Linearizability

Key takeaways

1. A linearizable op takes effect at one instant between call and return.
2. Real time orders non-overlapping ops; only overlap lets you reorder.
3. Point operations are linearizable; scans stay ordered, real, and complete.

Common misunderstandings of linearizability

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

"Concurrent operations must be ordered by when their calls began."

"A read returning an older value is always a linearizability violation."

"A range scan must represent one atomic snapshot to be useful and correct."

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

The correctness bar for concurrent-index

Threads on cores, not machines across a network

- Your index returned OK for a write — but did the write actually happen?
- Linearizability is the bar that answers that: each operation appears to take effect at one instant between its call and its return
- In concurrent-index, every point operation meets that bar; an ordered scan has a weaker, explicit safety contract

“It returned OK” vs. “it happened”

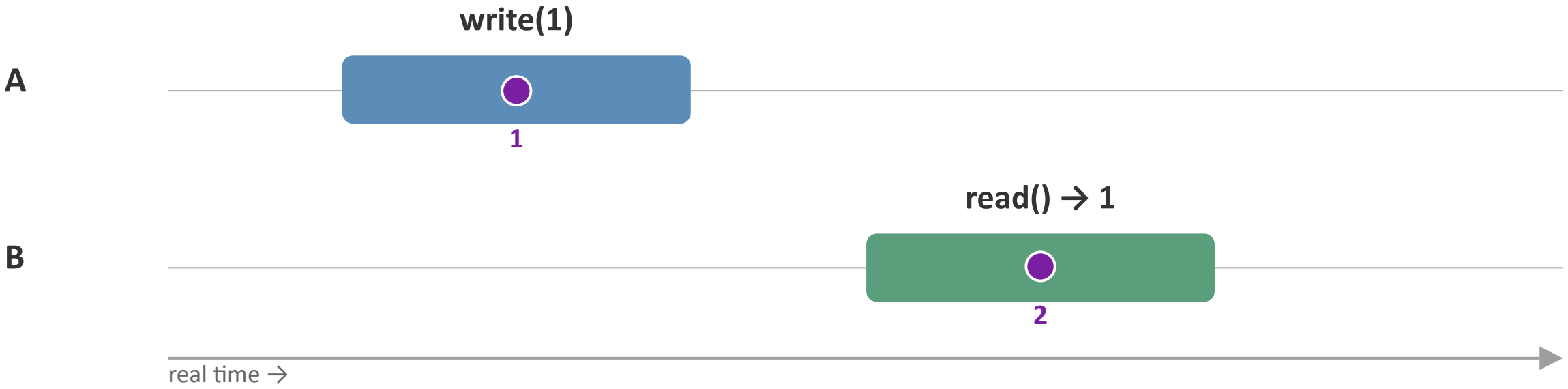
A consistency bar decides which concurrent behaviors count as correct. The strongest common bar is linearizability.

Principle (Herlihy & Shavit 3.5.1): each method call appears to take effect instantaneously at some moment between its invocation and its response.

- That instant is the operation’s linearization point — where it ‘really’ happened
- The results must match SOME single-threaded run in which each op fires at its point
- Real time is not optional: an op that already returned is already done

Read a history as timelines — find the points

Register starts at 0. Write completes, then a later read sees it. Linearizable.

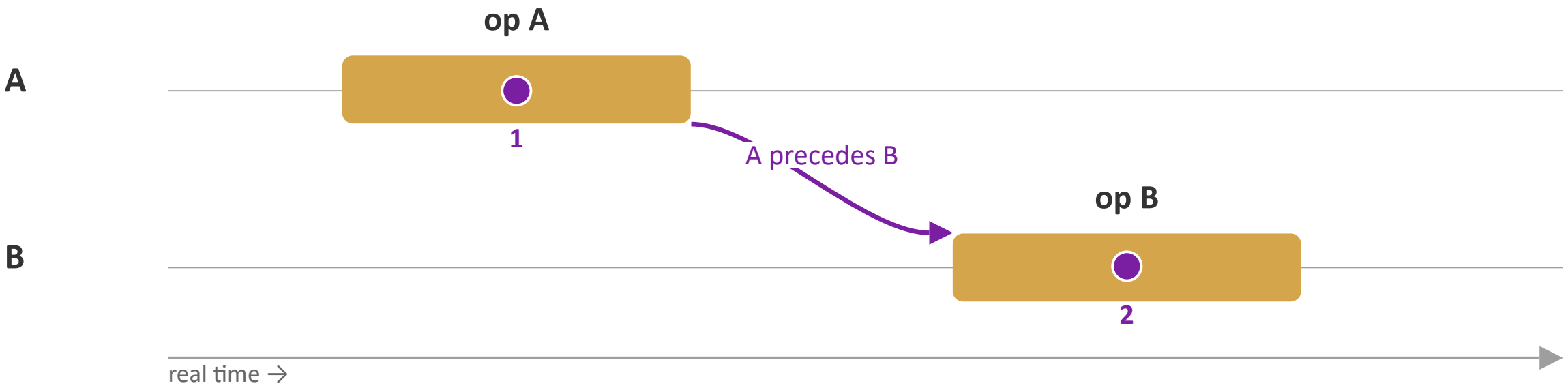


bar = operation (call → return) • dot = its linearization point • numbers = the legal order

✓ Witness: write(1) then read() → 1. Real time respected, read sees the last write.

