CS4740
CLOUD COMPUTING

Isolation and Consistency

Prof. Chang Lou, UVA CS, Spring 2024

GUEST TALK

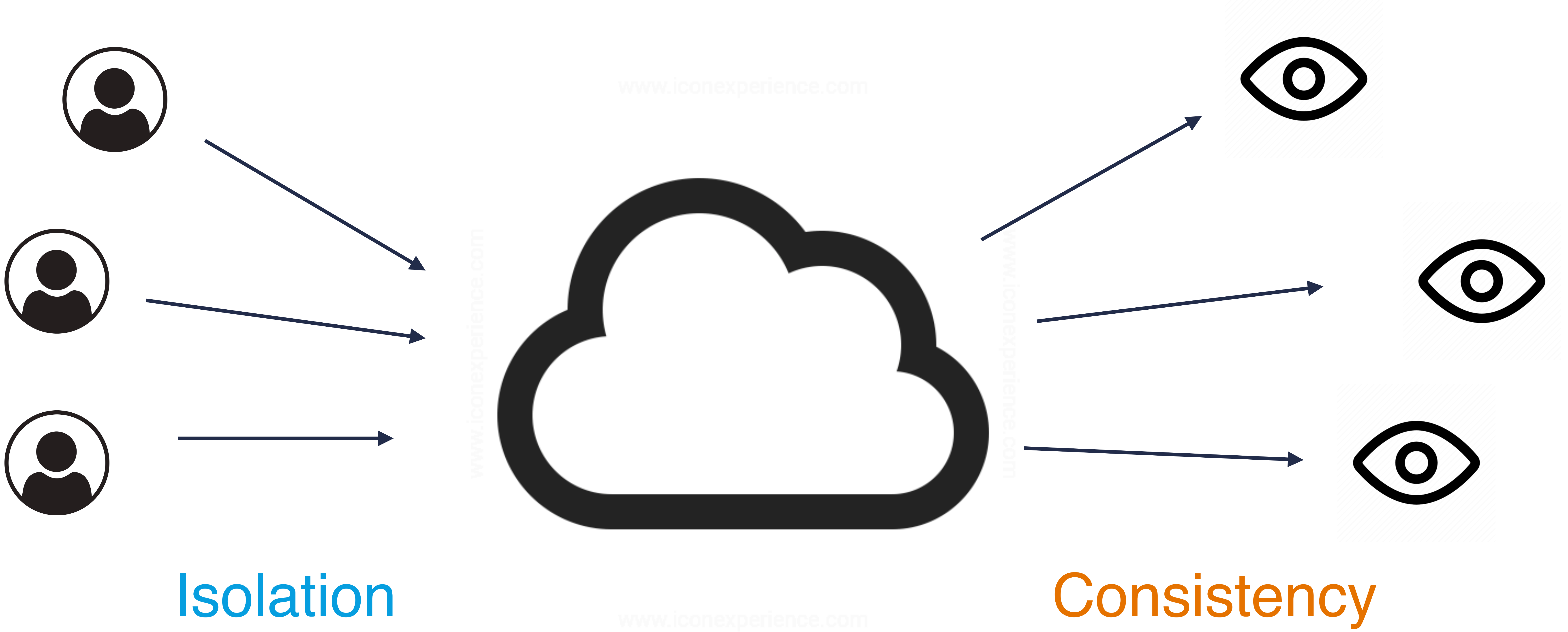
- **Block Store over the Cloud**
  - Speaker: Erci Xu, Alibaba Cloud
  - Date&Location: 3/25, next Mon class

Alibaba Cloud

- Erci Xu serves as a research scientist at Alibaba Cloud Storage, where his primary focus lies in the development of distributed storage systems and the enhancement of both software and hardware reliability. He has authored multiple papers in top conferences such as USENIX OSDI, FAST, ATC, and ACM Eurosys. He is the recipient of two USENIX FAST Best Paper Awards (FAST'23 and FAST'24) and 2023 ACM SIGOPS China Rising Star Award.

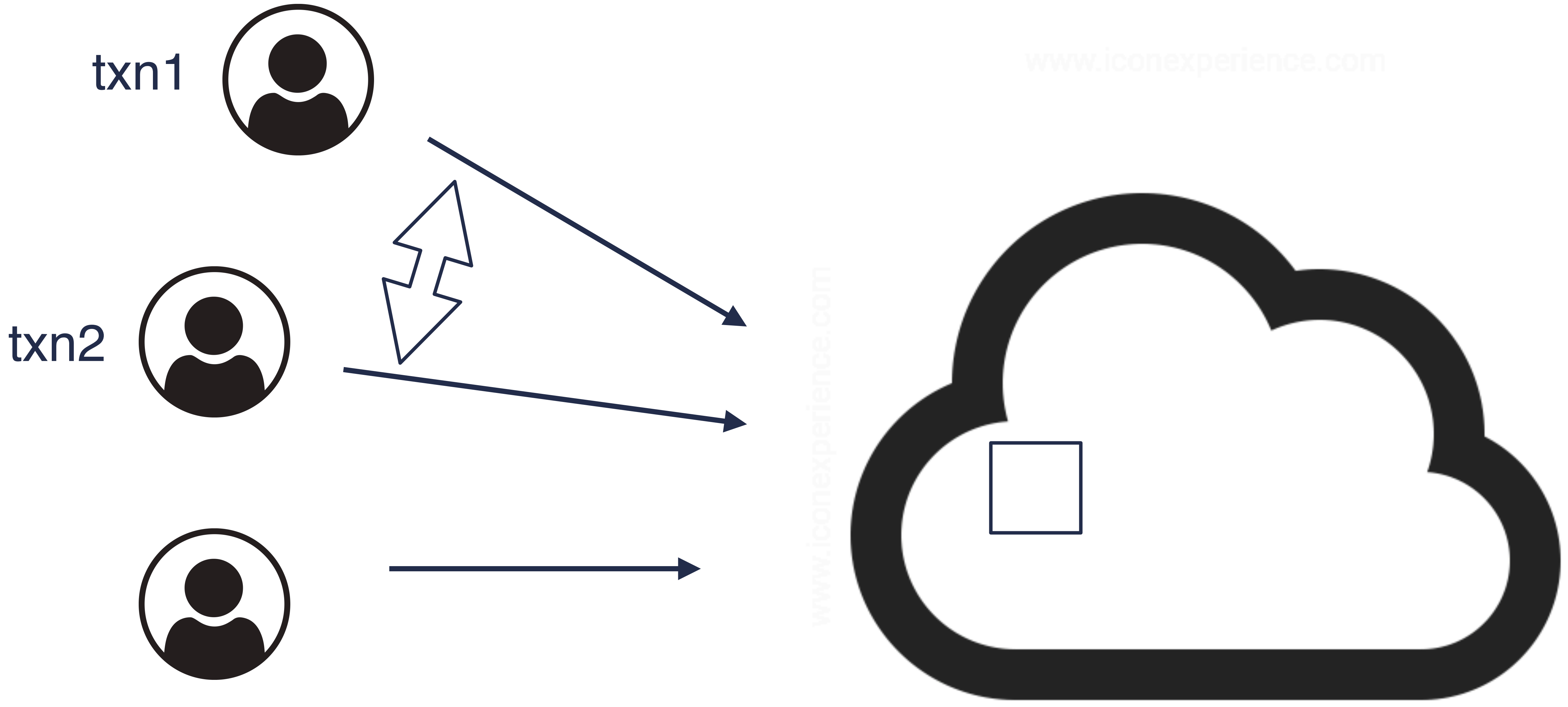
CONTEXT

- Today we talk about different levels of isolation and consistency
  - and what are the tradeoffs



CONTEXT

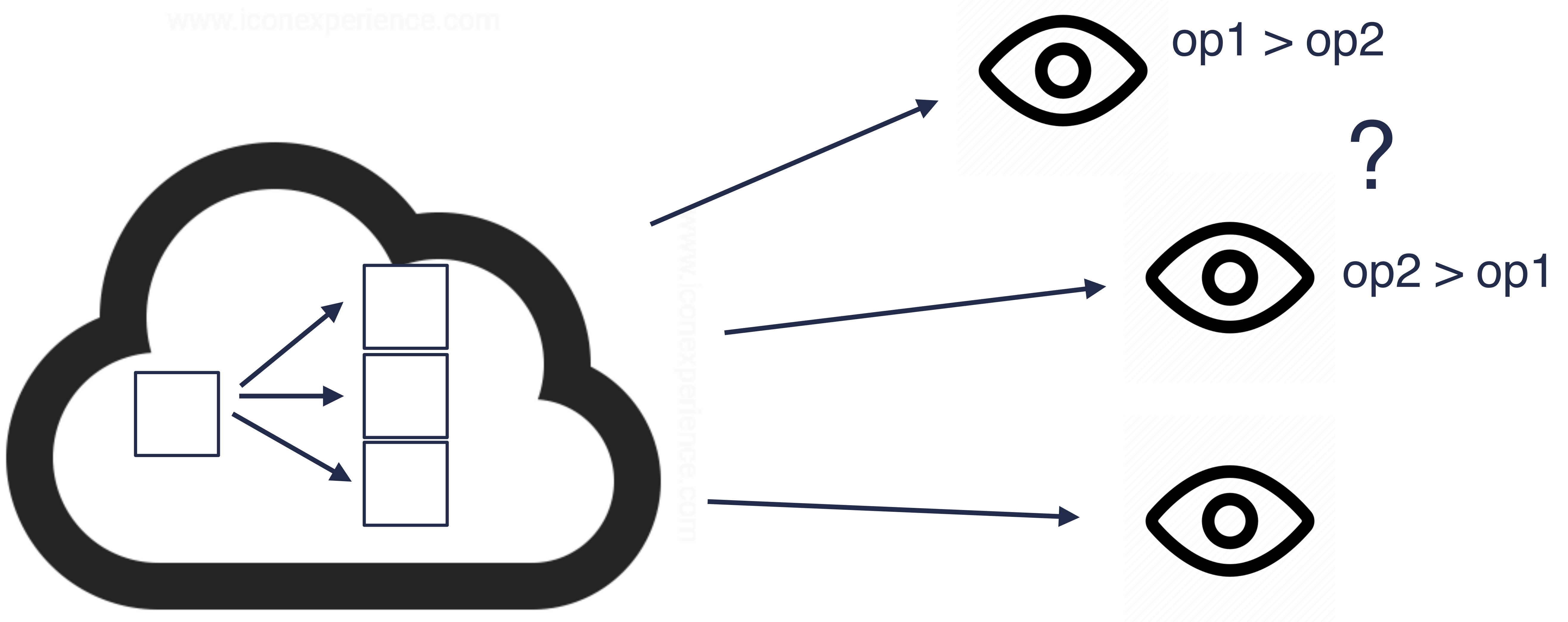
- Isolation: relevant only for transactional APIs.
  - Define how **concurrent** transactions interact with each other, i.e., whether individual effects of ongoing transactions can be witnessed by other transactions or not.



Isolation

CONTEXT

- Consistency: relevant for both Tx and non-Tx APIs (our focus).
  - Constrain the **order** in which individual operations (or individual transactions for a Tx API) are witnessed by different readers.



Consistency

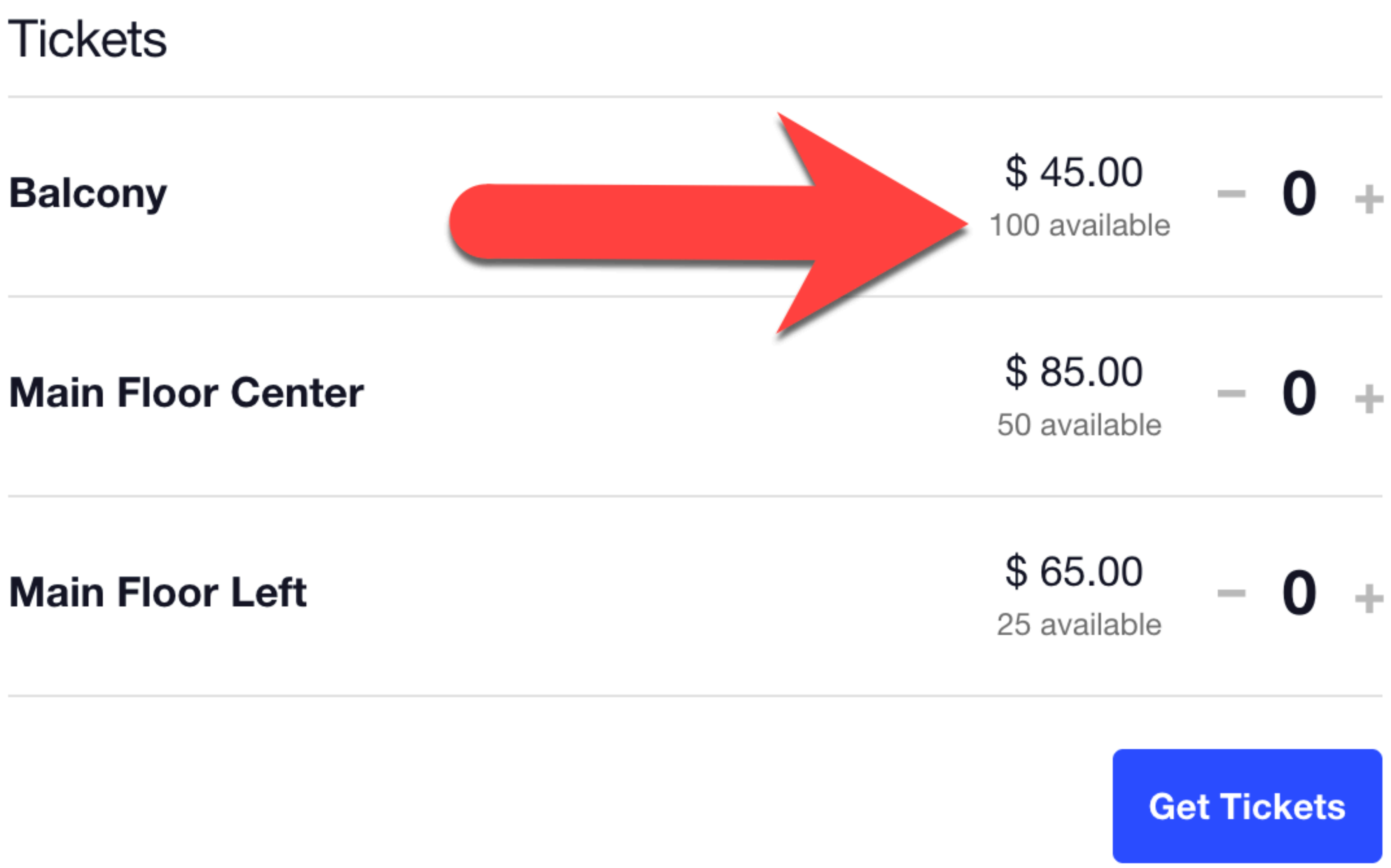
CONTEXT

– Why should you care about isolation and consistency?

