Contention & False Sharing

Key takeaways

1. Pad unrelated hot fields that share a cache line or they still contend.
2. Judge concurrent designs by the scaling curve, not one peak throughput.
3. Choose concurrency control for the workload; let measurements defend it.

Common misunderstandings of contention and false sharing

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

"Threads writing different variables cannot interfere with one another."

"If throughput rises at one thread count, the design scales."

"Lock-free code avoids synchronization costs and memory-management tradeoffs."

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

P3 scores scaling, not asymptotics

Cores, caches, and one 64-byte line set the scaling curve

- Correctness is the gate (linearizable point operations and safe ordered scans); the SCORE is how throughput curves with thread count
- That curve is decided by cores, caches, and one 64-byte cache line — not by the asymptotics of your data structure
- You pick a structure × a concurrency-control scheme and defend it; this deck is the physics behind that choice

The memory system is the arbiter

Cores do not share memory one byte at a time — they share it one cache line at a time. The line (64 bytes on x86-64) is the unit of coherence: when any core writes one byte in a line, every other core's cached copy of the WHOLE line is invalidated and must be re-fetched.

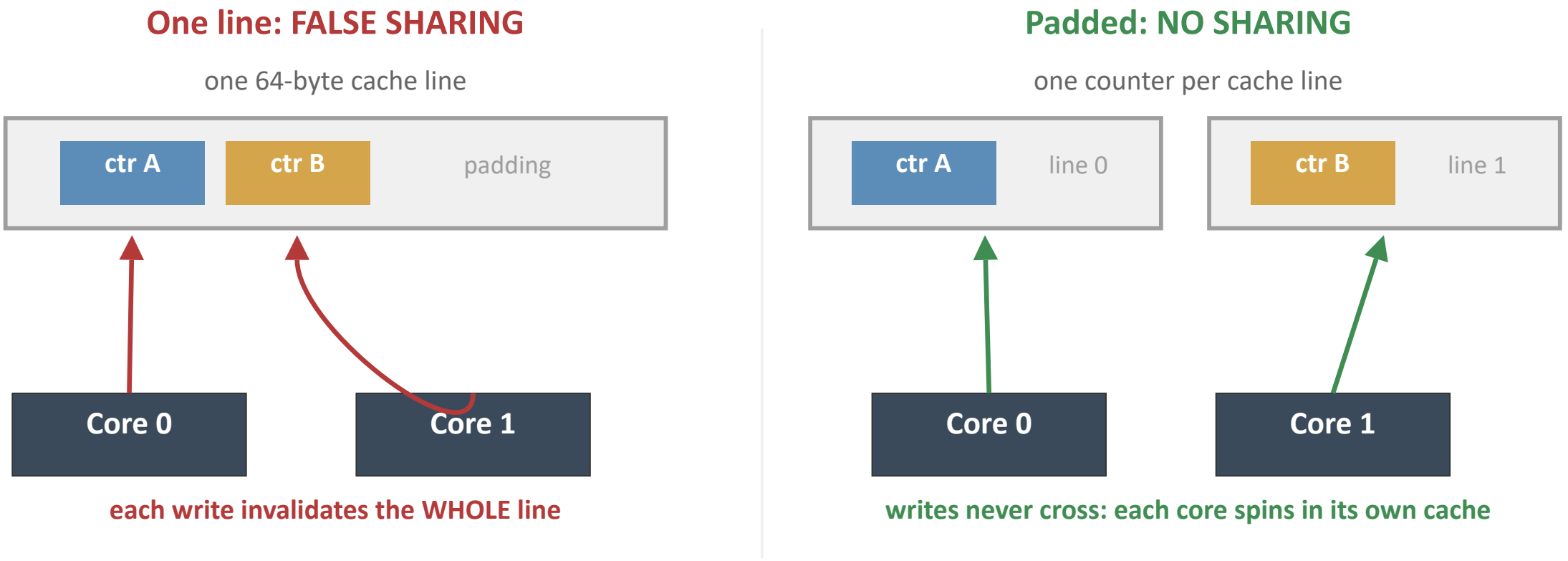
The coherence unit, and the floor you are scored against

```
// The unit of coherence is the line, not the byte.
const CACHE_LINE: usize = 64;   // bytes, x86-64

// The P3 baseline: one lock over one BTreeMap. Correct, and
// FLAT — one core in the critical section at a time.
struct GlobalLockIndex { inner: RwLock<BTreeMap<Key, Val>> }
```

The graded run is on a many-core Linux server. Contention that is invisible at 8 cores can dominate at 20+ — so the curve, not one peak number, is the evidence.

