
Linearizability

Key takeaways

1. Explain a concurrent execution with a correct sequential history. The correctness condition determines which operation orders are allowed.
2. Linearizability makes each call appear to happen at one instant between its call and return. A call that returns before another starts must come first.
3. Making each get and put atomic does not make a get-then-put atomic. Use an atomic increment, a transaction, or one lock around the whole sequence.

Common misunderstandings of concurrency contracts

Which claim would you defend right now? Choose before the evidence.

“If some sequential order explains every result, the execution is linearizable.”

“A read overlapping a write may return either value, regardless of other reads.”

“If individual calls are atomic, a sequence of client calls is atomic too.”

Keep your choice. Look for the mechanism or counterexample that would make it fail.

Sequential histories can be correct or incorrect

Key k starts at 0. Each column gives a sequence of calls and their results.

Correct sequential history

```
put(k, 2) → OK  
get(k)   → 2
```

Incorrect sequential history

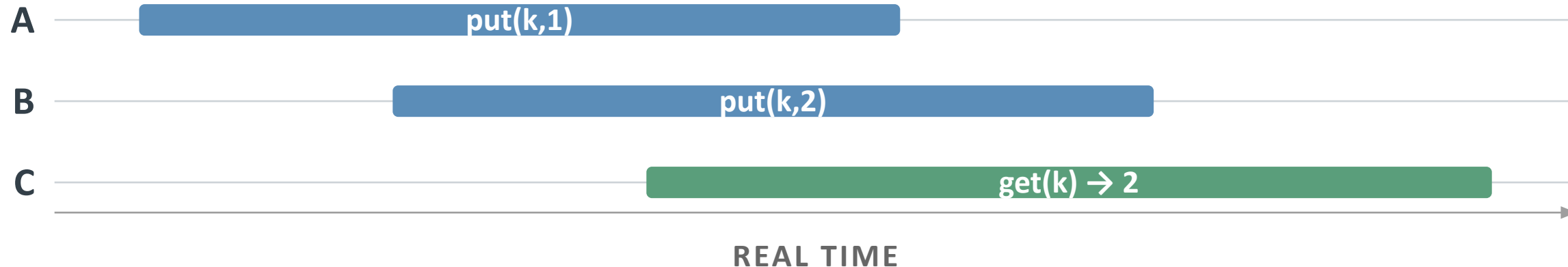
```
put(k, 2) → OK  
get(k)   → 0
```

$\text{put}(k, v)$ sets the value of k . A later $\text{get}(k)$ must return that value until another put changes it.

Familiar data structures have known sequential correctness rules: a stack pops the most recent item; a queue removes the oldest; a search tree implements its set or map operations.

Concurrent histories need an ordering rule

1. Record the concurrent history: overlapping calls and their results.



2. Choose an order. The correctness condition decides which orders are allowed.



3. Check the recorded results: in this order, get returns the last value put.

Instapoll

Instapoll · Review · Oct 6 B

In a key-value store, key k starts with value 0. Thread A calls $\text{put}(k, 2)$, which returns OK. Only then does thread B call $\text{get}(k)$, which returns 0. There are no other writes. Must every concurrency correctness condition reject this result?

- A. No; a cached old value is acceptable under every correctness condition
- B. Yes; every correctness condition must preserve real-time order
- C. No; a condition can allow get before put in the sequential explanation
- D. Yes, unless A and B ran on different physical processor cores

Linearizability is a good choice

It supports sequential reasoning, respects completed work, and composes across objects.

- 1 One instant between the call and return**
Each operation appears to take effect at a single point: its linearization point.
- 2 The points give a legal sequential history**
In that order, the recorded results must follow the object's sequential rules.