CONTEXT

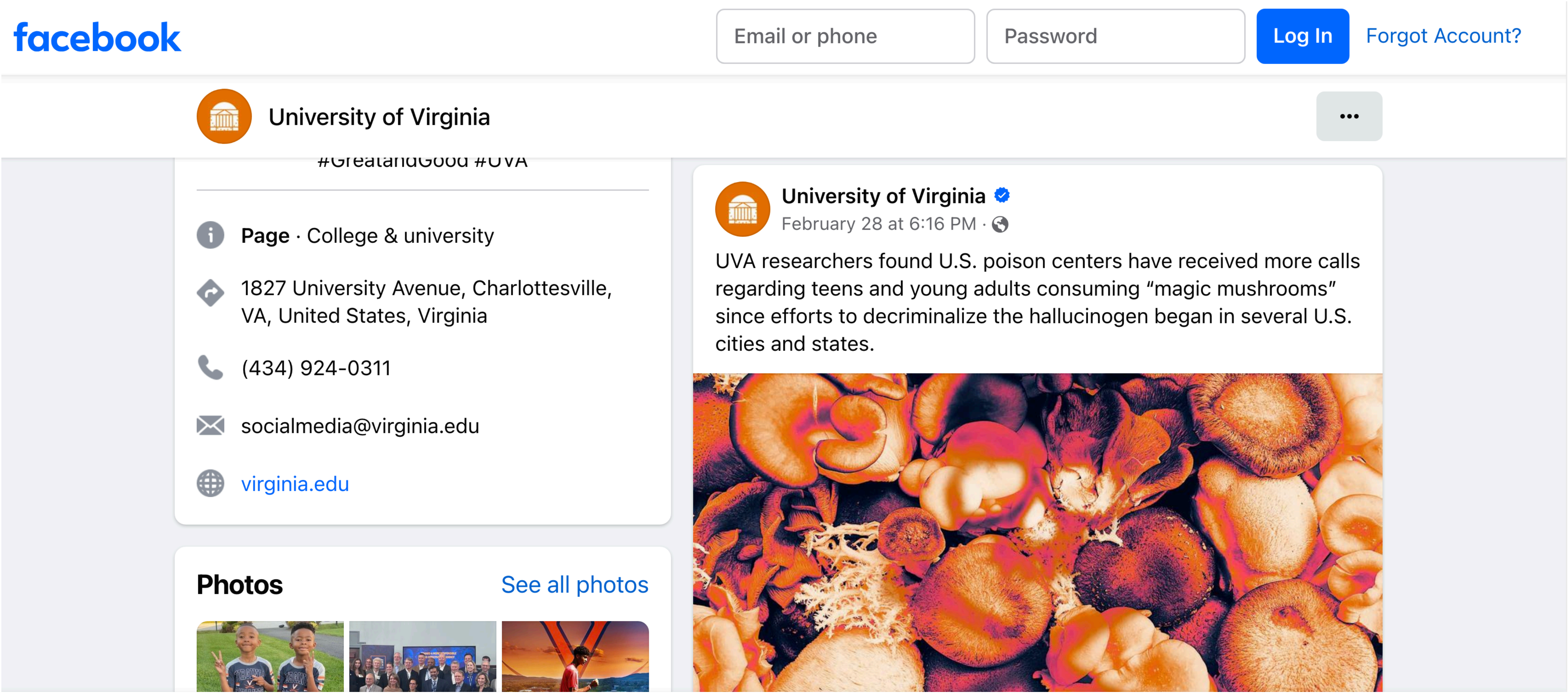
- Why should you care about isolation and consistency?
  - Together they provide correctness guarantees.
  - Without them, a lot of weird stuff will happen!

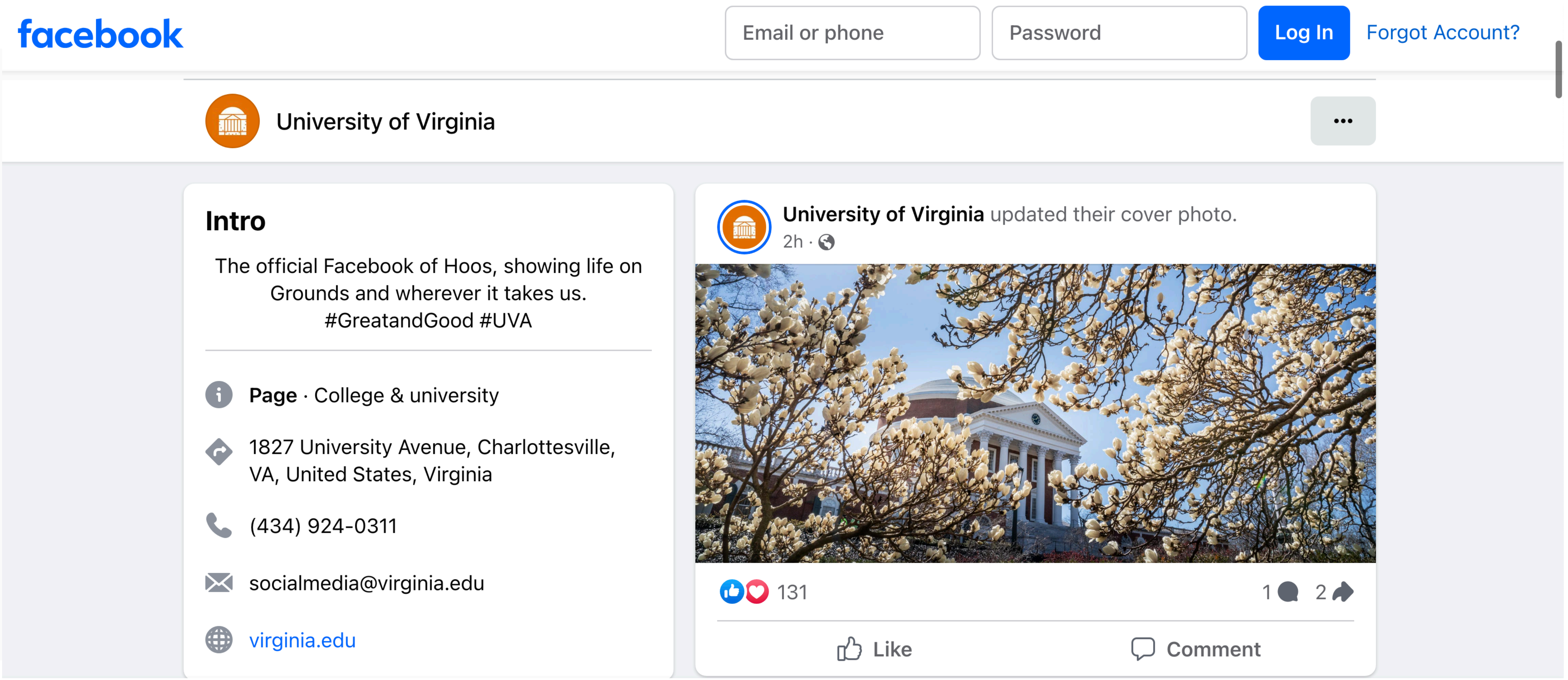
IMAGINE A WORLD WITH NO ISOLATION



Never-Ending Ticket Sale!

IMAGINE A WORLD WITH NO CONSISTENCY





Some users see latest updates, some users don't..

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

ISOLATION SEMANTICS

- Gold standard: **serializability**
  - transactions are completely isolated from each other.
  - for this, the DB engine must serialize conflicting transactions.
- Downside?
  - expensive!
  - solution: provide other isolation levels 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

- Anomaly will happen!

1. DIRTY READS

– A dirty read (aka uncommitted dependency) occurs when a transaction retrieves a row that has been updated by another transaction that is **not yet committed**.

Transaction 1

```
BEGIN;
SELECT age FROM users WHERE id = 1;
-- retrieves 20
```

Transaction 2

```
BEGIN;
UPDATE users SET age = 21 WHERE id = 1;
-- no commit here
```

Transaction 1

```
SELECT age FROM users WHERE id = 1;
-- READ UNCOMMITTED retrieves 21 (dirty read)
-- READ COMMITTED retrieves 20 (dirty read has been avoided)
-- REPEATABLE READ retrieves 20 (dirty read has been avoided)
-- SERIALIZABLE retrieves 20 (dirty read has been avoided)
COMMIT;
```

Transaction 2

```
ROLLBACK;
```

2. NON-REPEATABLE READS (FUZZY READS)

– A non-repeatable read occurs when a transaction retrieves a row twice and that row is updated by another transaction that is **committed in between**.

Transaction 1

```
BEGIN;
SELECT age FROM users WHERE id = 1;
-- retrieves 20
```

Transaction 2

```
BEGIN;
UPDATE users SET age = 21 WHERE id =
1;
COMMIT;
```

Transaction 1

```
SELECT age FROM users WHERE id = 1;
-- READ UNCOMMITTED retrieves 21 (non-repeatable read)
-- READ COMMITTED retrieves 21 (non-repeatable read)
-- REPEATABLE READ retrieves 20 (non-repeatable read has been
avoided)
-- SERIALIZABLE retrieves 20 (non-repeatable read has been avoided)
COMMIT;
```

Q: Difference with Dirty Read?

A: Uncommitted/Committed

3. PHANTOM READS

– A phantom read occurs when a transaction retrieves **a set of rows** twice and new rows are inserted into or removed from that set by another transaction that is committed in between.

Transaction 1

```sql
BEGIN;
SELECT name FROM users WHERE age > 17;
-- retrieves Alice and Bob
```

Transaction 2

```sql
BEGIN;
INSERT INTO users VALUES (3, 'Carol',
26);
COMMIT;
```

Transaction 1

```sql
SELECT name FROM users WHERE age > 17;
-- READ UNCOMMITTED retrieves Alice, Bob and Carol (phantom read)
-- READ COMMITTED retrieves Alice, Bob and Carol (phantom read)
-- REPEATABLE READ retrieves Alice, Bob and Carol (phantom read)
-- SERIALIZABLE retrieves Alice and Bob (phantom read has been
avoided)
COMMIT;
```

Q: Difference with Fuzzy Read?

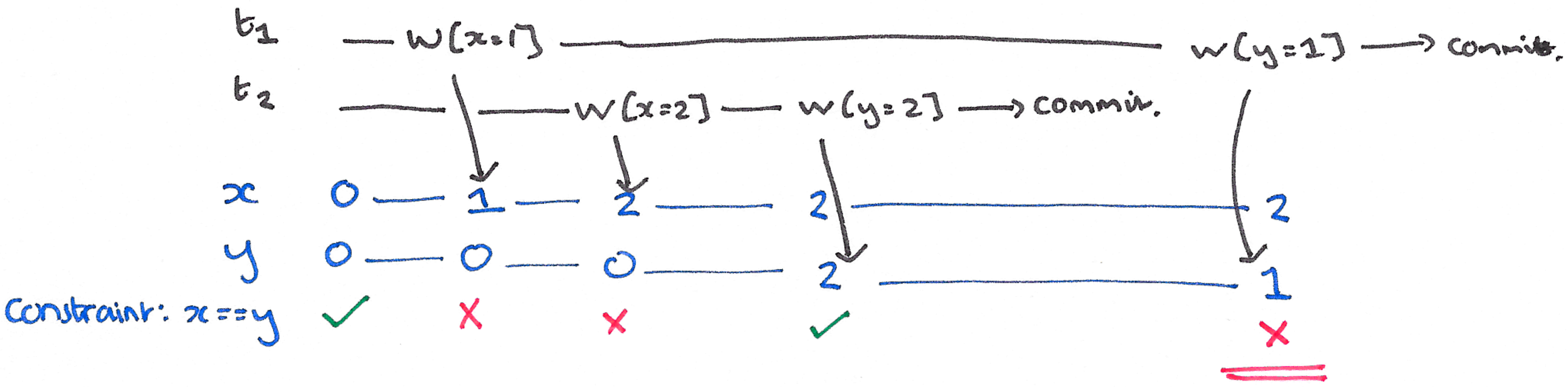
A: Row/Set of rows

ISOLATION SEMANTICS

Read phenomenon / Isolation level	Dirty read	Non-repeatable read	Phantom read
Serializable	no	no	no
Repeatable read	no	no	yes
Read committed	no	yes	yes
Read uncommitted	yes	yes	yes

"READ UNCOMMITTED" == NO ISOLATION?

