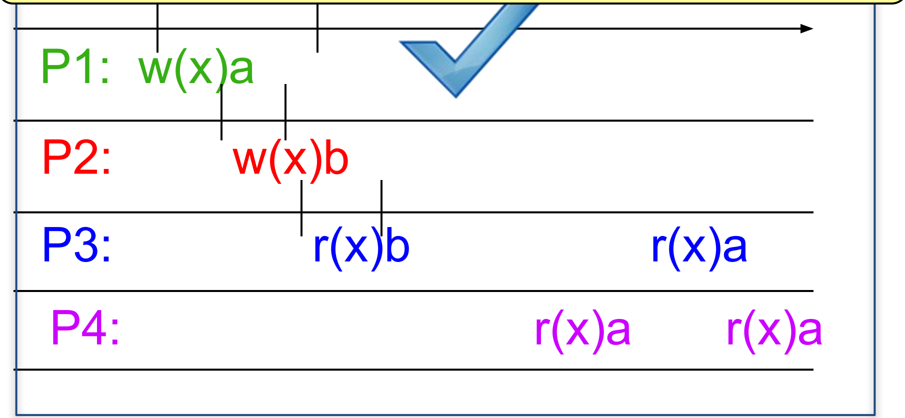


These are also strictly consistent





These aren’t strictly consistent

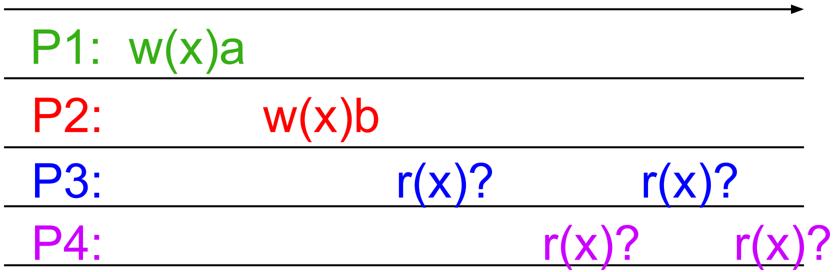


Implementability

- Strict consistency isn’t implementable, but linearizability is.
- Spanner guarantees linearizability for its operations (view each transaction as an operation) based on the commit waits and combo of protocols we discussed last time.
- The best way to describe Spanner’s semantic is that it gives serializable isolation with linearizability consistency guarantee.
  - This semantic is traditionally called strict serializability.
  - In the Spanner paper, they call it external consistency.
- But the linearizability semantic applies to non-transactional systems too (e.g., DSMs, key/value stores, file systems, ...).

Sequential Consistency

- **Defn:** Any execution is the same as if all read/write ops were executed in **some global order,** and the ops of each client process appear in the **order specified by its program.** (This global order that adheres to program order is called global sequential order.)
- **Therefore:** (1) Reads may be stale in real time, but not in logical time; (2) Writes are totally ordered according to logical time across all replicas.



P1: w(x)a

P2: w(x)b

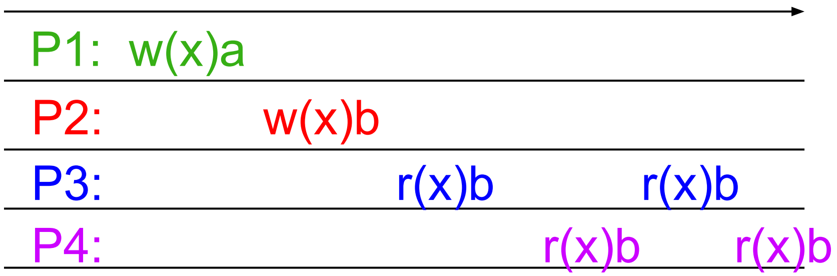
P3: r(x)? r(x)?

P4: r(x)? r(x)?

if DSM is sequentially consistent, what can these reads return?

