
Testing concurrent code

Key takeaways

1. A passing run covers only the schedules it produced. Make the overlap a bug needs likely: few keys, several threads, a common start, short rounds.
2. Check with an oracle that needs no schedule: invariants over final state and scans, or a recorded history some legal order must explain.
3. A hang is a failed test and a rare failure is a real one. Bound each test with a timeout; save the seed, measure the rate, and shrink the case.

Common misunderstandings of testing concurrent code

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

“A stress test that passes hundreds of runs in a row shows the code has no race.”

“Checking the final state after the threads finish catches every wrong result.”

“A failure that disappears when the test is rerun was a flaky test, not a bug.”

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

Instapoll

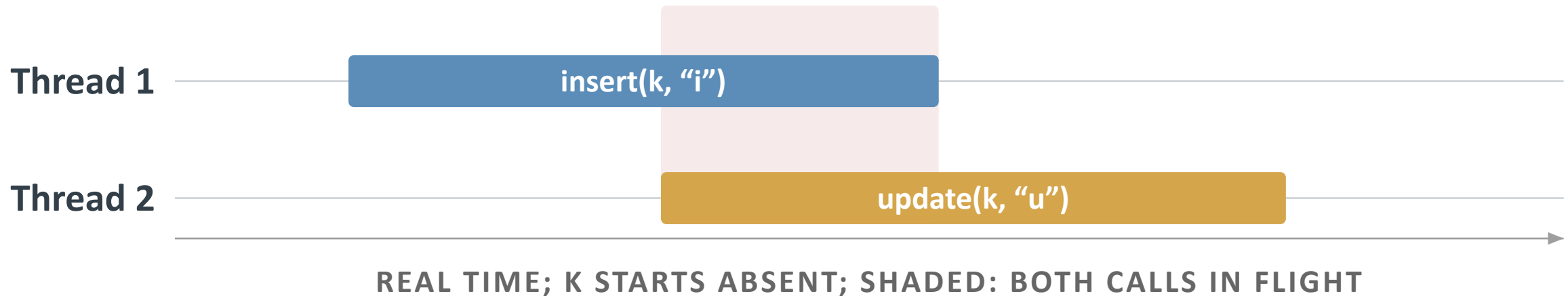
Instapoll · Review · Oct 15 A

A team's stress test runs 8 threads of random inserts, updates, and gets over 1,000,000 keys, then checks that every inserted key is present. It passes 500 runs in a row. What do those runs establish?

- A. Point operations are linearizable, because no run ended in a wrong final state
- B. The index has no race, because 500 independent schedules all passed the test
- C. The schedules those runs happened to produce passed the one check the test makes
- D. Any later failure is a flaky test, because a real bug would have appeared by now

A bug needs an overlap the test must produce

A race on key k can show only while two calls on k are in flight at once



8 threads over K keys, each on one uniformly chosen key. The chance another thread shares this operation's key (key collision only, not a bug rate):

Chance	K = 1,000,000	K = 1,000	K = 16
$1 - (1 - 1/K)^7$	about 7 in a million	about 7 in a thousand	about 36%

Raise the overlap rate on purpose

One short round, many times

```
for round in 0..ROUNDS {  
  let index = StudentIndex::new();  
  let start = Barrier::new(threads);  
  thread::scope(|s| for t in 0..threads {  
    let (index, start) = (&index, &start);  
    s.spawn(move || {  
      let _g = index.register_thread();  
      start.wait(); // common start  
      // a few ops on keys 0..16,  
      // seeded by (seed, round, t)  
    });  
  });  
  // check this round's oracle  
}
```

THE LEVERS

Few keys	16 to 64, so calls collide
Threads	2, 4, 8: new schedules
Barrier	all threads start together
Fresh state	the race's start recurs
Conflicts	insert, update, get on shared keys

Vary the seed and log it: a seed replays the operations, not the schedule.

An oracle must hold under every interleaving

Check properties every legal schedule shares, never one expected order of events

After the threads join

- Each thread inserts its own keys (key mod threads = thread id). Every one is present with the value its thread wrote.
- Every written value is unique, such as (thread, sequence number), so a returned value names the write that produced it.