- No, nearly all isolation levels prevent **dirty writes**
  - Suppose T1 modifies x and T2 further modifies x before T1 commits or aborts. If either T1 or T2 aborts, it is unclear what the real value of x should be



Source: Morning paper

"READ UNCOMMITTED" == NO ISOLATION?

- No, nearly all isolation levels prevent **dirty writes**
  - Suppose T1 modifies x and T2 further modifies x before T1 commits or aborts. If either T1 or T2 aborts, it is unclear what the real value of x should be

Table 3. ANSI SQL Isolation Levels Defined in terms of the four phenomena

Isolation Level	P0 Dirty Write	P1 Dirty Read	P2 Fuzzy Read	P3 Phantom
READ UNCOMMITTED	Not Possible	Possible	Possible	Possible
READ COMMITTED	Not Possible	Not Possible	Possible	Possible
REPEATABLE READ	Not Possible	Not Possible	Not Possible	Possible
SERIALIZABLE	Not Possible	Not Possible	Not Possible	Not Possible

"A Critique of ANSI SQL Isolation Levels," Proc. ACM SIGMOD 95

HOW TO IMPLEMENT

- 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).

HOW TO IMPLEMENT

- 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).
  - why still use it: performance + debugging long queries

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

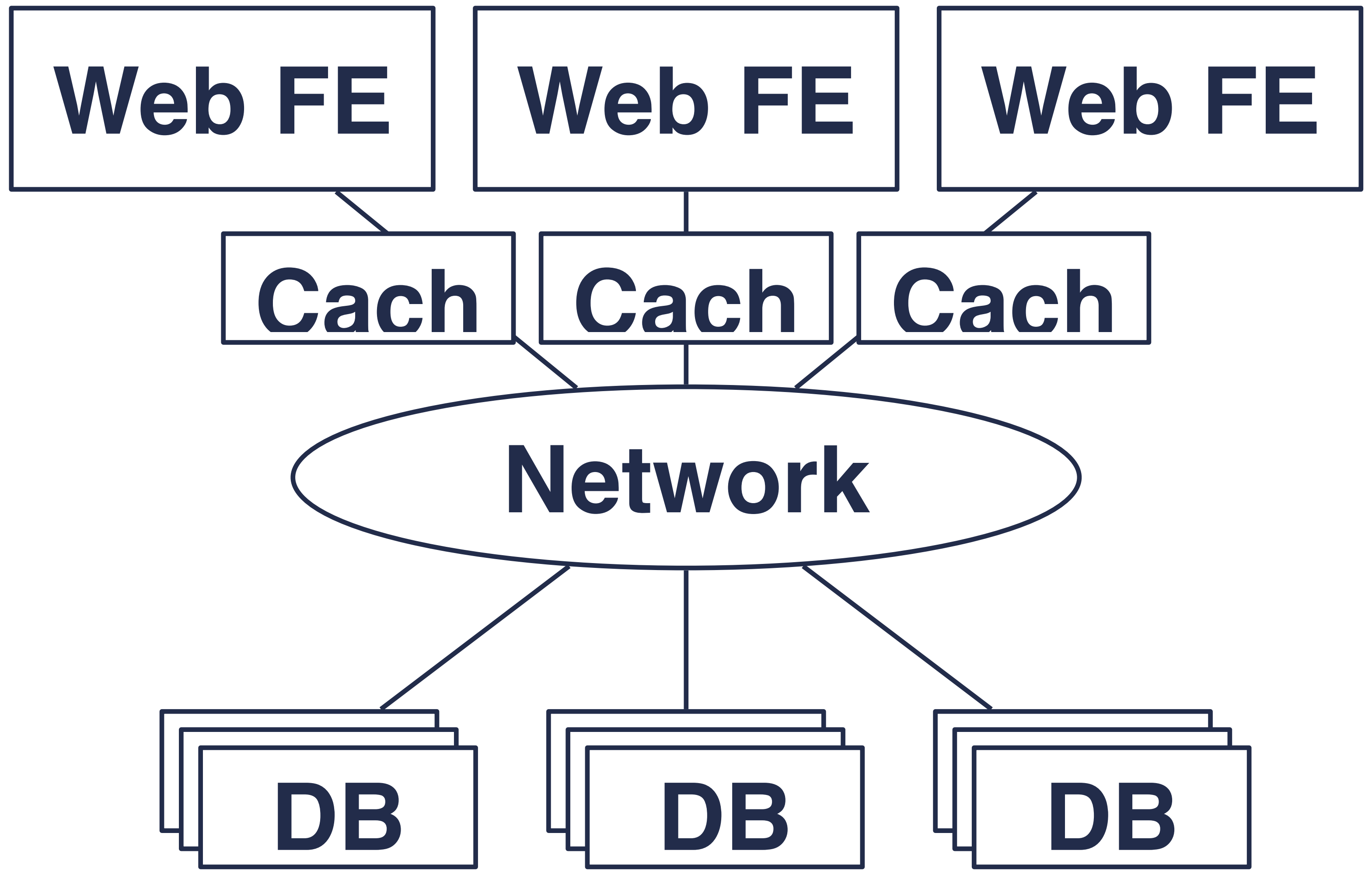
- Gold standard is **linearizability**: operations are seen in the real time order in which they are "committed" (finished). For this, the storage system must coordinate among replicas/shards, wait out clock uncertainty, etc. -- all of which can be very expensive.
- Other consistency models exist that offer weaker semantics (and hence more corner cases to consider when programming against them) but better performance, scalability, and sometimes availability.

CONSISTENCY SEMANTICS

– What can go wrong?

1. THE STALE READ

- A stale read is when a read operation does not return the most recent value.
- The user has $2000 in his account. Now he transfers $1000 to his children and he still sees $2000 in his account.



2. THE IMMORTAL WRITE

- The user wants to change his username from ‘Hans’ to ‘Peter’, but changes to ‘Peteeer’ instead.
- The user corrects the username to ‘Peteeer’. The next time the user logs in to online banking, however, he see the username ‘Peteeer’ again.

3. THE CAUSAL REVERSE

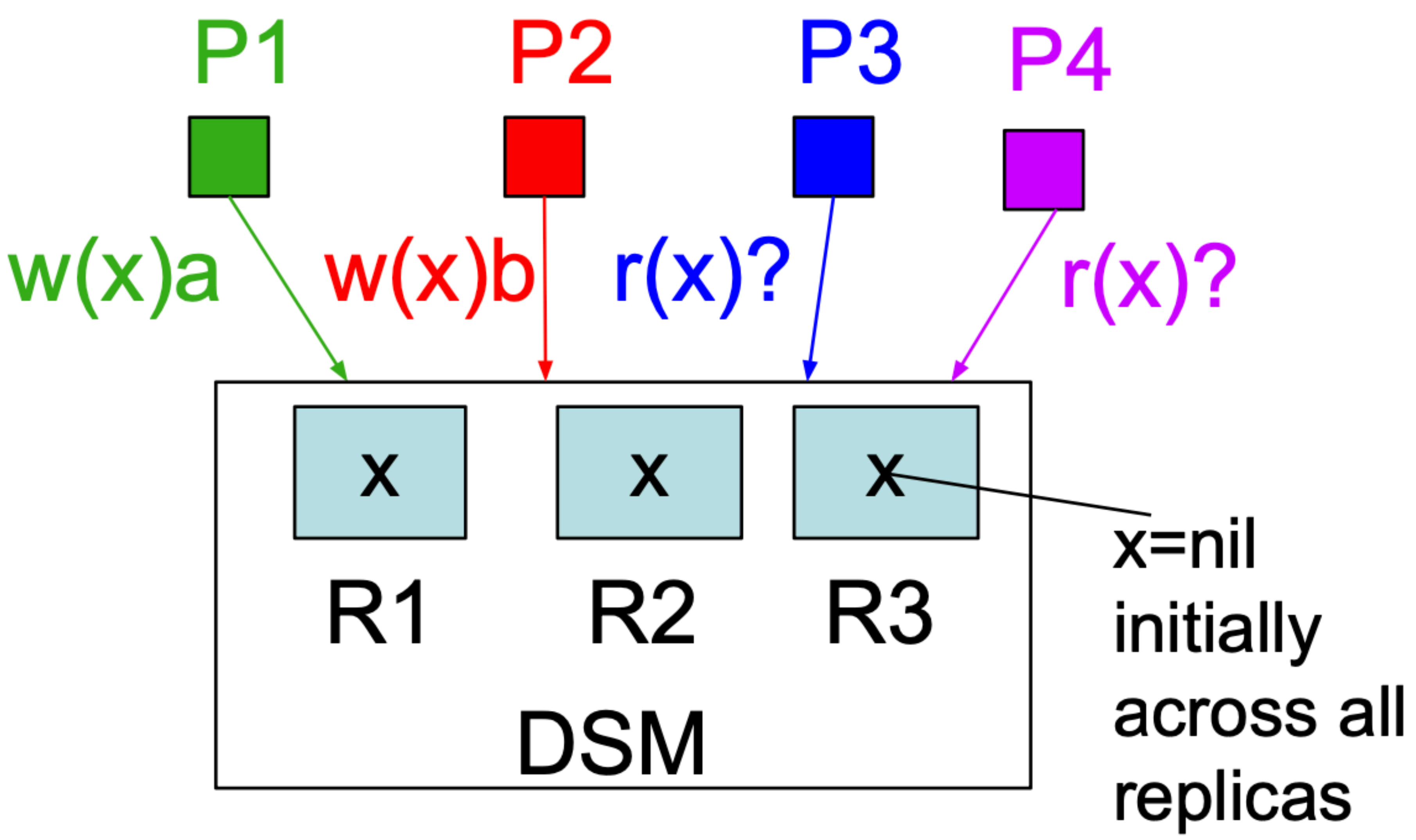
- Now the user wants to transfer $30,000 from his savings account to his bank account. In his savings account he has exactly the $30,000 and in his bank account he has $1,000, but after the transfer he will see a total of $61,000 in his accounts.

BEST KNOWN CONSISTENCY MODELS

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

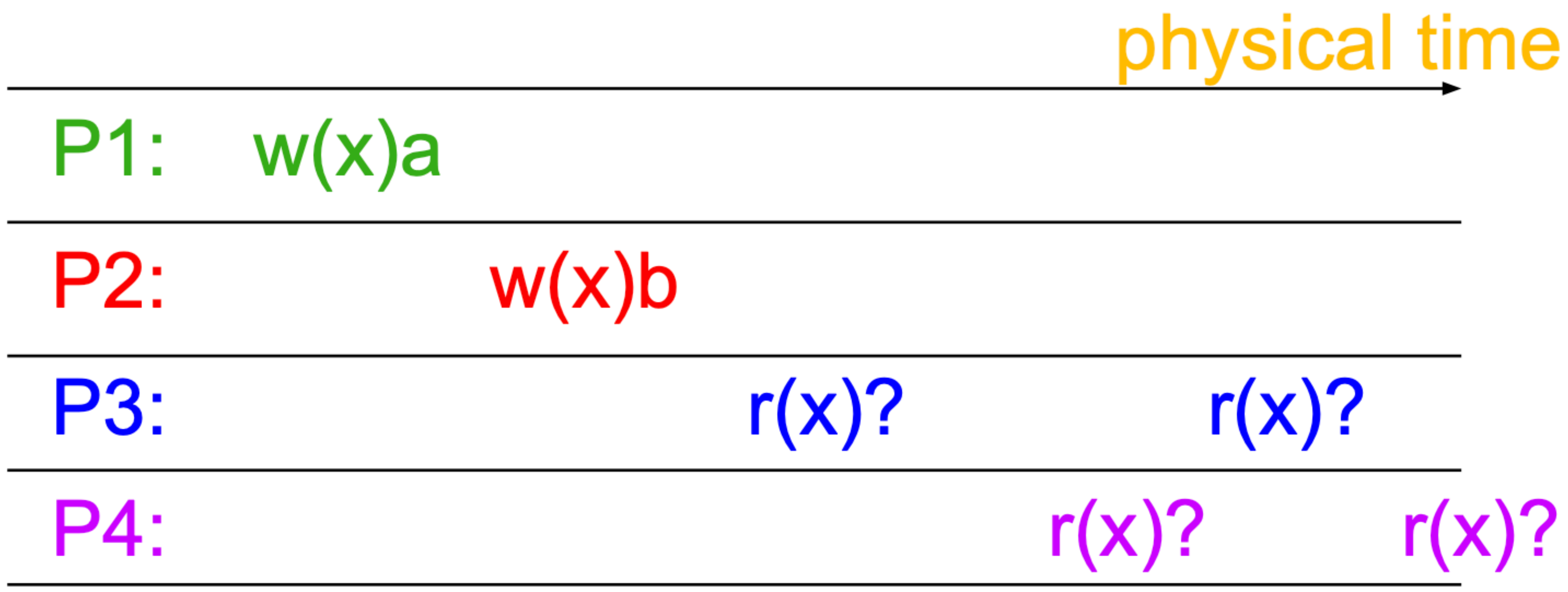
- Variations boil down to: (1) the allowable staleness of reads and (2) the ordering of writes across all replicas.