Sequential Consistency

- **Defn:** Any execution is the same as if all read/write ops were executed in **some global order,** and the ops of each client process appear in the **order specified by its program.**

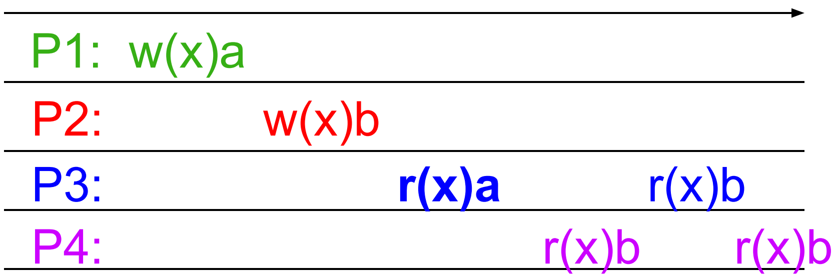


P1: w(x)a

P2: w(x)b

P3: r(x)b r(x)b

P4: r(x)b r(x)b



P1: w(x)a

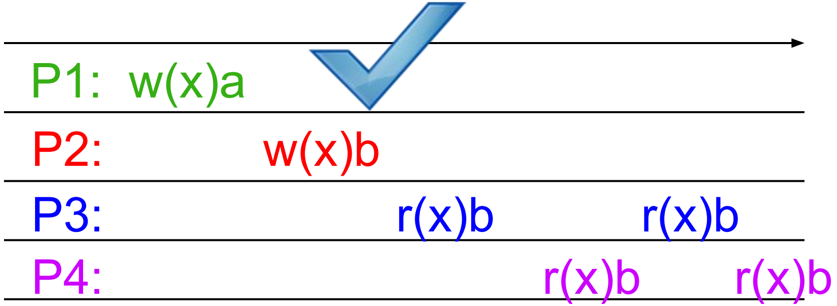
P2: w(x)b

P3: **r(x)a** r(x)b

P4: r(x)b r(x)b

Sequential Consistency

- **Defn:** Any execution is the same as if all read/write ops were executed in **some global order,** and the ops of each client process appear in the **order specified by its program.**



P1: w(x)a

P2: w(x)b

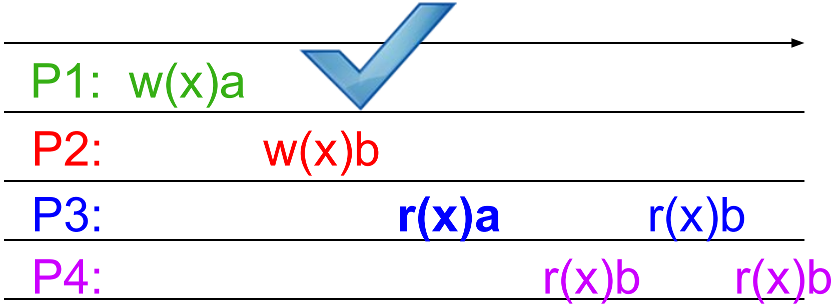
P3: r(x)b r(x)b

P4: r(x)b r(x)b

What's a global sequential order that can explain these results?

physical-time ordering

This was also linearizable



P1: w(x)a

P2: w(x)b

P3: **r(x)a** r(x)b

P4: r(x)b r(x)b

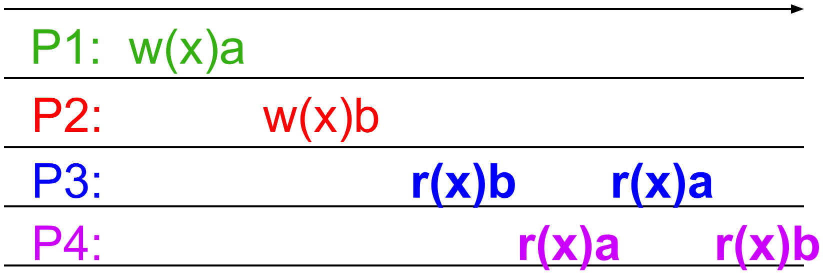
What's a global sequential order that can explain these results?

w(x)a, r(x)a, w(x)b, r(x)b, …

This wasn't linearizable

Sequential Consistency

- **Defn:** Any execution is the same as if all read/write ops were executed in **some global order,** and the ops of each client process appear in the **order specified by its program.**