False sharing: two counters, one line

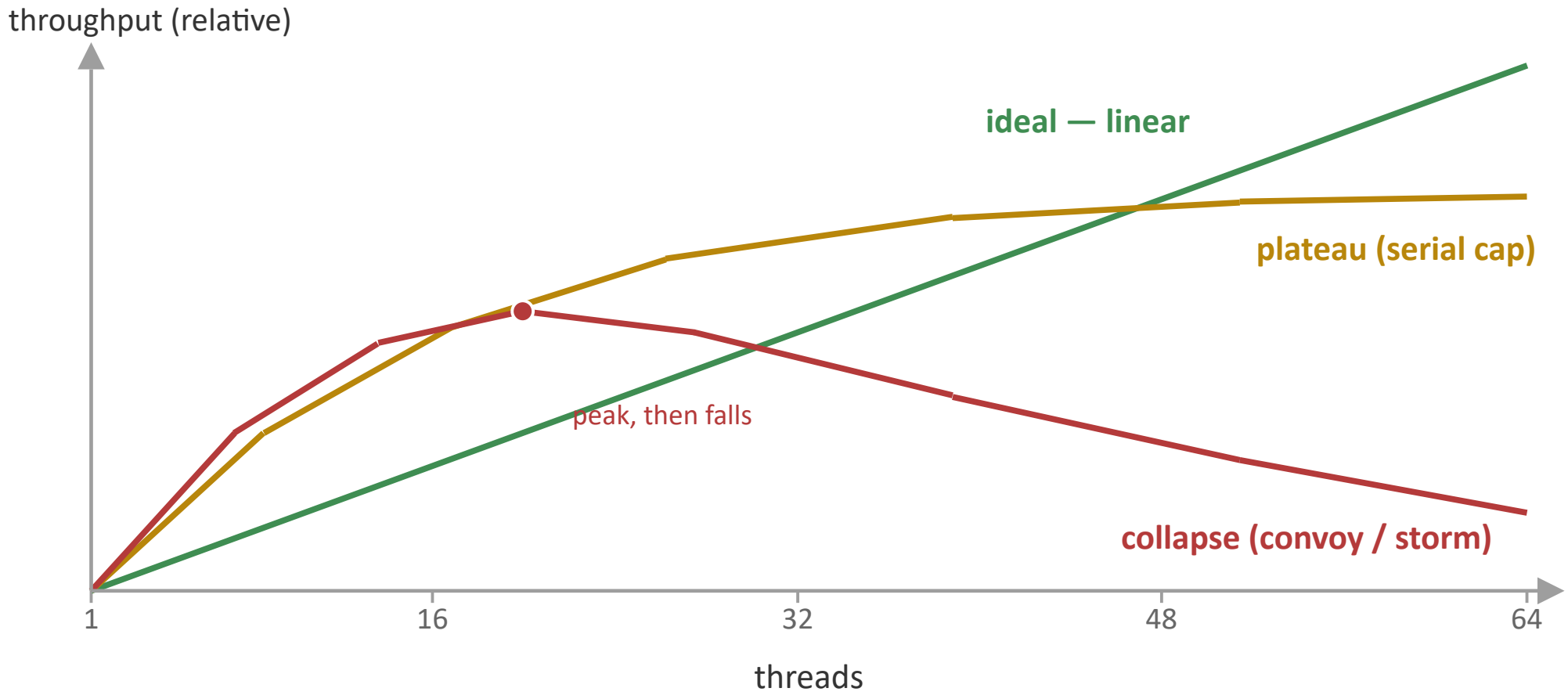
Two unrelated counters, zero shared data — one shared cache line makes them fight anyway.