EXAMPLES WITH REPLICATED DISTRIBUTED SHARED MEMORY (DSM)



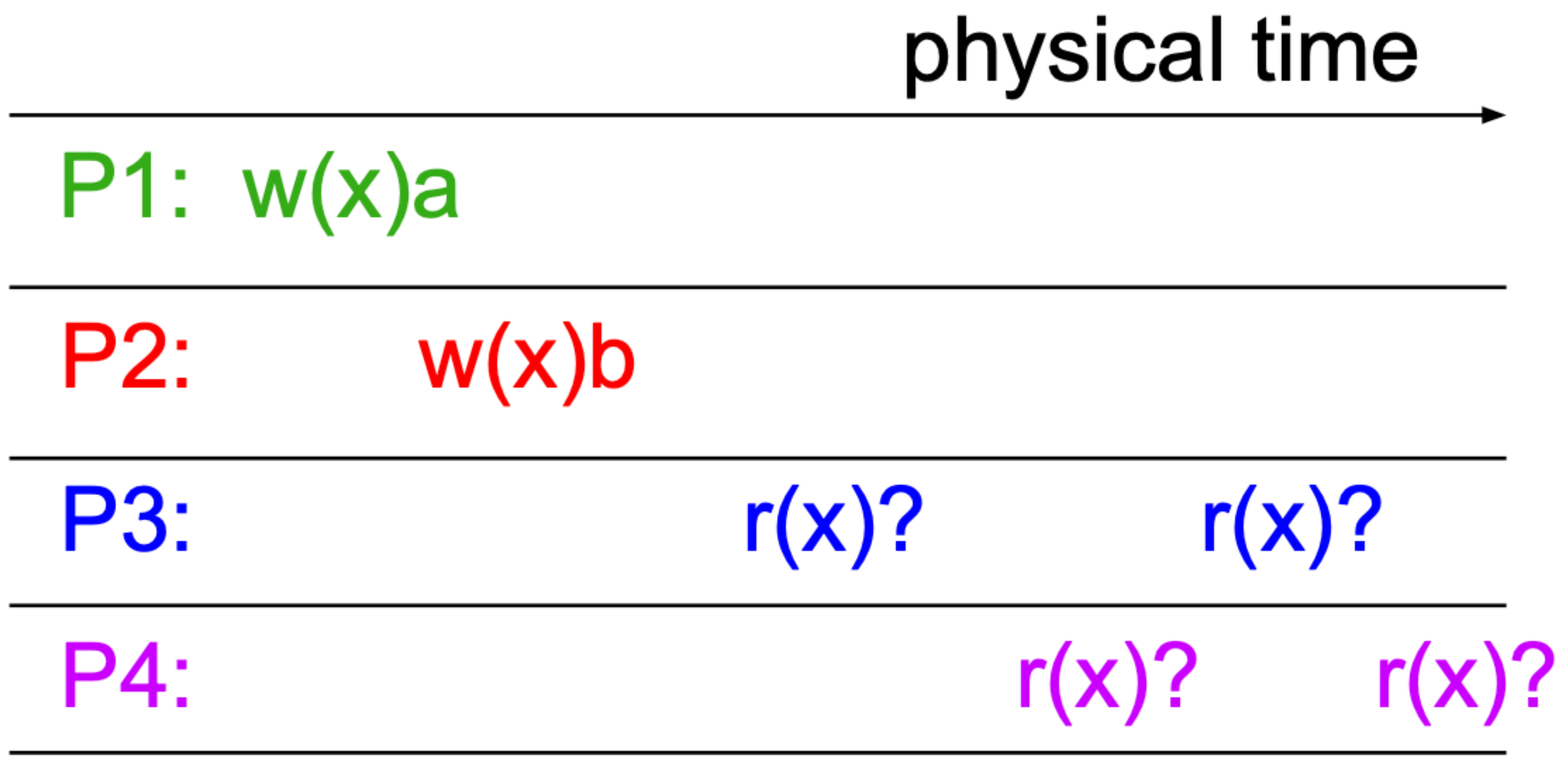
- Distributed shared memory (DSM): a form of memory architecture where physically separated memories can be addressed as a single shared address space.
- In the slides, we will use individual examples to show what's admissible vs. not for a given semantic.

STRUCTURE OF AN EXAMPLE



STRICT CONSISTENCY

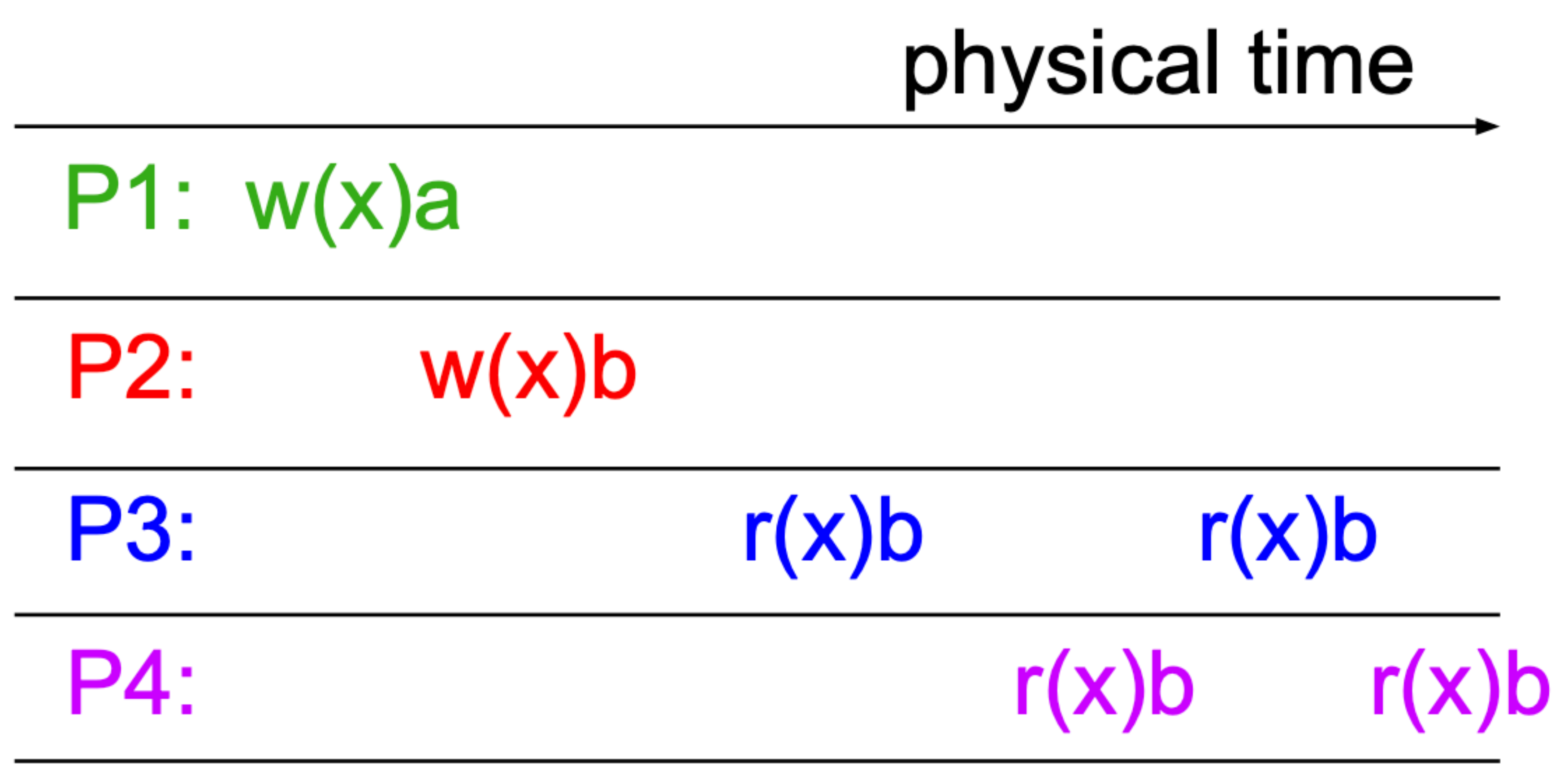
- Defn: Any execution is the same as if all read/write ops were executed in order of physical time at which they were issued.
- Therefore: (1) Reads are never stale; (2) all replicas enforce physical-time ordering for all writes.

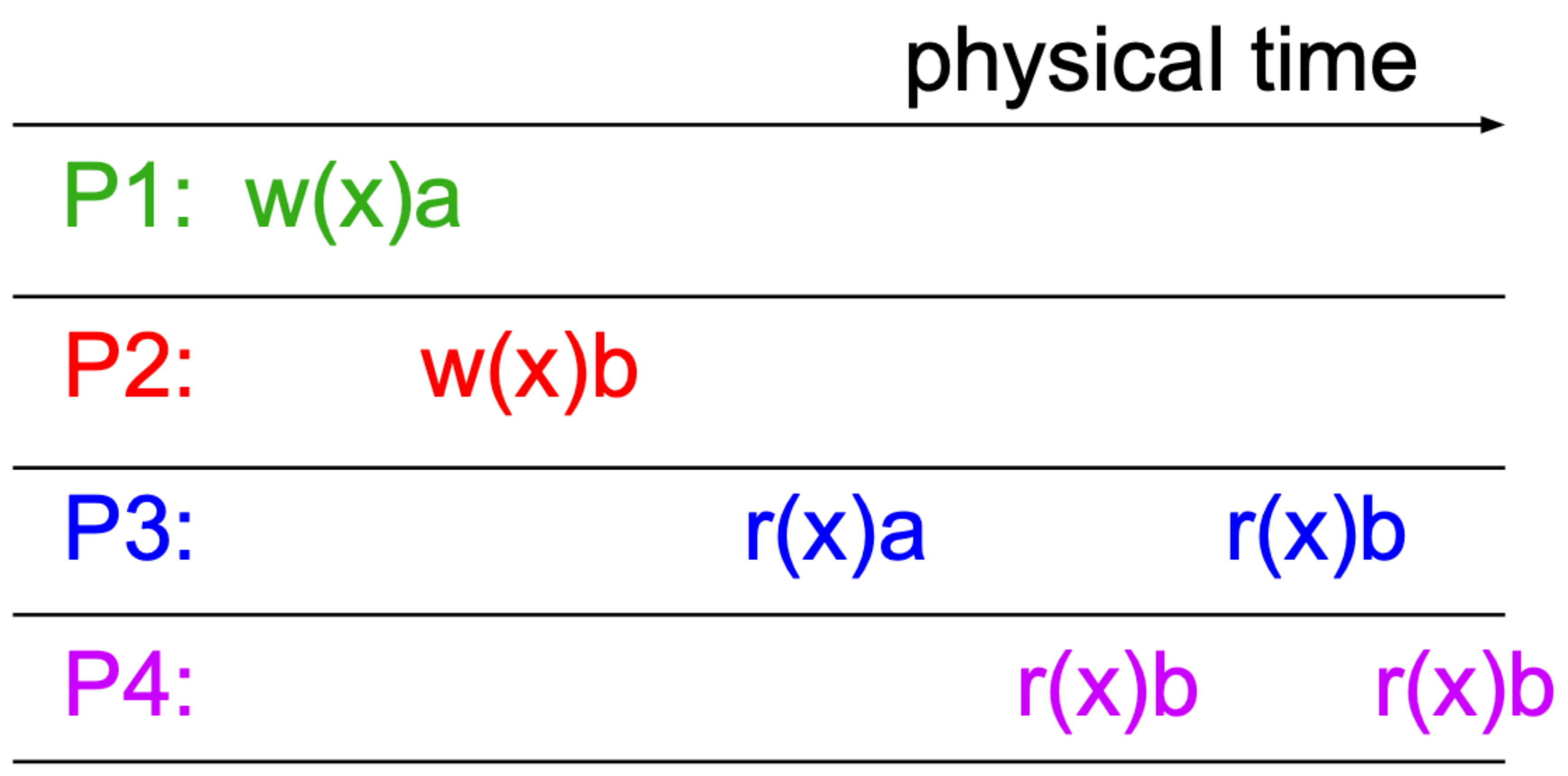


if DSM is strictly consistent, what can these reads return?