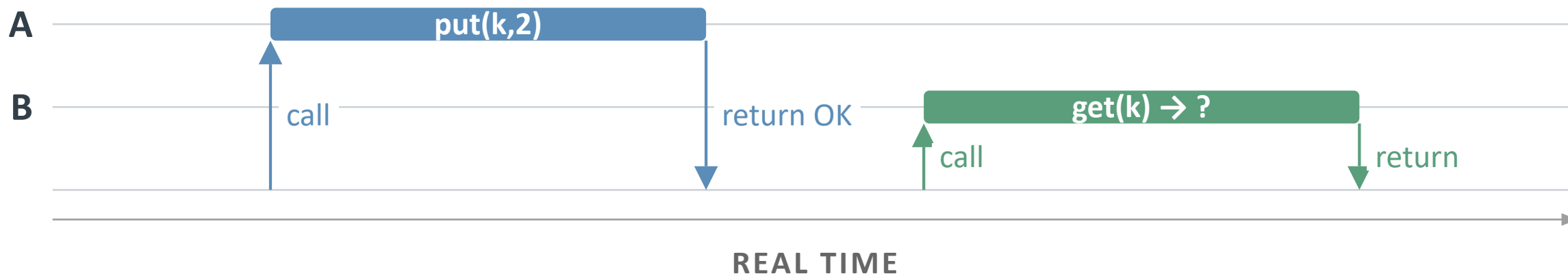
Real-time order follows: if A returns before B is called, A's point must be before B's point.



We will build an understanding by placing points in the next histories.

Linearizability preserves real-time order

Initially $k=0$. The put returns before get is called; there are no other writes.



Linearizability



Must keep put first. The get must return 2.

Serializability

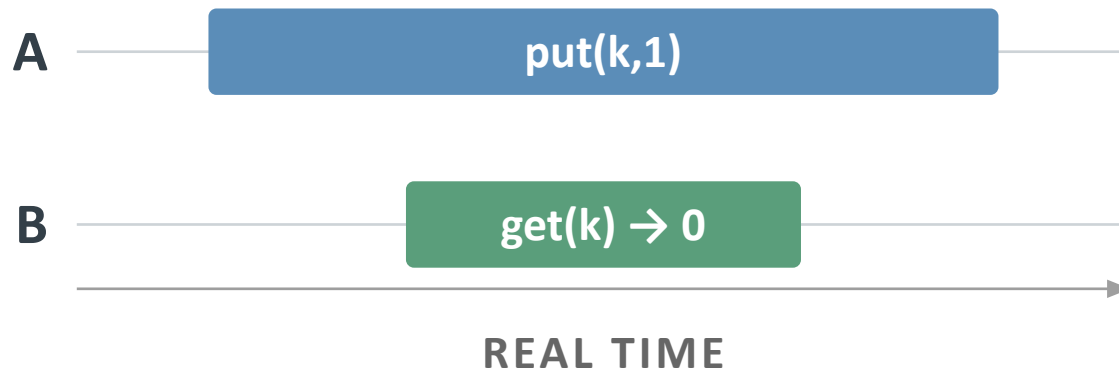


Either order is legal; get may return 0 or 2.

Both overlapping orders are linearizable

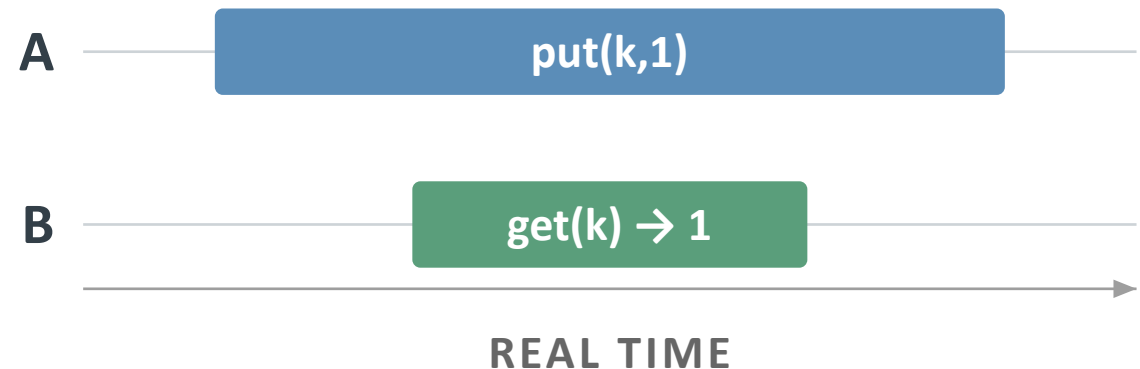
Same call intervals; k starts at 0. Each operation takes effect between call and return.

get → put



The get takes effect first and returns the initial value 0.