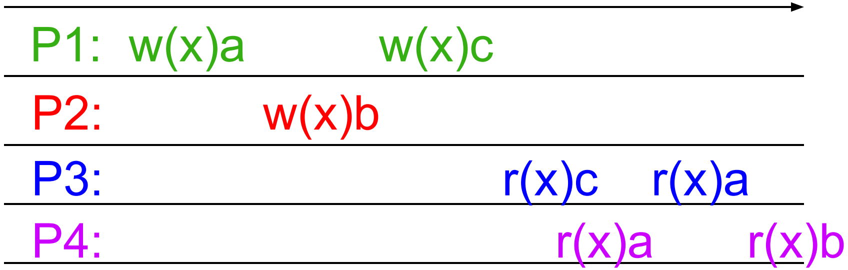


P1: w(x)a

P2: w(x)b

P3: **r(x)b** **r(x)a**

P4: **r(x)a** **r(x)b**



P1: w(x)a w(x)c

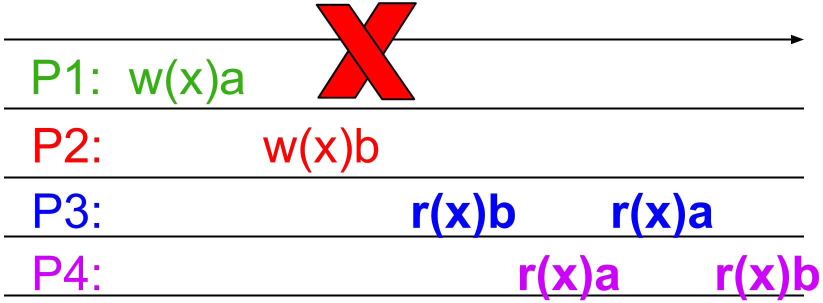
P2: w(x)b

P3: r(x)c r(x)a

P4: r(x)a r(x)b

Sequential Consistency

- **Defn:** Any execution is the same as if all read/write ops were executed in **some global order,** and the ops of each client process appear in the **order specified by its program.**



P1: w(x)a

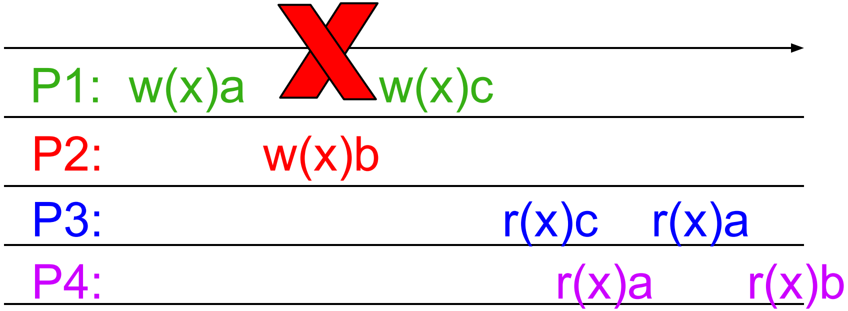
P2: w(x)b

P3: **r(x)b** **r(x)a**

P4: **r(x)a** **r(x)b**

No global order can explain these results…

=> not seq. consistent



P1: w(x)a w(x)c

P2: w(x)b

P3: r(x)c r(x)a

P4: r(x)a r(x)b

No global *sequential* order can explain results.

E.g.: the following global order doesn't preserve P1's ordering:

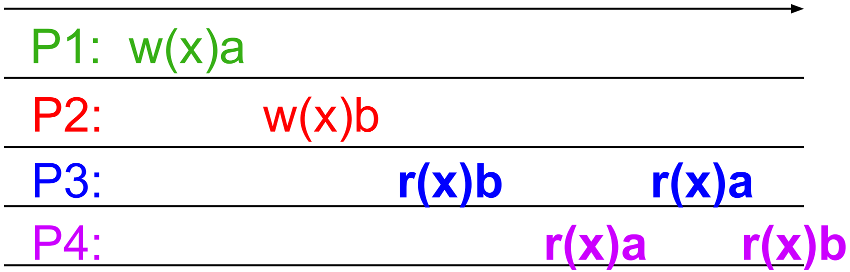
w(x)c, r(x)c, w(x)a, r(x)a, w(x)b, …

Causal Consistency

- **Defn**: Any execution is the same as if all **causally-related** read/write ops were executed in an **order that reflects their causality.**
  - All **concurrent** ops may be seen in different orders.
- **Therefore**: (1) Reads are fresh only w.r.t. the writes that they are causally dependent on; (2) Only causally-related writes are ordered by all replicas in the same way, but concurrent writes may be committed in different orders by different replicas, and hence read in different orders by different applications.