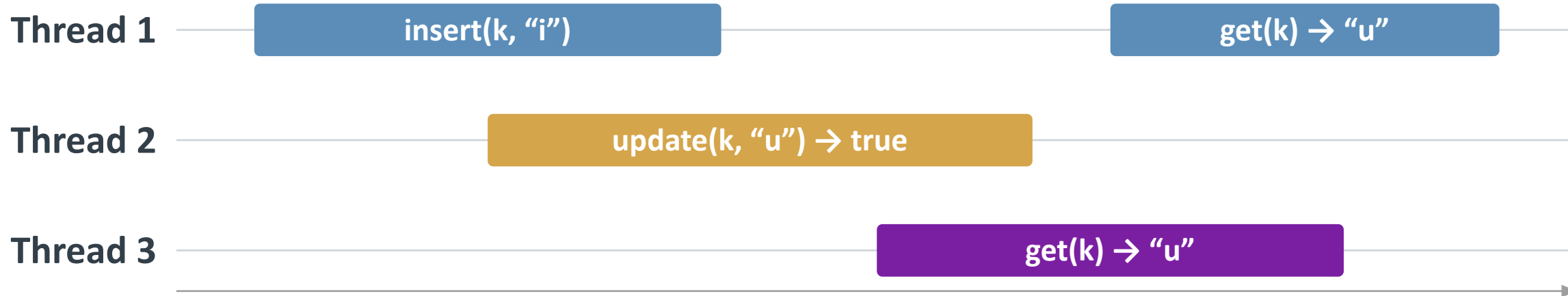
During every scan

- Keys strictly ascending: no duplicates, nothing out of order.
- Every key inside [lo, hi), and every returned pair written by some call.
- Every preloaded key the test never removes appears.

No check here depends on the order of events, so a failure is a bug, not bad luck.

Record the history: call, return, result

One legal order explains these results: insert, update, then both gets



LOGICAL TIME: ONE SHARED COUNTER STAMPS EVERY CALL AND RETURN

```
// harness crate: an empty index, a unique value per write
let history = record_history(&regime, &index);
assert!(check(&history, REQUIRED_BAR), "seed {seed}: {history:?}");
```

A checker searches for one legal order

The Linearizability lesson's rule: each call takes effect at one instant between its call and return

Overlapping calls	Candidate orders
3	6
5	120
8	40,320
12	479,001,600

When k calls all overlap, real time fixes none of their order: $k!$ candidates. Pruning helps; the worst case still grows exponentially.

The supplied check splits point operations by key, so its cost tracks how many calls on one key overlap.

Its scan check never rejects a correct index, and some subtle violations pass. A pass is evidence, not proof.

Run many short histories with a few overlapping calls per key, not one long one.

What a final-state check cannot see

update(k, “u”) races insert(k, “i”) on an absent key; afterward, the test reads get(k)

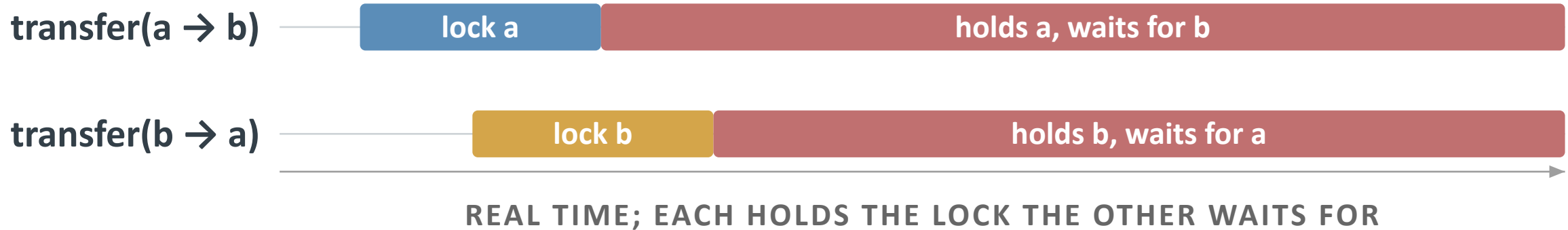
update returns	then get(k)	Verdict	Why
false	“i”	legal	update ran first and found no key
true	“u”	legal	insert ran first; update overwrote it
true	“i”	illegal	update found the key, yet its write is lost
false	“u”	illegal	update found no key, yet its value is stored

Row 3 ends with the same state as row 1: k holds “i”. Only update’s recorded return value separates the legal run from the illegal one.