put → get



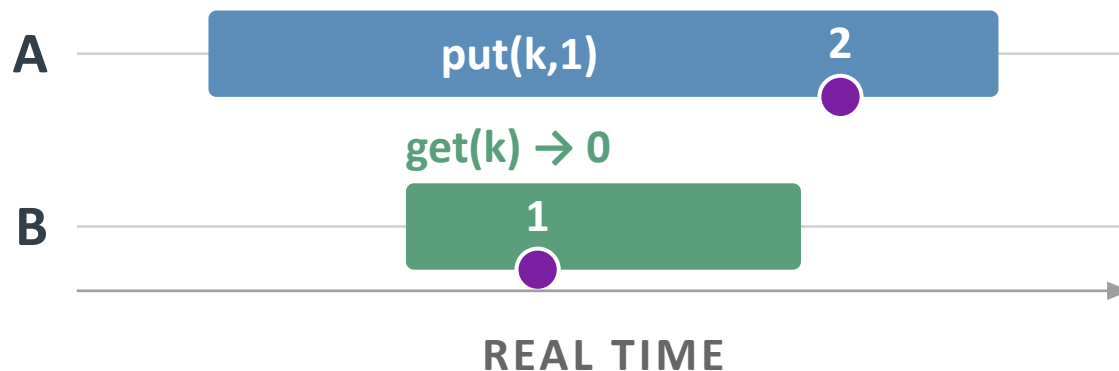
The put takes effect first, so the get returns 1.

Both are linearizable. Each order explains its recorded result.

Both overlapping orders are linearizable

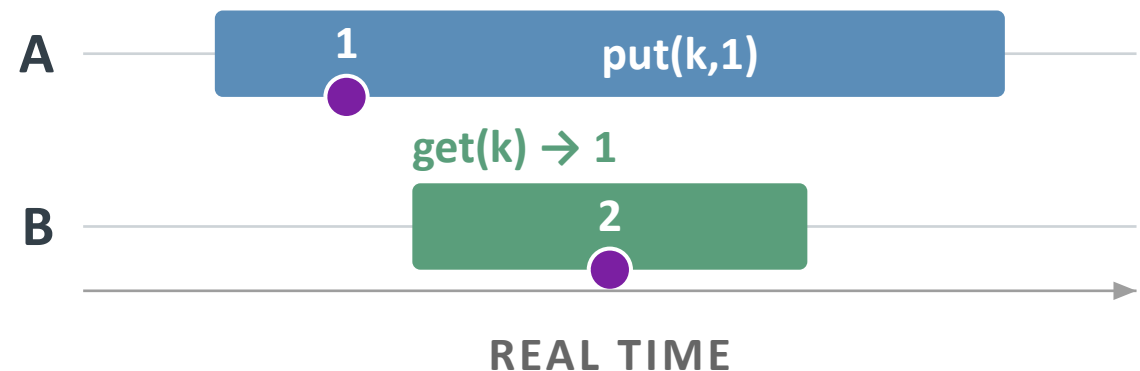
Same call intervals; k starts at 0. Purple dots mark linearization points.

get → put



The get takes effect first and returns the initial value 0.