✓ Same data, 10x apart: padding to distinct lines is the fix — and in P3 it is your code (#[repr(align(64))]), not the allocator.

Contention shapes: read the curve

Three ways a design scales. A flatter curve that holds beats a tall one that collapses.

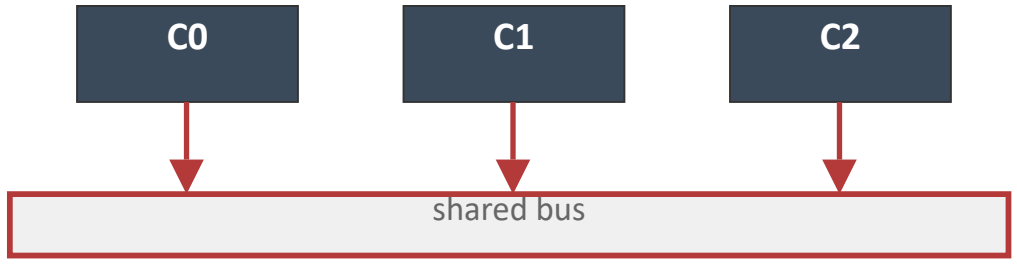


Why waiting politely matters: TAS vs. TTAS

A spin that WRITES broadcasts every iteration; a spin that READS stays in local cache until the lock frees. The write-spin is why more threads make a lock design SLOWER.

TAS — spin = swap (an atomic write)

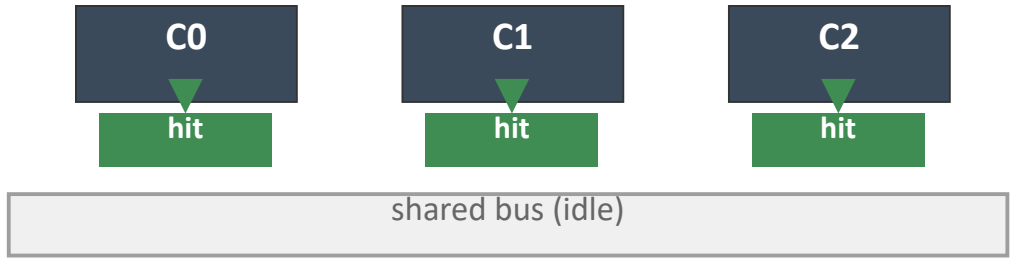
```
while lock.swap(true, Acquire) {
    spin_loop();
}
// every swap storms the bus
```



every iteration broadcasts — a coherence storm

TTAS — spin = load (a read)

```
loop {
  while lock.load(Relaxed) { spin_loop(); }
  if !lock.swap(true, Acquire) { break; }
}
// reads hit cache; swap only when free
```



reads stay in cache — one storm only at release

Design levers: how you answer contention

No lever wins every regime — match it to where the workload leans

Striping / sharding

Split the data into N stripes, one lock each, for N-way parallel writes. A single hot key still bottlenecks its own stripe.

Pad each stripe onto its own cache line.

Snapshot / COW

Immutable root; a writer CASes in a new version. Readers never block, scans get an atomic snapshot, writes amplify.

Choose it when reads outnumber writes.

Optimism (OCC)