STRICT CONSISTENCY

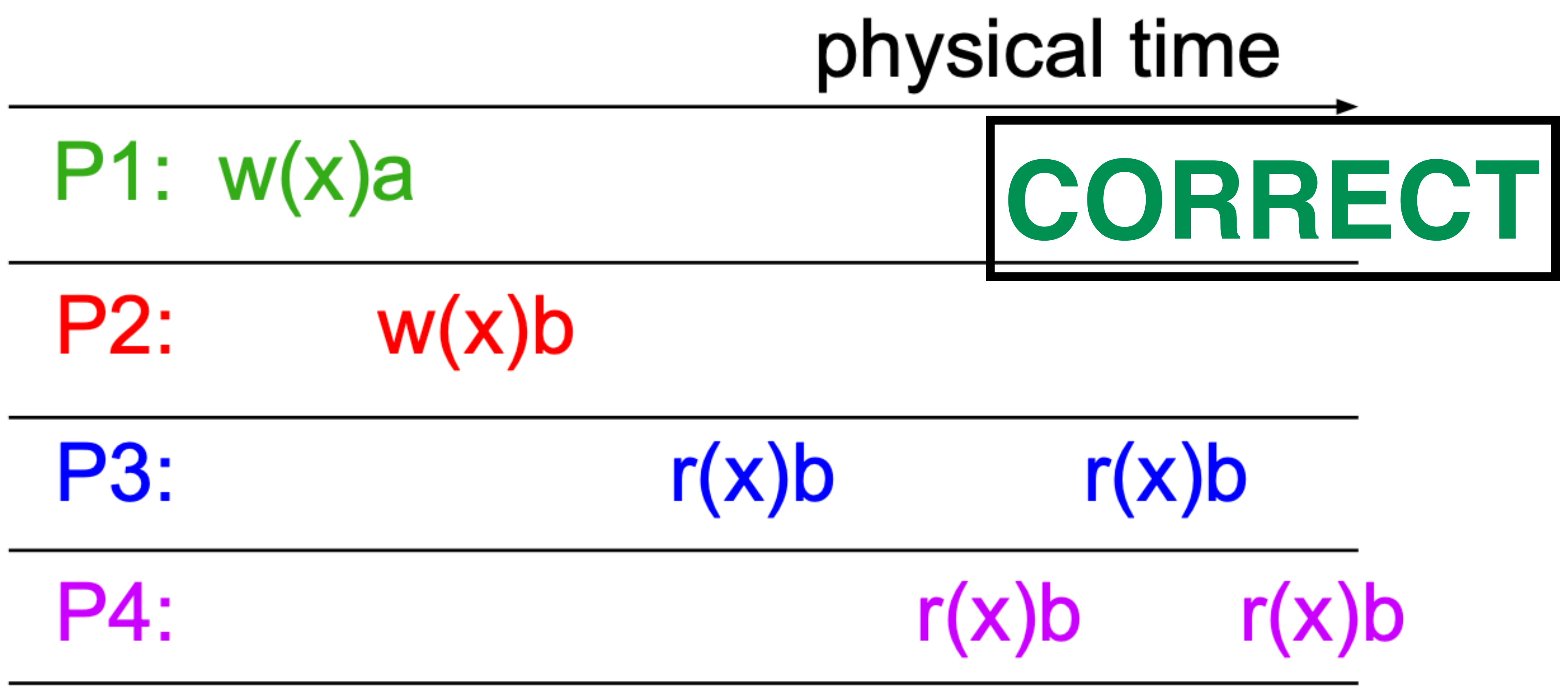
- Defn: Any execution is the same as if all read/write ops were executed in order of physical time at which they were issued.
- Therefore: (1) Reads are never stale; (2) all replicas enforce physical-time ordering for all writes.

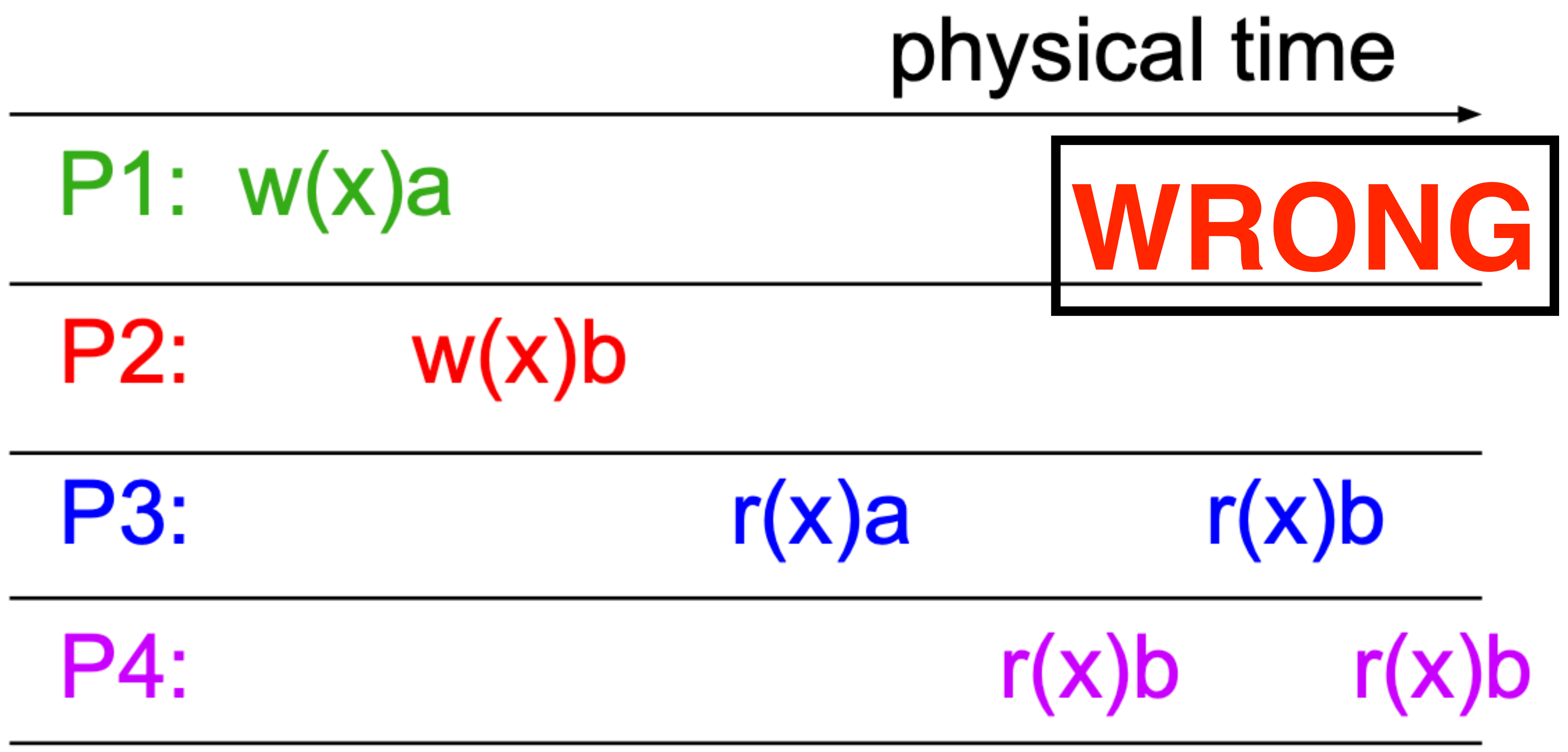




STRICT CONSISTENCY

- 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.





STRICT CONSISTENCY

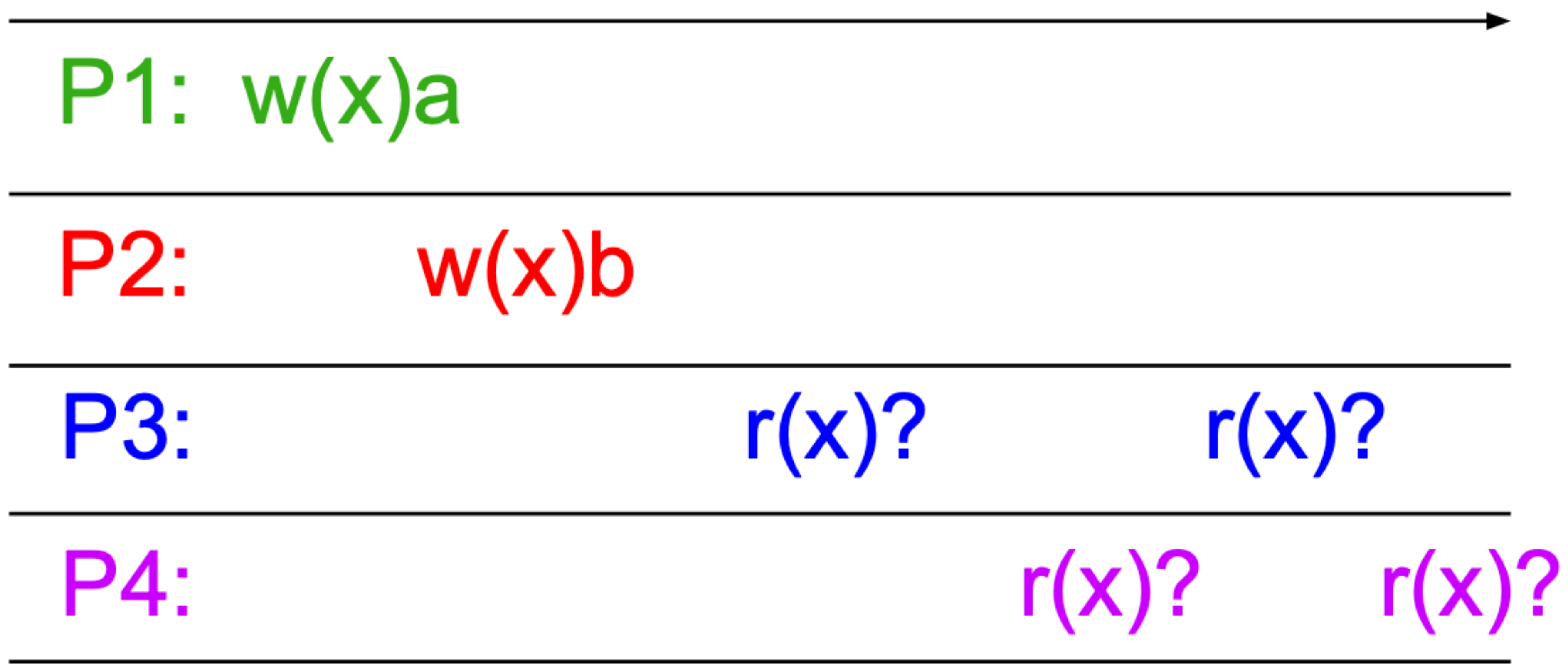
- However, strict consistency isn't implementable..
  - Why?

STRICT CONSISTENCY

- However, strict consistency isn't implementable..
  - Why?
  - instantaneous message exchange is impossible
  - a thought experiment and formalism