Causal Consistency

- **Defn**: Any execution is the same as if all **causally-related** read/write ops were executed in an **order that reflects their causality.**
  - All **concurrent** ops may be seen in different orders.

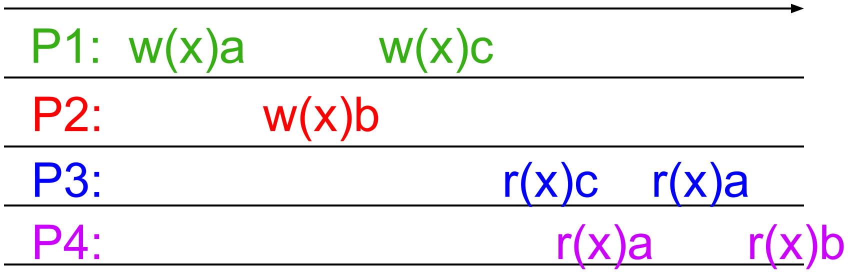


P1: w(x)a

P2: w(x)b

P3: **r(x)b** **r(x)a**

P4: **r(x)a** **r(x)b**



P1: w(x)a w(x)c

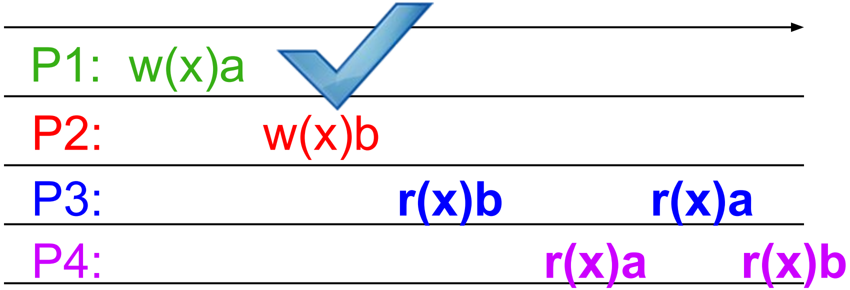
P2: w(x)b

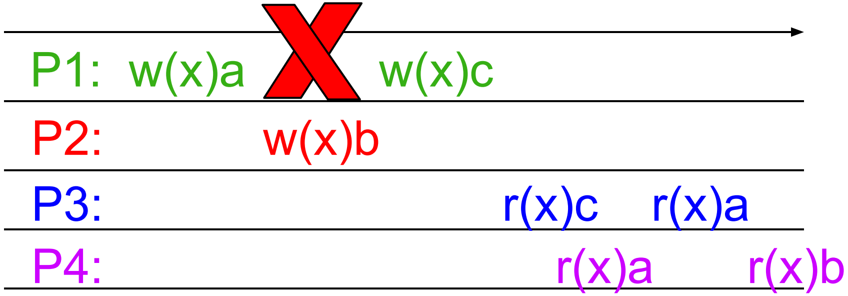
P3: r(x)c r(x)a

P4: r(x)a r(x)b

Causal Consistency

- **Defn**: Any execution is the same as if all **causally-related** read/write ops were executed in an **order that reflects their causality.**
  - All **concurrent** ops may be seen in different orders.





Only per-process ordering restrictions:

w(x)b < r(x)b; r(x)b < r(x)a; …

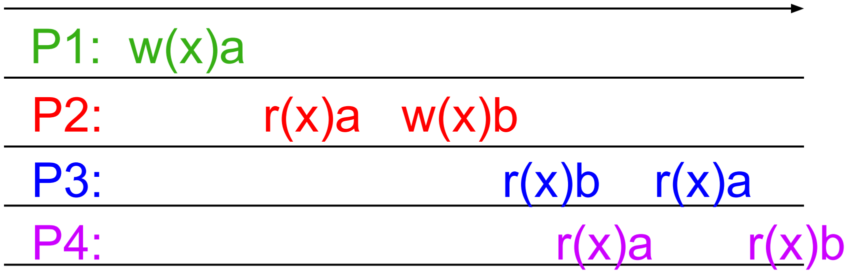
w(x)a || w(x)b, hence they can be seen

This wasn't sequentially consistent.