put → get

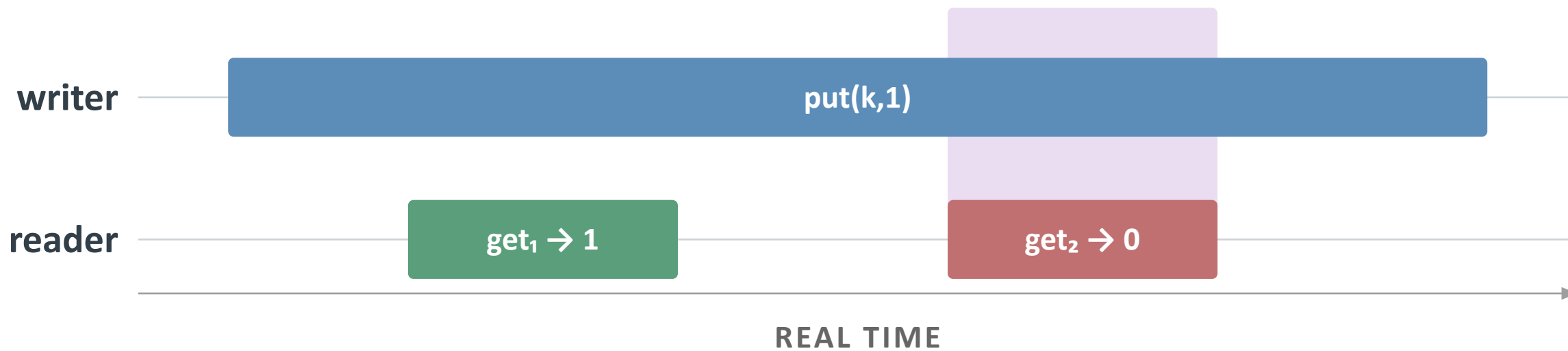


The put takes effect first, so the get returns 1.

Both are linearizable. Each order explains its recorded result.

The second get must return 1

Initially $k=0$. The only put writes 1. The gets report 1, then 0.

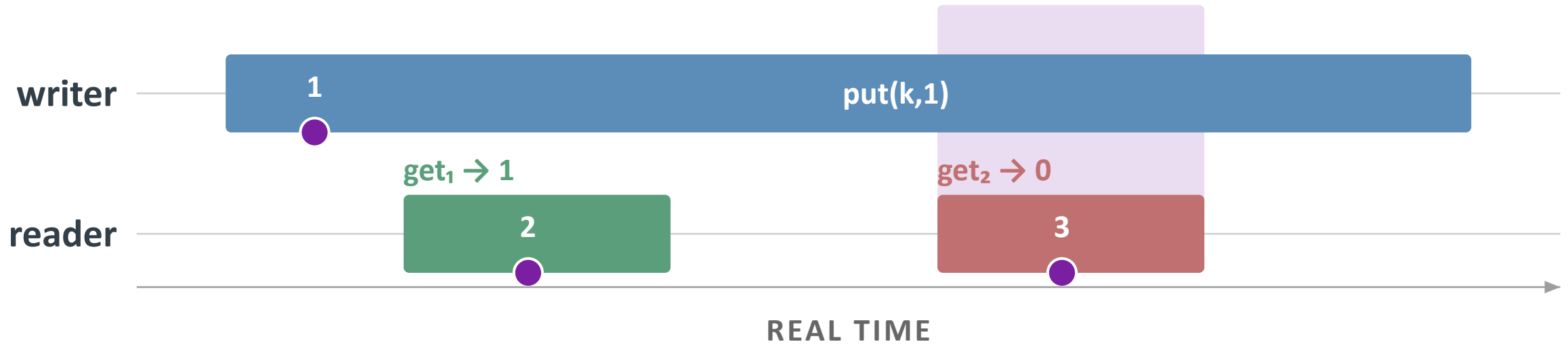


- `get1` returns 1, so the put's point must precede `get1`'s point.
- `get1` returns before `get2` is called. Every point in the shaded interval is later; no other put can restore 0.

Every point inside `get2` must read 1. Its recorded 0 is not linearizable.

The second get must return 1

Initially $k=0$. The only put writes 1. The gets report 1, then 0.



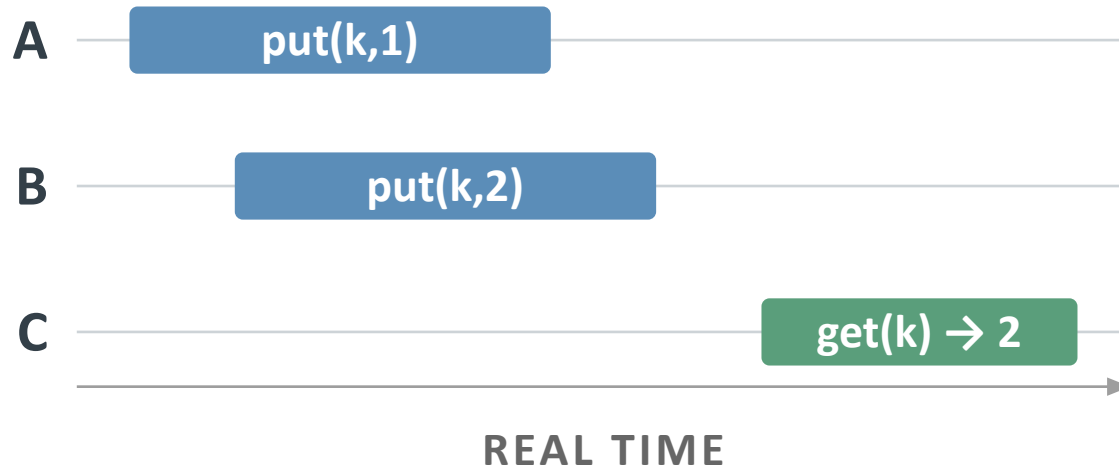
- `get1` returns 1, so the `put`'s point must precede `get1`'s point.
- `get1` returns before `get2` is called. Every point in the shaded interval is later; no other `put` can restore 0.