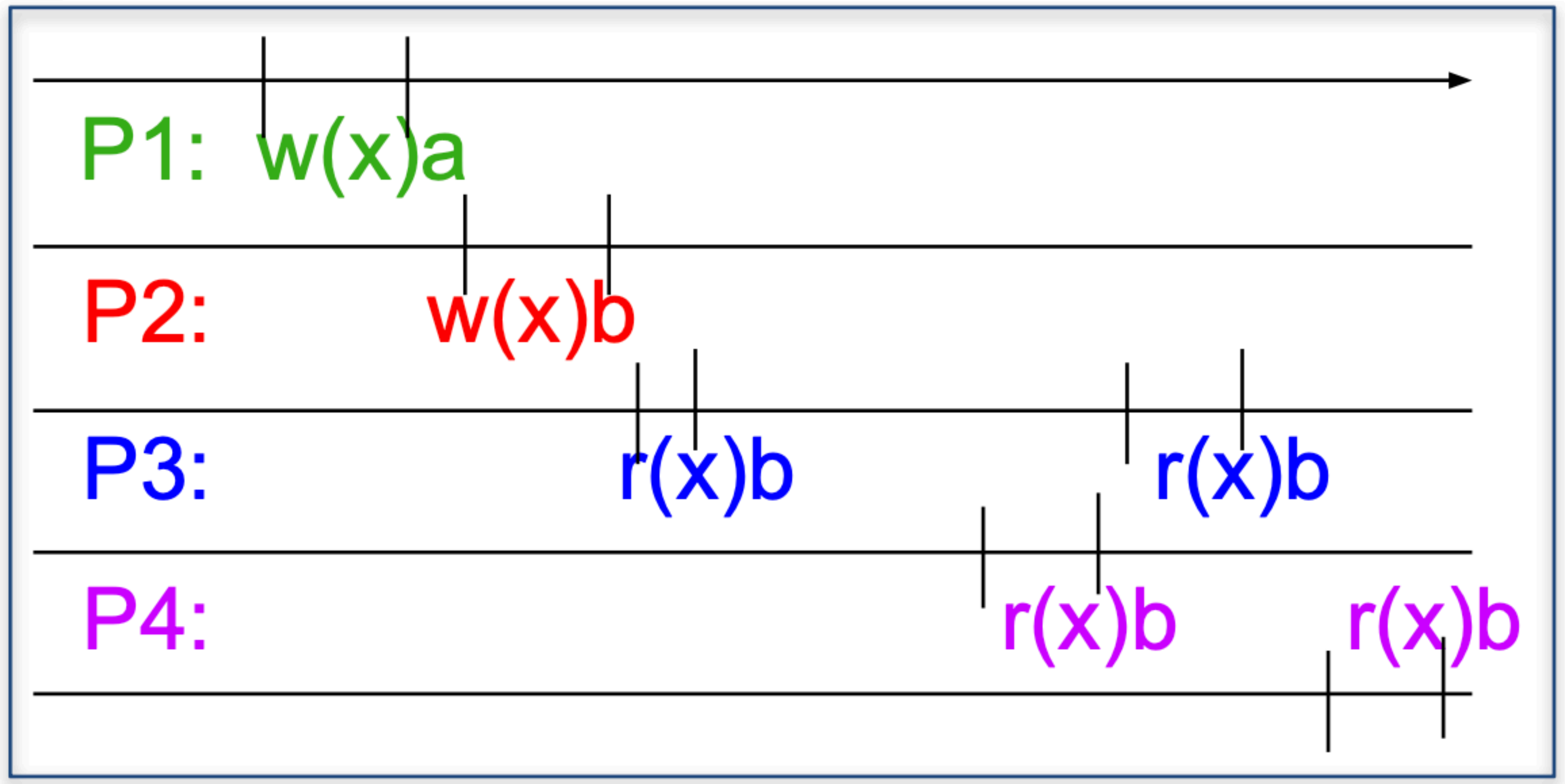
LINEARIZABILITY

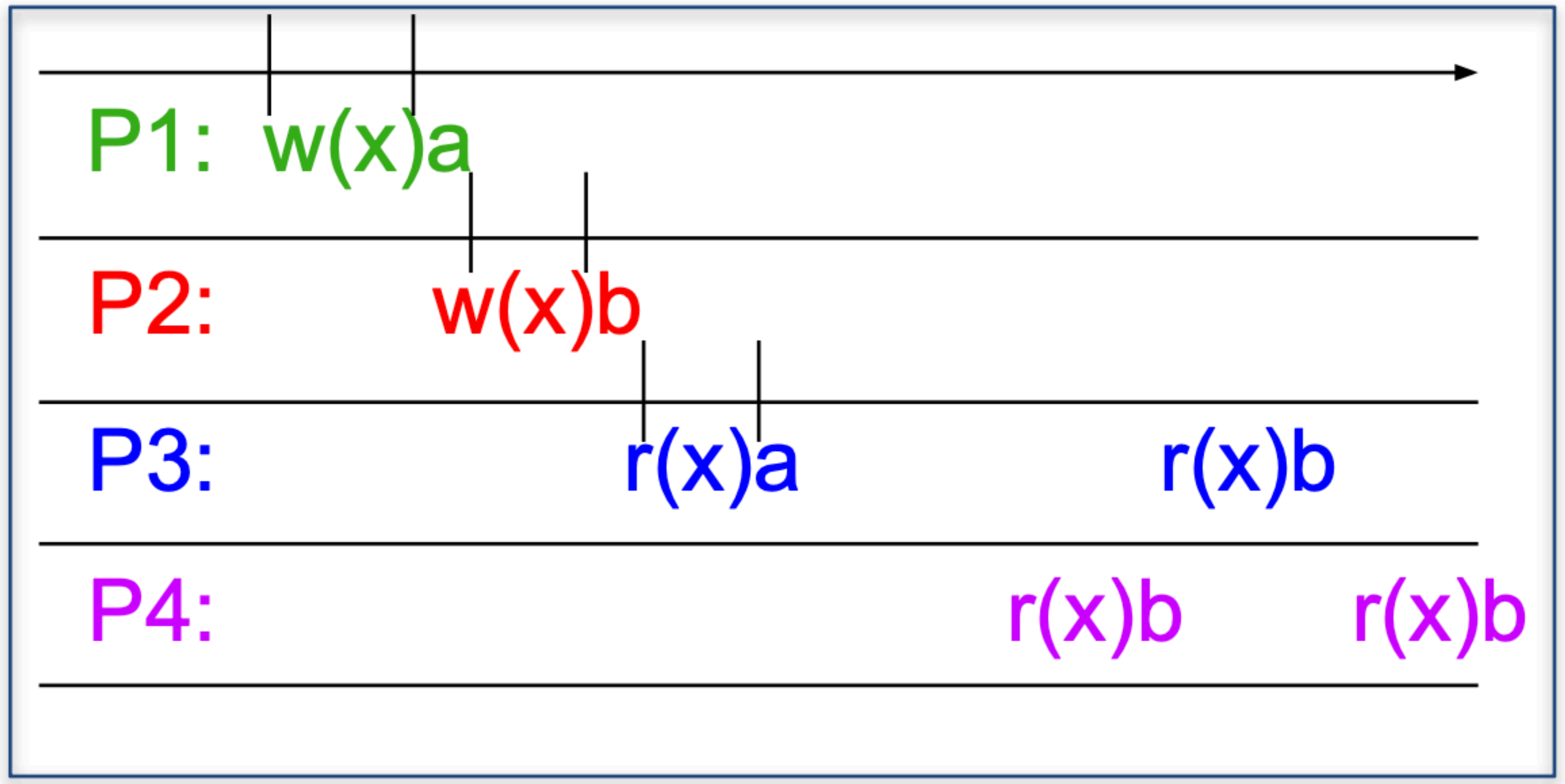
- Defn: Any execution is the same as if all read/write ops were executed in some global order s.t. any read returns the value of the most recent completed write at that location.
- Therefore: (1) Once a write completes, all later reads return the value of that write or of a later write. (2) Once a read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

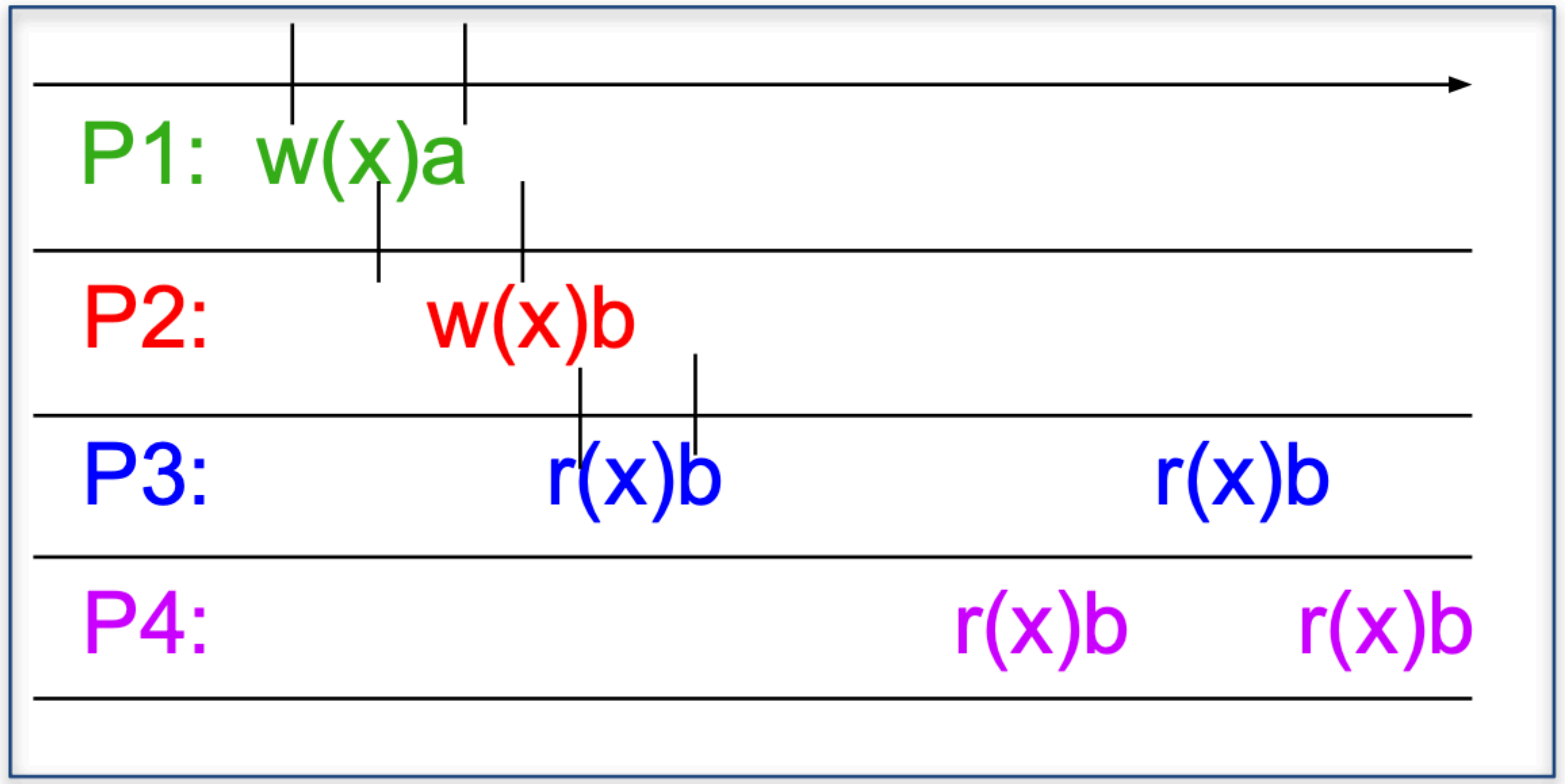


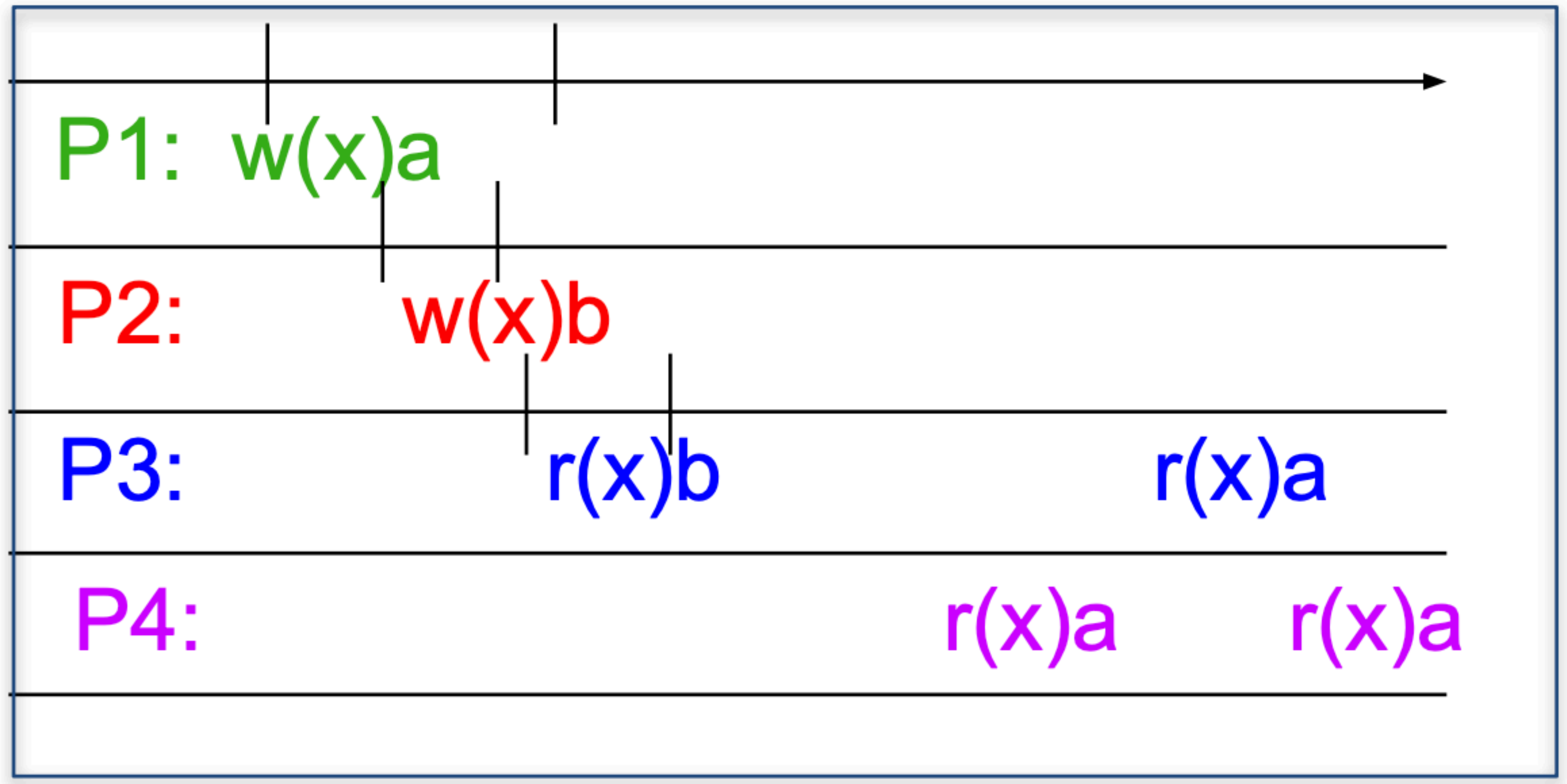
if DSM is strictly linearizable, what can these reads return?