Real-time precedence: the one hard rule

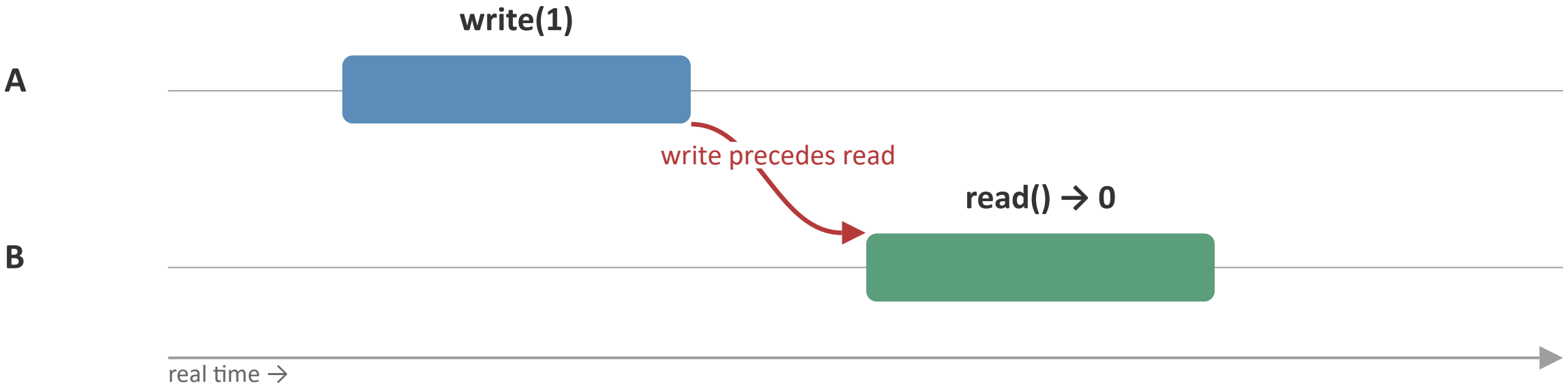
If A returned before B was called, every linearization must put A before B.



✓ Non-overlapping ops are ordered by real time. Overlapping ops are the ONLY ones you may reorder.

Stale read after ack: not linearizable

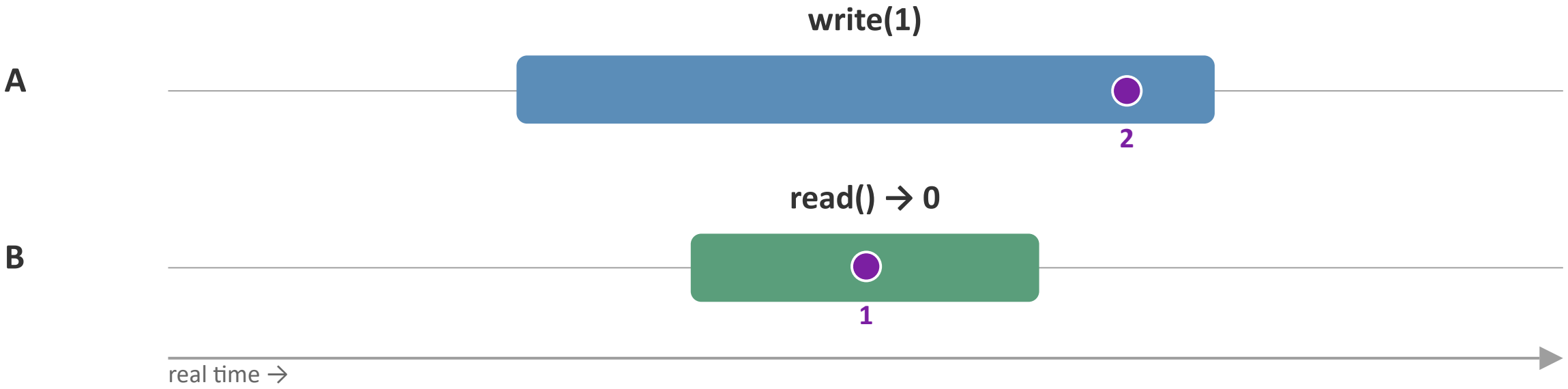
Same timing as before, but the read returns the OLD value after the write already returned.