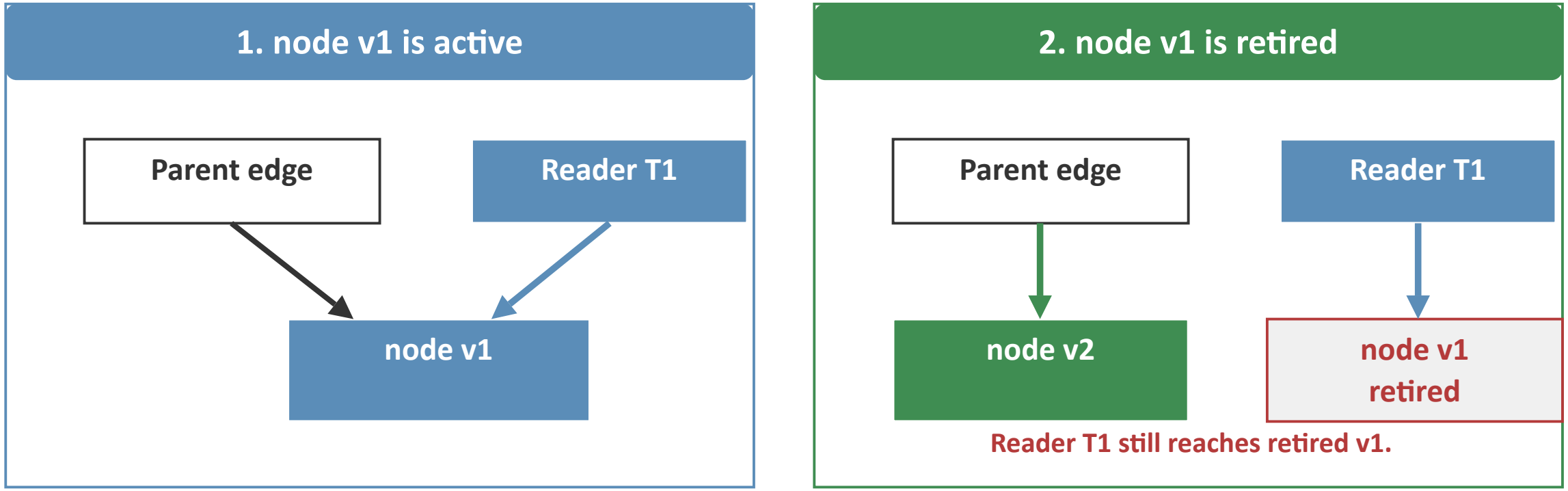
Read without locking, then validate a version stamp and retry on conflict. Cheap uncontended, retry storms under heavy writes.

Measure the retry rate before you trust it.

Contention is a property of the workload, not a bug in your code: pick the lever that matches where the reads and writes actually land.

Memory reclamation: why free() is hard

Unreachable is not safe to free while a reader holds the old pointer



Epochs and hazard pointers both delay free until no reader can still reach v1.

In P3, ThreadGuard records which readers may still reach retired nodes.

Epoch guards make the waiting rule executable

Pin before load; retire after unlink; reclaim after old readers exit

The reader and writer cooperate through a guard

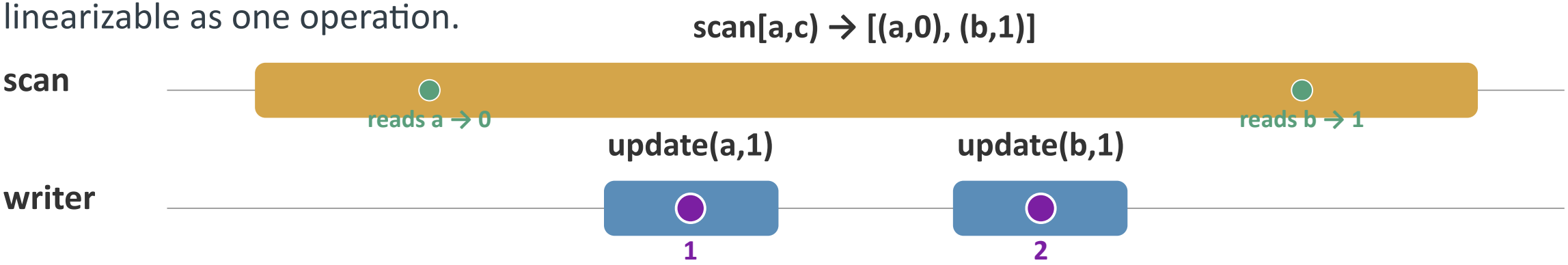
```
let guard = epoch::pin();                    // reader becomes visible
let node = root.load(Acquire, &guard);
walk(node);                                  // v1 stays alive

let old = root.swap(v2, Release, &guard); // unlink v1
unsafe { guard.defer_destroy(old); }      // retire now; free later
drop(guard);                                 // reader releases protection
```

- Reader: pin before loading; reached nodes stay alive while the guard lives.
- Writer: unlink first, then defer destruction instead of freeing immediately.
- Reclaimer: free only after readers from the retirement epoch have exited.