LINEARIZABILITY

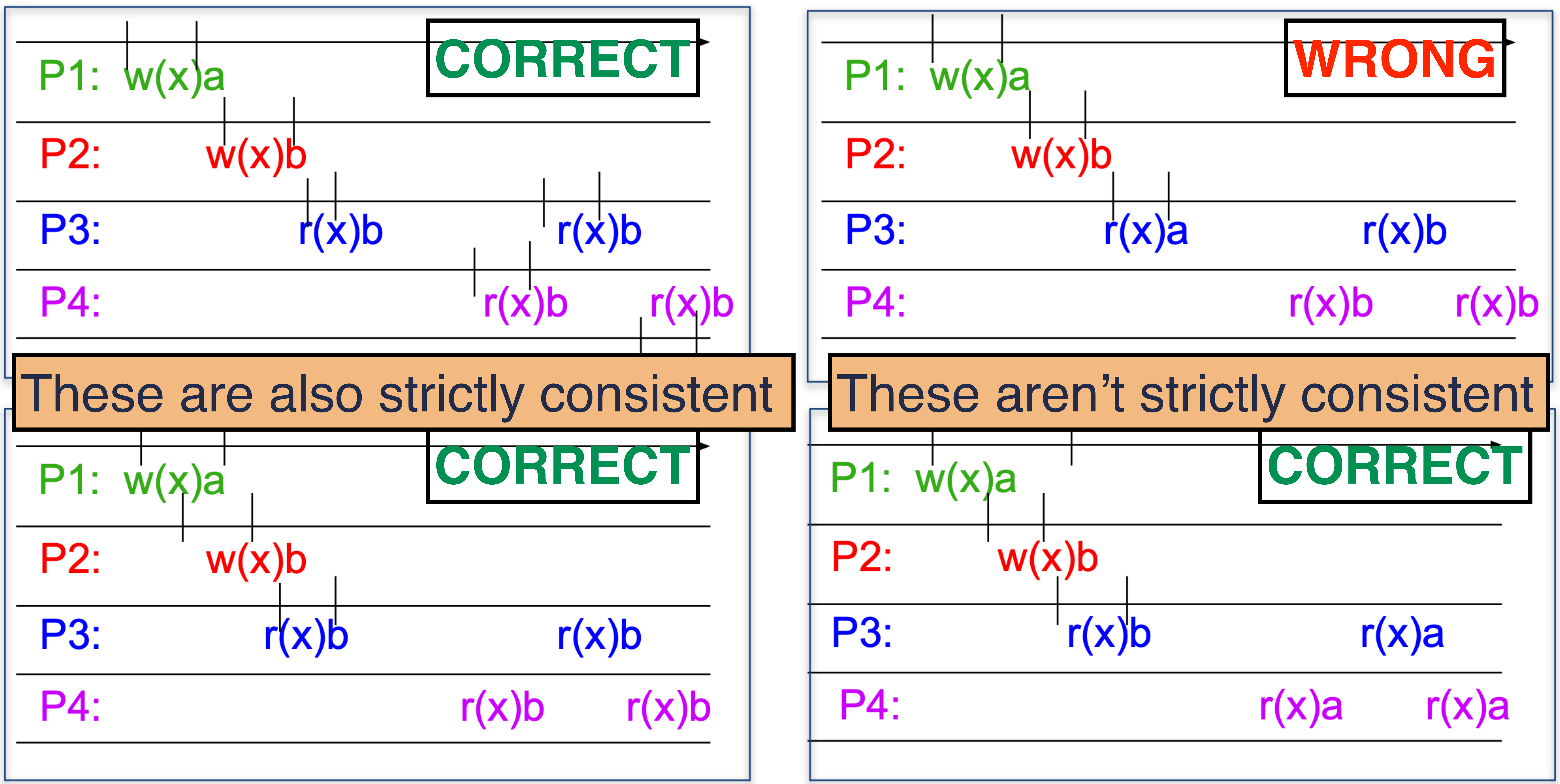








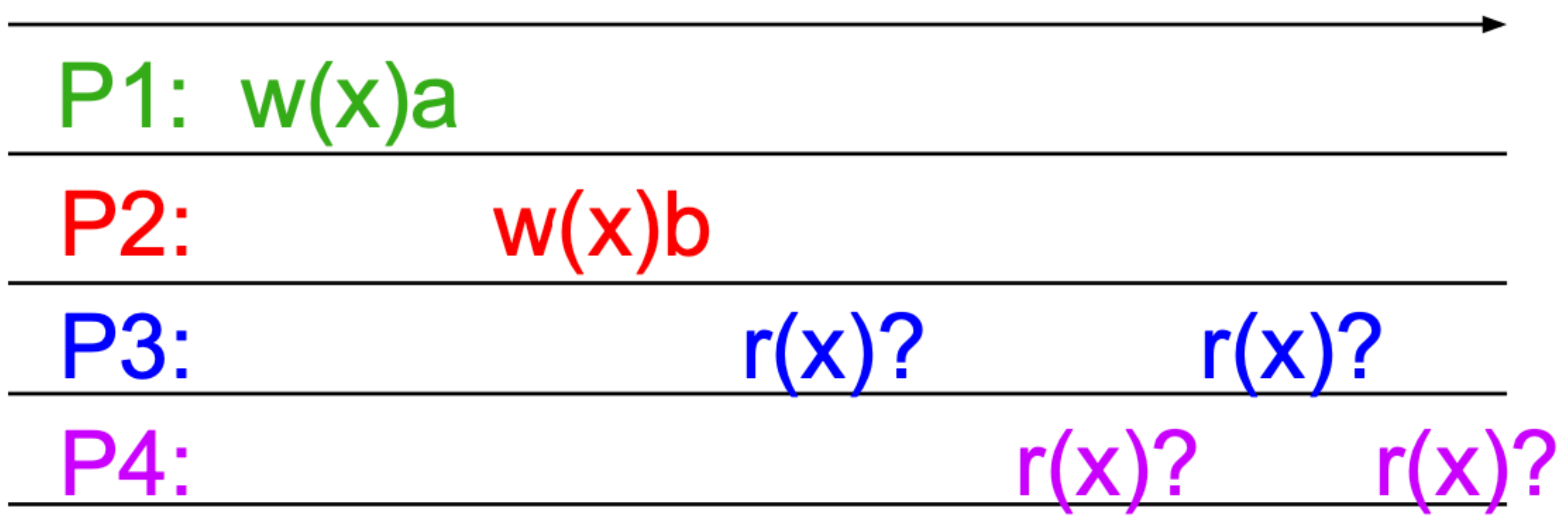
LINEARIZABILITY



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.)

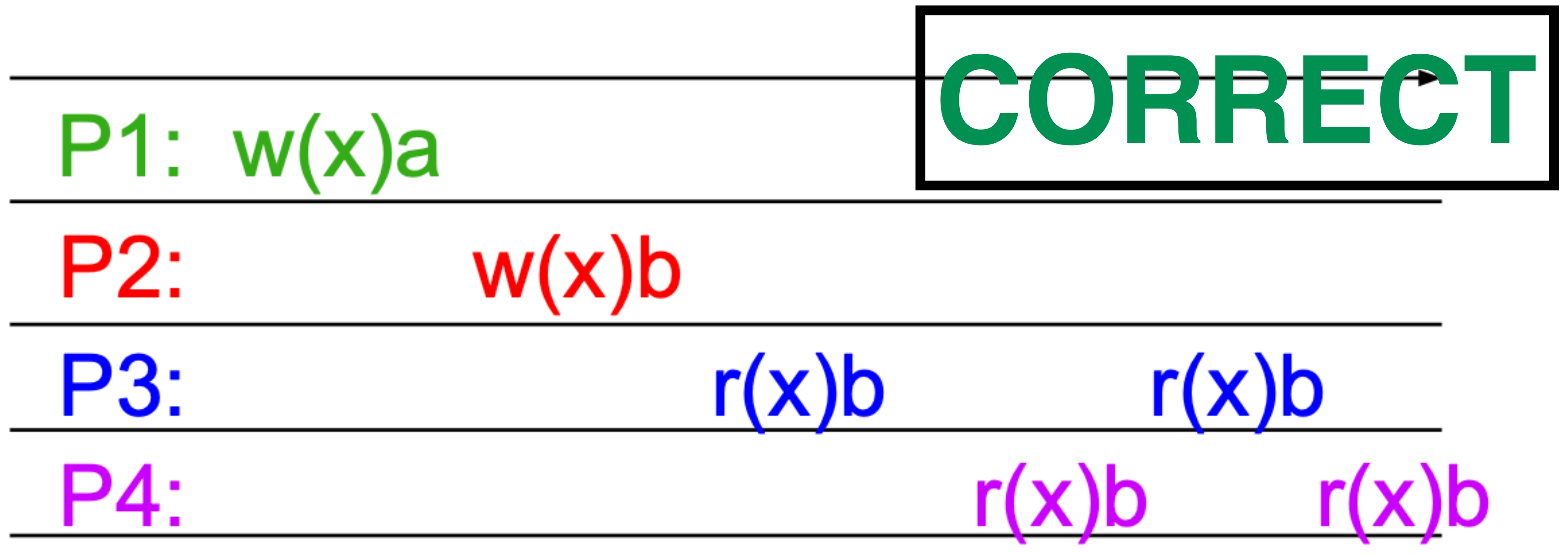
– Therefore: (1) Reads may be stale in real time, but not in logical time; (2) Writes are totally ordered according to logical time across all replicas.



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

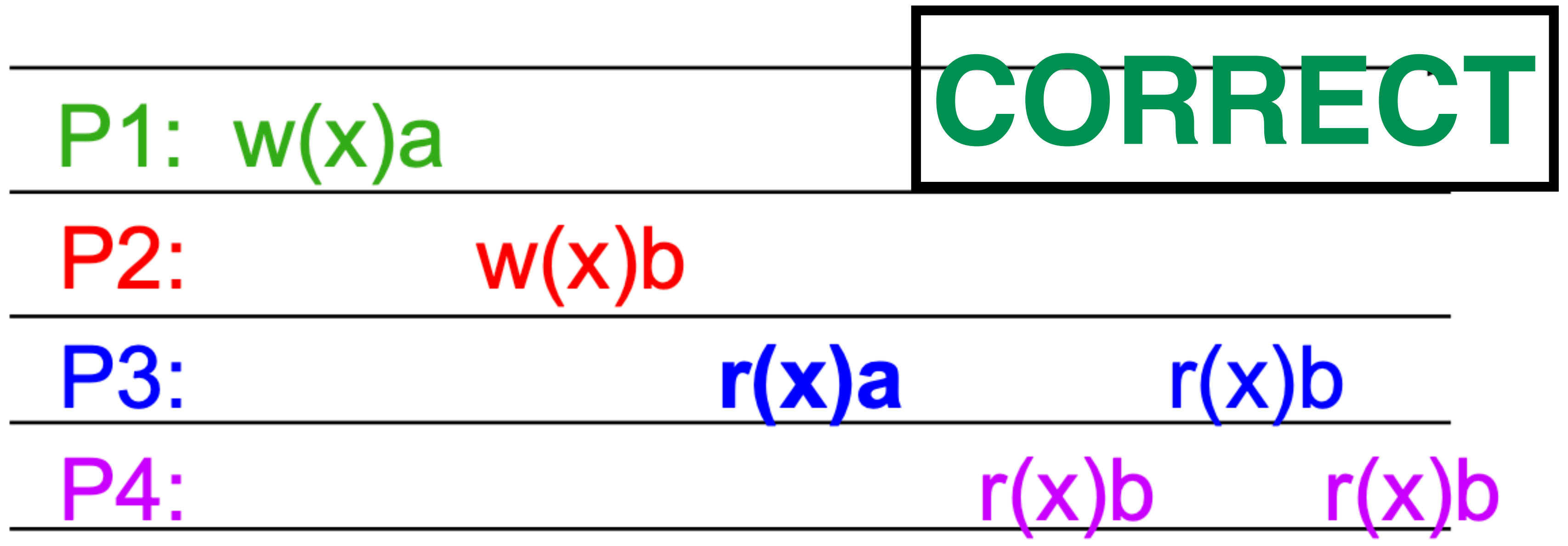
SEQUENTIAL CONSISTENCY

– 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. (This global order that adheres to program order is called global sequential order.)