Every point inside `get2` must read 1. Its recorded 0 is not linearizable.

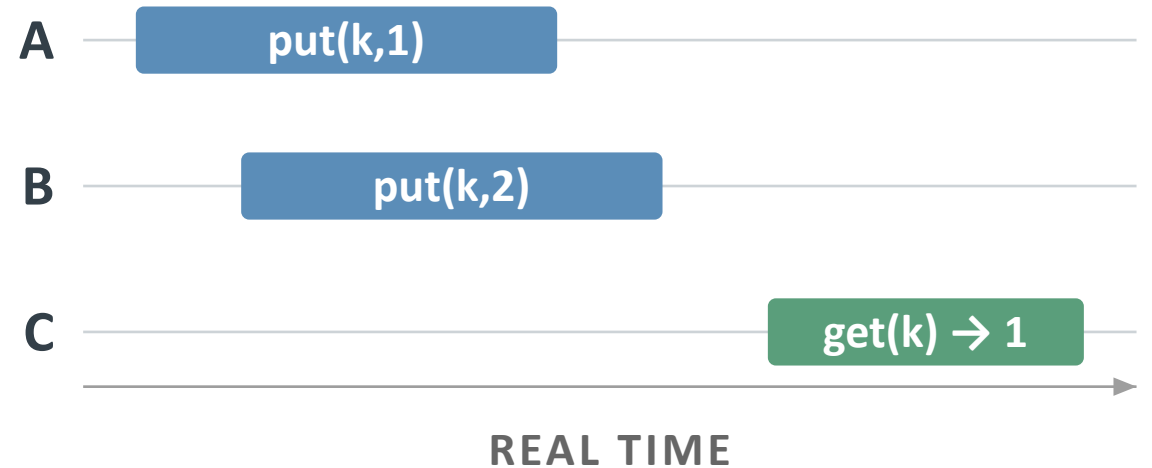
Different runs can linearize differently

Each run starts with $k=0$. Two puts overlap; get is called after both return.

Run 1: A, then B



Run 2: B, then A



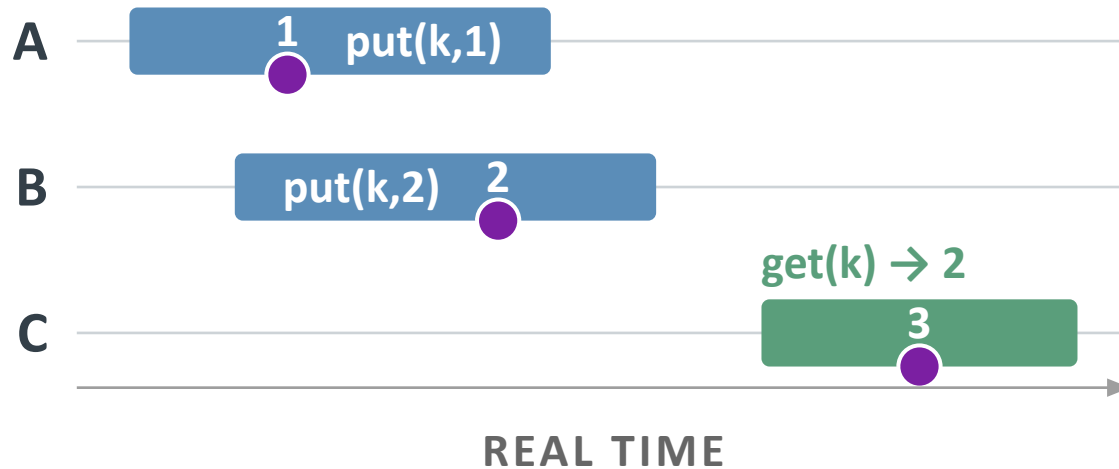
B's put takes effect last. The later get returns 2. A's put takes effect last. The later get returns 1.