P3’s ThreadGuard enforces this lifetime after register_thread().

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 P3: ordered, no duplicates, no invented rows, and no stable key omitted.

Reprise: points are atomic; scans may advance

Same linearizability bar as Week 5 for get, insert, update, and delete. A scan has a deliberately weaker contract when writes overlap it.

A concurrent scan stays in key order, returns no duplicate or invented row, and includes every in-range key present for its whole run. Without an overlapping write, it returns the exact range.

One legal design: visit ordered shards separately instead of freezing one snapshot

```
fn scan(&self, lo: &[u8], hi: &[u8], sink: &mut Sink) {
    for shard in shards_in_order(lo, hi) {
        let guard = shard.read();
        emit_range(&guard, lo, hi, sink);
    } // later shards may include overlapping writes; rows stay ordered
}
```

Rust for directors: the workspace

You are steering an agent through Rust, not writing it fluently. Two facts keep you in control. First: it is a cargo workspace, and your code lives in exactly one crate.

One workspace, many crates; you edit index-student only

```
crates/
  index-api/       # the FROZEN `trait Index` -- do not edit
  index-student/   # StudentIndex -- your code lives HERE
  probe/           # local self-eval: gate verdict + practice vector

$ cargo run --release -p probe    # your current verdict, any time
```

What the borrow checker buys: in concurrent code it rejects data races at COMPILE time. That is why the correctness gate is about logic bugs (a lost update or malformed scan) — not segfaults. Trust that fence; spend your review on the design, not on use-after-free.

Rust for directors: Send + Sync

The harness shares ONE &dyn Index across every worker with no harness-side lock. So the frozen trait demands Send + Sync — and that single bound is the whole assignment in disguise.

The smell: silence the compiler

```
struct StudentIndex {
    map: UnsafeCell<BTreeMap<..>>,
}
// compiler: not Sync. "Fix":
unsafe impl Sync for StudentIndex {}
// a promise you now must keep by hand
// -- a red flag in review, not a fix
```

The design: earn Sync

```
struct StudentIndex {
    shards: Box<[Mutex<Shard>]>,
}
// Sync for free: every &self op hashes
// the key to ONE shard's Mutex. The
// synchronization IS the design you
// are graded on -- no unsafe needed.
```

Send = ownership can move to another thread; Sync = &T is safe to share. Tell the agent: make it Sync WITHOUT a global lock.

You vs. the Agent: concurrent-index

Your Responsibility	Agent's Responsibility
Pick the structure × concurrency-control operating point	Write the Rust: the split logic, the OCC validation loop
Predict the scaling curve; decide what regime to sacrifice	Add #[repr(align(64))] padding where you point it
Choose the cache-line layout: what gets padded, what gets packed	Wire epoch reclamation; make the type Send + Sync
Judge honesty: ratio-to-baseline, not laptop peak numbers	Run the probe, minimize a failing history to a repro

Shared: the agent can build any lever — only you can choose which one to defend

Key Takeaways

1. The cache line is the unit of coherence: two unrelated items on one line contend anyway (false sharing) — pad them apart
2. Read the scaling CURVE, not the peak: linear vs. plateau (a lock serializes everyone) vs. collapse (a convoy or a TAS coherence storm)
3. You own the lever (striping / snapshot-COW / optimism) and the honesty; the agent writes the Rust and the borrow checker kills the segfaults

predict the scaling curves

10 min

- Your pair gets one workload: read-heavy uniform or write-heavy hot-set. Choose two designs:
- 1) global RwLock<BTreeMap>
- 2) 64 striped Mutexes; adjacent counters
- 3) copy-on-write B+tree; CAS-swapped root
- 4) OCC skip list; version-validated writes
- Sketch each design’s throughput at 1, 8, and 64 threads. Choose which you would measure first and name its likely bottleneck.
- Join a pair with the other workload. Identify one ranking reversal or explain why there is none. Deliverable: two curves and one disagreement.