What's a global sequential order that can explain these results?
physical-time ordering

This was also linearizable

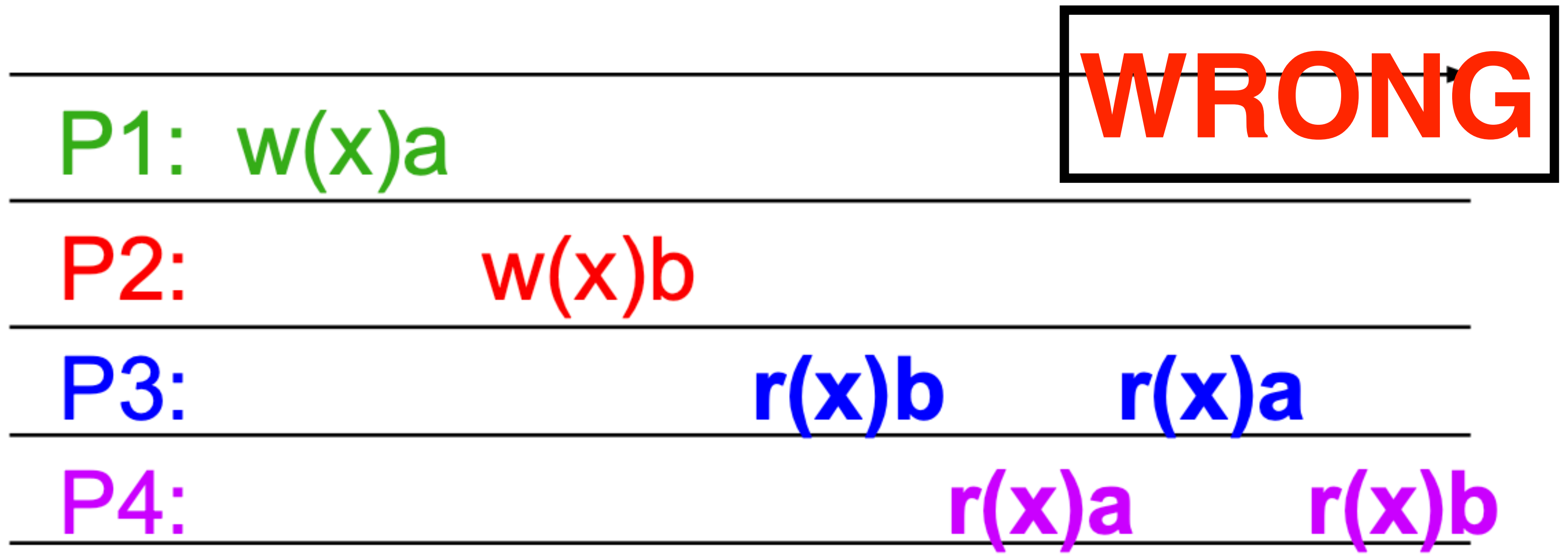


What's a global sequential order that can explain these results?
w(x)a, r(x)a, w(x)b, r(x)b, ...

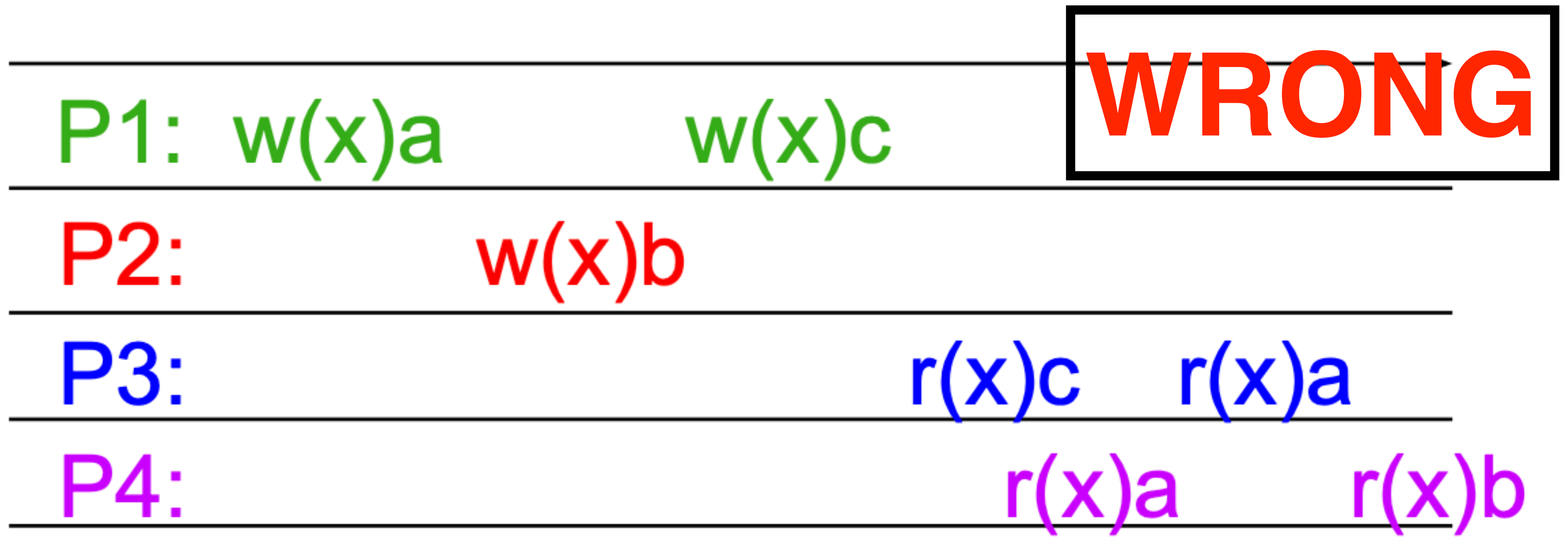
This wasn't linearizable

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. (This global order that adheres to program order is called global sequential order.)



No global order can explain these results...
=> not seq. consistent



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.
- Sound Strange? Think about Twitter Timeline.

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.

CORRECT

P1: w(x)a
P2: w(x)b
P3: **r(x)b** **r(x)a**
P4: **r(x)a** **r(x)b**

Only per-process ordering restrictions: w(x)b < r(x)b; r(x)b < r(x)a; ... w(x)a II w(x)b, hence they can be seen

This wasn’t sequentially consistent.

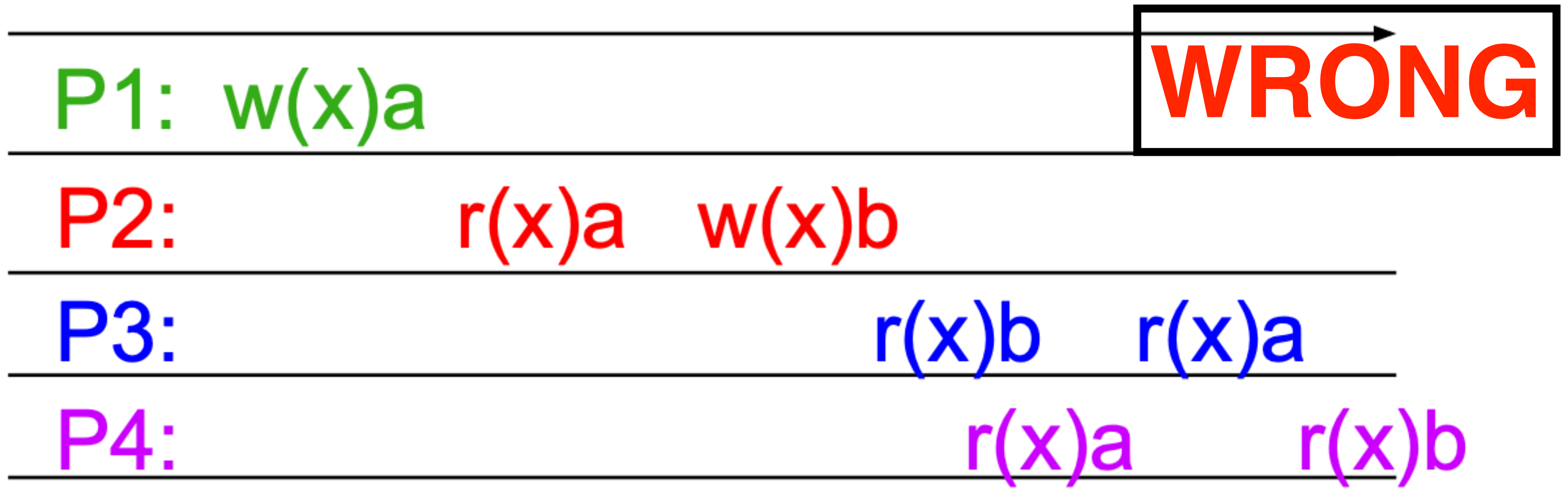
WRONG

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

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.



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 (OPTIONAL)

— Allow stale reads, but ensure that reads will eventually reflect previously written values, even after a long time.

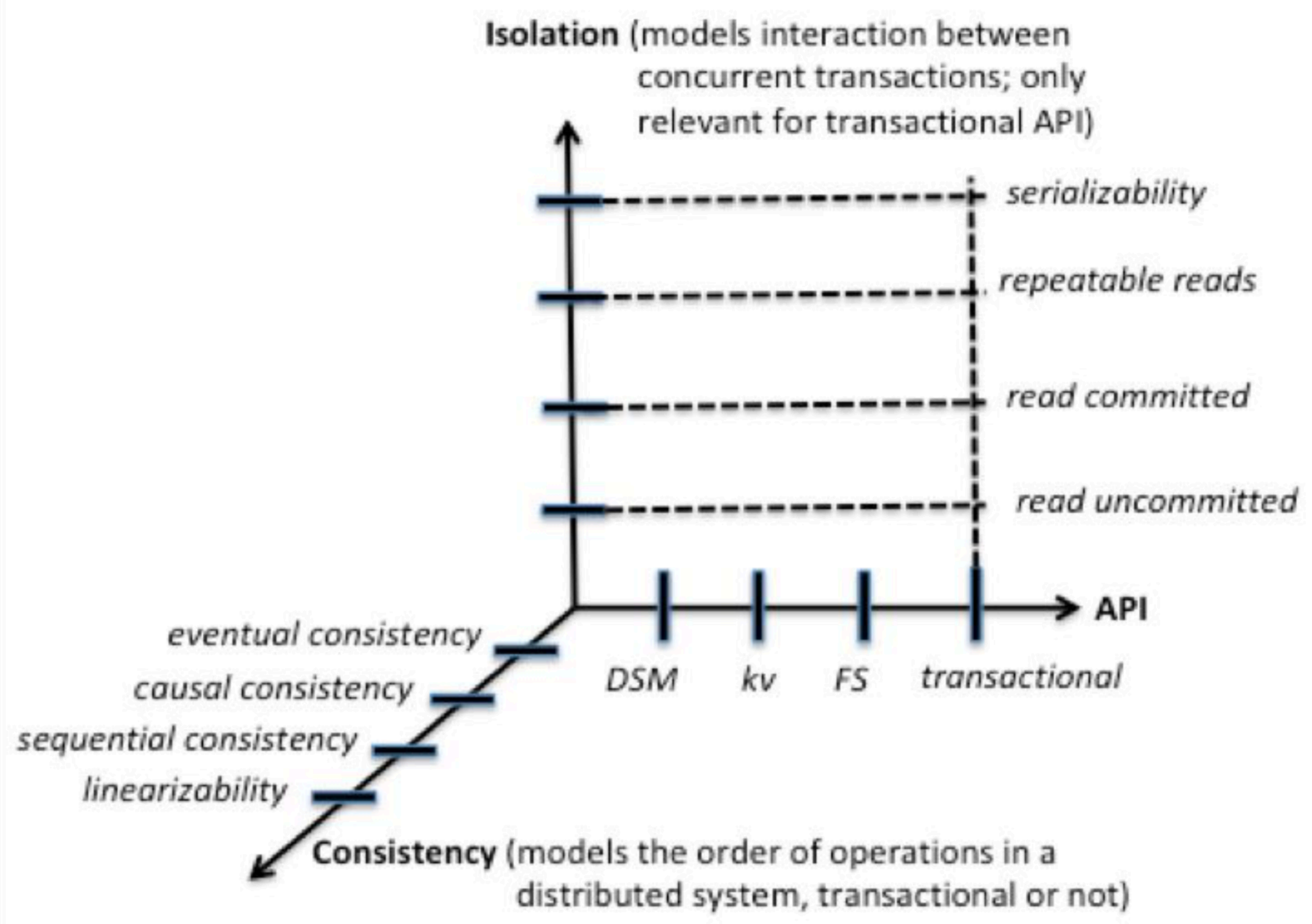
— Doesn't order writes as they are executed, which might create conflicts later: which write was first?

— Used in Amazon's Dynamo, a key/value store

  — Plus a lot of academic systems

  — Plus file synchronization

  — Plus source control systems like... git!


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)

TAKEAWAYS

- Different isolation and consistency levels have different tradeoffs
- When using weaker isolation+consistency... fun begins ☺

- Last class in "Fundamentals" section, from next week: "Real-world Cloud"
- Next class: **Guest talk from Alibaba Cloud**

ACKNOWLEDGEMENT

THIS COURSE IS DEVELOPED HEAVILY BASED ON COURSE MATERIALS SHARED BY PROF. INDRANIL GUPTA, PROF. ROBERT MORRIS, PROF. MICHAEL FREEDMAN, PROF. KYLE JAMIESON, PROF. WYATT LLOYD AND PROF. ROXANA GEAMBASU. MANY APPRECIATIONS FOR GENEROUSLY SHARING THEIR MATERIALS AND TEACHING INSIGHTS.

THIS SLIDES INCLUDES CONTENTS FROM BLOG: HTTPS://BLOG.MI.HDM-STUTTGART.DE/INDEX.PHP/2020/03/06/ISOLATION-AND-CONSISTENCY-IN-DATABASES/