Both runs are linearizable. The operations need not take effect in the same order.

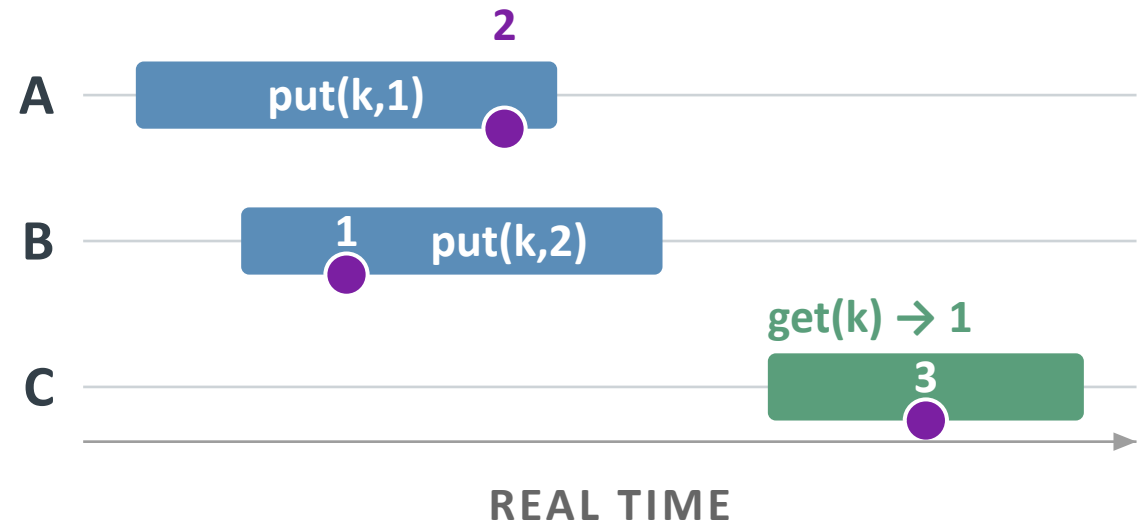
Different runs can linearize differently

Each run starts with $k=0$. Two puts overlap; get is called after both return.

Run 1: A, then B



Run 2: B, then A



B's put takes effect last. The later get returns 2. A's put takes effect last. The later get returns 1.

Both runs are linearizable. The operations need not take effect in the same order.

Linearizability composes across objects

Example: two puts overlap, then two gets overlap; keys a and b each start at 0

Key a: a legal local order

- ① `put(a, 1)`
- ④ `get(a) → 1`

Key b: a legal local order

- ② `put(b, 7)`
- ③ `get(b) → 7`

One legal order of the combined calls:



- Each call here touches one object. If each object's history is linearizable, the combined history is too, including real-time order across objects.
- This supports independent implementations and proofs. It does not require one global lock, and it does not turn several calls into one atomic call.

Atomic calls do not make an atomic increment

Each get and put is linearizable; the client wants the pair to add one

Thread A

```
① v = get(k); // returns 0  
③ put(k, v + 1);
```

Thread B

```
② v = get(k); // returns 0  
④ put(k, v + 1);
```



This is a legal order of get/put calls and a lost update for the intended increment. Use an atomic increment operation, a transaction, or a lock covering the whole sequence that all incrementing clients follow.

Key Takeaways

1. Explain the recorded calls and results with a legal sequential history. Linearizability requires that order to respect real time; serializability does not. The ordering rule is part of the correctness condition.
2. Different runs may choose different legal orders. Linearizability composes across independent objects: if each object's history is linearizable, the combined history is too, with no global lock.
3. Composition across objects is not composition across calls. An atomic get and an atomic put do not make an atomic increment; use an atomic increment, a transaction, or one lock around the whole sequence.