Having read c (r(x)c), P3 must continue to read c or some newer value (perhaps b), but can't go back to a, b/c w(x)c was conditional upon w(x)a having finished.

Causal Consistency

- **Defn**: Any execution is the same as if all **causally-related** read/write ops were executed in an **order that reflects their causality.**
  - All **concurrent** ops may be seen in different orders.



P1: w(x)a

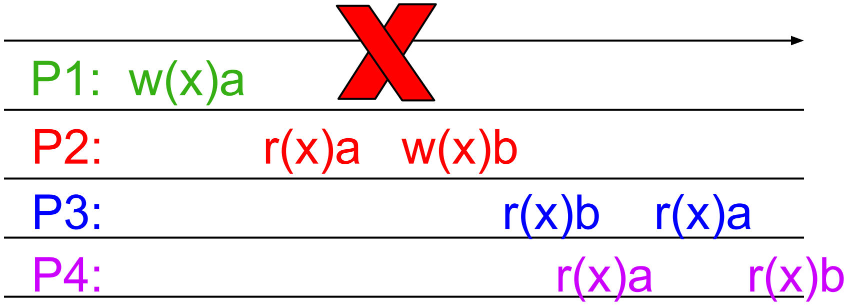
P2: r(x)a w(x)b

P3: r(x)b r(x)a

P4: r(x)a r(x)b

Causal Consistency

- **Defn**: Any execution is the same as if all **causally-related** read/write ops were executed in an **order that reflects their causality.**
  - All **concurrent** ops may be seen in different orders.



P1: w(x)a

P2: r(x)a w(x)b

P3: r(x)b r(x)a

P4: r(x)a r(x)b

w(x)b is causally-related on r(x)a, which is causally-related on w(x)a.
Therefore, system must enforce w(x)a < w(x)b ordering.
But P3 violates that ordering, b/c it reads a after reading b.

Why Causal Consistency?

- Causal consistency is **strictly weaker** than sequential consistency and can give **weird results**, as you've seen.
- BUT: it also requires less coordination, hence **better performance**.

- Note that in causally consistent systems, you don't actually ever have inversions of concurrent updates on the same object, it's very easy and efficient to prevent that.
  - But concurrent updates on different objects (e.g., w(x)5 || w(y)7) can be seen in different orders by different replicas.

Eventual Consistency

- Allow stale reads, but ensure that reads will **eventually** reflect previously written values, even after a long time.
- Doesn’t order writes as they are executed, which might create **conflicts** later: which write was first?
- Used in Amazon’s Dynamo, a key/value store
  - Plus a lot of academic systems
  - Plus file synchronization
  - Plus source control systems like… **git!**

Sequential vs. Eventual Consistency

- Sequential: **pessimistic** concurrency handling
  - Decide on update order as updates are executed

- Eventual: **optimistic** concurrency handling
  - Let updates happen, worry about deciding their order later
  - May raise **conflicts**
    - Think about when you code offline for a while – you may need to resolve conflicts with other team members when you commit
    - Resolving conflicts is not that difficult with code, but it’s really hard in general (e.g., think about resolving conflicts when two people update an image or a binary -- how to merge?)

Why (Not) Eventual Consistency?

- ✓ Supports disconnected operations or network partitions
  - Better to read a stale value than nothing
  - Better to save writes somewhere than nothing

- ✓ Supports for increased parallelism
  - But that’s not what people have typically used this for

- ● Can lead to anomalous application behavior
  - Stale reads and conflicting writes…

Many Other Consistency Models Exist

- Other standard consistency models
  - Monotonic reads
  - Monotonic writes
  - … read Tanenbaum 7.3 if interested (these are not required for exam)

- In-house consistency models:
  - Andrew File System’s close-to-open
  - Google File System’s atomic at-most-once appends

Useful References

Google's doc for Cloud Spanner:

https://cloud.google.com/spanner/docs/true-time-external-consistency