Keep each call’s result. A final-state check passes this bug.

Deadlock needs a cycle of waiting

transfer(from, to) locks from, then to; two transfers run in opposite directions



One global lock order

Every transfer locks the lower account id first. Every wait then points to a higher id, so no cycle forms.

Try, then back off

If try_lock on the second fails, release the first and retry. No wait lasts, but a thread can lose every retry: starvation, as in Concurrency Fundamentals.

A test that hangs has failed

cargo test has no per-test timeout, so a deadlocked test waits forever

A watchdog you write: run the round on its own thread; wait with a limit

```
fn within<T: Send + 'static>(
    limit: Duration, f: impl FnOnce() -> T + Send + 'static,
) -> T {
    let (tx, rx) = mpsc::channel();
    thread::spawn(move || { let _ = tx.send(f()); });
    rx.recv_timeout(limit)
        .expect("no result in time: hung or panicked")
}
```

- Stuck threads stay stuck, but the test fails, reports, and the suite moves on. Print the round's seed in the failure.
- Set the limit far above a normal round's time, so a slow machine cannot fail a correct run.

Safe Rust rules out data races only

Not race conditions: every access below holds the lock, and two withdrawals still overdraw

Compiles in safe Rust; two threads can both pass the check

```
fn withdraw(balance: &Mutex<i64>, amt: i64) -> bool {  
    if *balance.lock().unwrap() < amt { return false; } // check  
    *balance.lock().unwrap() -= amt; // act: the lock is taken again  
    true  
}
```

- The race sits in the gap between two critical sections, where the compiler has nothing to check. Two threads withdrawing 100 from 100, three runs of 400,000 rounds on one laptop: 2, 2, and 2 overdrafts.
- unsafe moves the proof to you: write the invariant each unsafe block relies on beside it, as a `// SAFETY:` comment, and test it.

The compiler's silence is not a test result.

A rare failure is a lead, not noise

A sleep or a retry moves the window; it does not close it

1

Keep the seed and the history

The seed replays the operations; the history shows results no order explains.

2

Measure the rate over a loop

At 1 failure in 200 runs, 200 clean runs in a row still happen 37% of the time.

3

Shrink while it still fails

Fewer threads, fewer keys, shorter rounds: a smaller history is easier to explain.

4

Keep it as a regression test

After the fix, 600 clean runs bound the rate below 1 in 200 at 95% confidence.

When an agent says it passed on rerun, ask for the seed, the failing history, and the failure rate before and after the fix over the same number of runs.

Rerun the whole suite after every design change: the final capture's correctness gates its performance credit.

Key Takeaways

1. A passing run covers only the schedules it happened to produce. Make the overlap a bug needs likely: few keys, several threads, a common start, and many short rounds of conflicting operations on fresh state, over varied seeds.
2. Check with an oracle that does not need the schedule: invariants over the final state and each scan, or a recorded history some legal order must explain. Only recorded results expose an illegal run whose final state looks legal.
3. A hang is a failed test and a rare failure is a real one. Remove deadlock with one lock order and bound every test with a timeout you write. Save the seed, measure the rate, shrink the case, and keep it as a regression test.

Design the test that catches it

10 min

- A team suspects `update(k, "u")` racing `insert(k, "i")` on an absent key can return `true` while `k` ends holding `"i"`. Their test runs 2 threads of 1,000,000 random operations each over 1,000,000 keys and checks only that nothing panics.

update(k) returns	final get(k)	Verdict
false	"i"	legal
true	"u"	legal
true	"i"	the suspected bug

- Write a test plan: threads, key-space size, round structure, operations per round, number of rounds, and the oracle.
- Add one sentence arguing why your test fails if the bug exists.
- Decide: can a final-state invariant catch this bug, or must the test check recorded results?