✕ No legal order: real time forces write before read, so the read must return 1 — but it returned 0.

Overlap buys freedom: some order linearizes

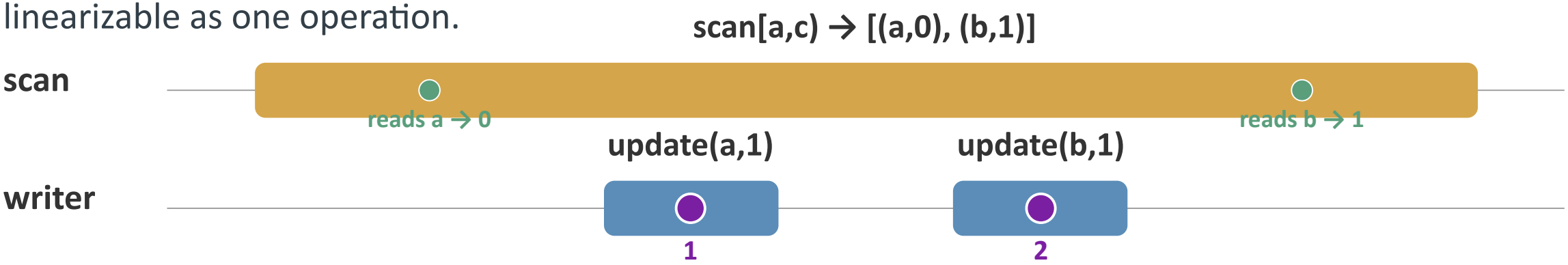
Same stale 0 — but now the read OVERLAPS the write, so no order is forced.



✓ Witness: read() → 0 then write(1). Legal, because concurrent ops may be ordered either way.

This scan result never existed as one snapshot

The scan reads a before two ordered updates and b afterward: each row is real, but the pair is not linearizable as one operation.



The whole map actually passed through:

✓ Not a linearizable snapshot — but valid for this homework: ordered, no duplicates, no invented rows, and no stable key omitted.

Anomalies with names

The data races in concurrent-index take a few recognizable shapes

Stale read

A read returns an old value after a newer write already responded. The acknowledged write is invisible to it.

Check each read against the last acked write.

Lost update

Two read-modify-writes race. Each reads the same value and writes it back, so one increment vanishes.

Make read-modify-write one atomic step.

Mixed scan

A scan reads some keys before a concurrent update and some after, so its result held at no single instant.

Permit it; still require ordered, real rows.

Name the shape first: it tells you which invariant to test, and whether the homework forbids the shape at all.

Why tests miss this: sample vs. search

A unit test fixes an input and checks one interleaving — usually the lucky one. A linearizability checker fixes the operations and searches the interleavings for any legal order.

A test SAMPLES one schedule

```
def test_write_then_read():
    kv.write("x", 1)
    assert kv.read("x") == 1
# passes every run — it never
# schedules a concurrent reader
```

A checker SEARCHES all of them

```
def linearizable(history):
    for order in orders(history):
        if respects_realtime(order) \
           and legal(order):
            return True   # a witness
    return False          # no order -> bug
```

You own the bar (what ‘correct’ means); the agent can generate the checker and fuzz the schedules — but only once you name the invariant.

Concurrent-index uses two explicit contracts

operation	required behavior
get / insert / update / delete	linearizable: one legal point per call
scan without an overlapping write	the exact sorted half-open range
scan with overlapping writes	sorted, no duplicates or invented rows; include stable keys
the checker	search point histories; validate each scan row and stable key

The weaker scan rule is deliberate, not vague. It preserves useful ordered scans while allowing designs such as Arctic that do not freeze the entire range into one snapshot.

You vs. the Agent: linearizability

Your Responsibility
Name the bar: which object spec, linearizable or weaker?
Design adversarial histories that expose the bug
Decide what ‘took effect’ means for each operation
Judge whether a proposed witness is actually legal

Agent’s Responsibility
Generate the checker; enumerate the interleavings
Fuzz interleavings and seeds at scale
Write the index code (Rust)
Minimize a failing history to a small repro

Shared: state the bar yourself — the agent can run the search, not choose it

Key Takeaways

1. Linearizable = every op appears to take effect at one instant between its call and return, consistent with real time
2. Precedence forces order, overlap frees it: a stale 0 is a bug after an ack (history b) but legal during an overlap (history c)
3. P3 splits the bar: point operations are linearizable; scans may mix overlapping writes but must stay ordered, real, and complete for stable keys