Concurrency Fundamentals

Key takeaways

1. One source line is not one indivisible action. x = x + 1 is a read, an add, and a write, so two threads running it can interleave and lose an update.

2. Mutual exclusion fixes that by making the whole read-modify-write indivisible. In Rust a MutexGuard marks that interval: it acquires on construction and releases when it drops.

3. One global lock caps speedup because independent work serializes. Finer locks can scale, but each one adds safety, progress, and fairness obligations you must prove.

Common misunderstandings of concurrency

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

"One source line is one indivisible machine action."

"A mutex chooses one deterministic order for the threads."

"A correct global lock keeps speeding up as cores are added."

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

How many different final values can x have?

x starts at 0; two threads each execute the same statement exactly once

The problem

```
x = 0;

// Thread A, once          // Thread B, once
x = x + 1;                 x = x + 1;
```

- Assume an assignment evaluates its right-hand side, then writes the result.
- Count distinct final values — do not count schedules that end at the same value twice.

Question: how MANY different values can x contain after both threads finish?

Instapoll

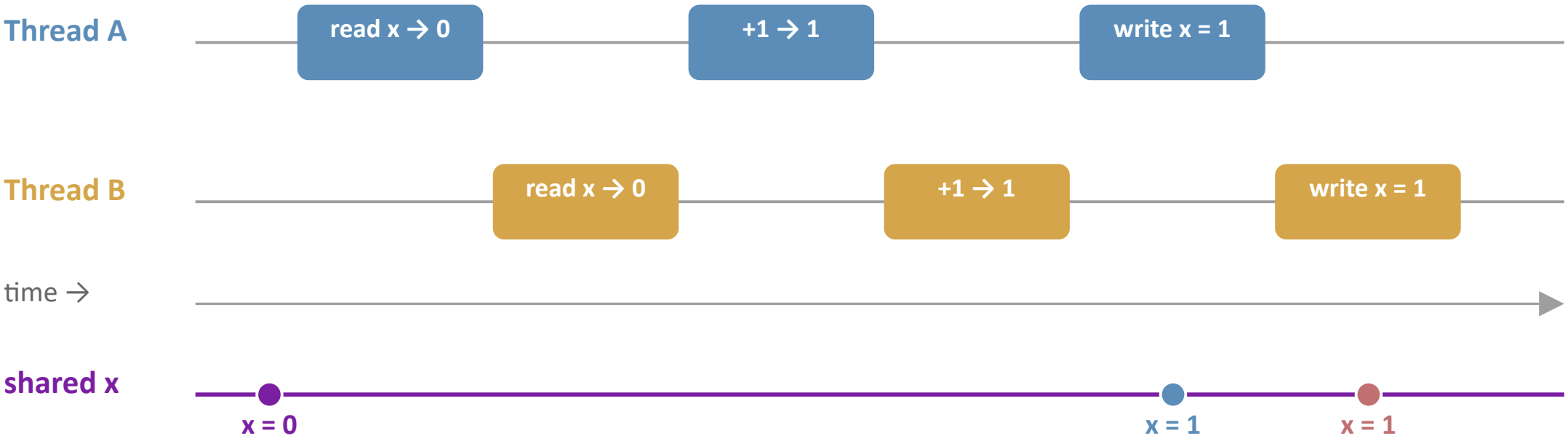
Instapoll · Review · Oct 6 B

X starts at zero. Two threads each execute x = x + 1 exactly once. How many different final values of x are possible?

- A. Exactly one outcome is possible: 0
- B. Exactly one outcome is possible: 1
- C. Exactly one outcome is possible: 2
- D. Two outcomes are possible: 1 or 2

One legal interleaving loses an update

The colors are threads; both execute the same read → add → write steps



Both threads computed from 0: two increments ran, but the final value is only 1.

Mutual exclusion rules out overlap

At most one thread may execute the protected critical section at a time

Conceptual protocol

```
lock(m);
x = x + 1;        // critical section
unlock(m);
```

- Mutual exclusion: no two threads are inside the same protected region together.
- The lock protects the whole read-modify-write, not the individual read or write.
- The order may be A then B or B then A; correctness must hold for either order.

A critical section is code whose shared-state invariant requires exclusive access.

Rust expresses locking with RAII

Resource acquisition is initialization: the guard's lifetime is the lock's lifetime

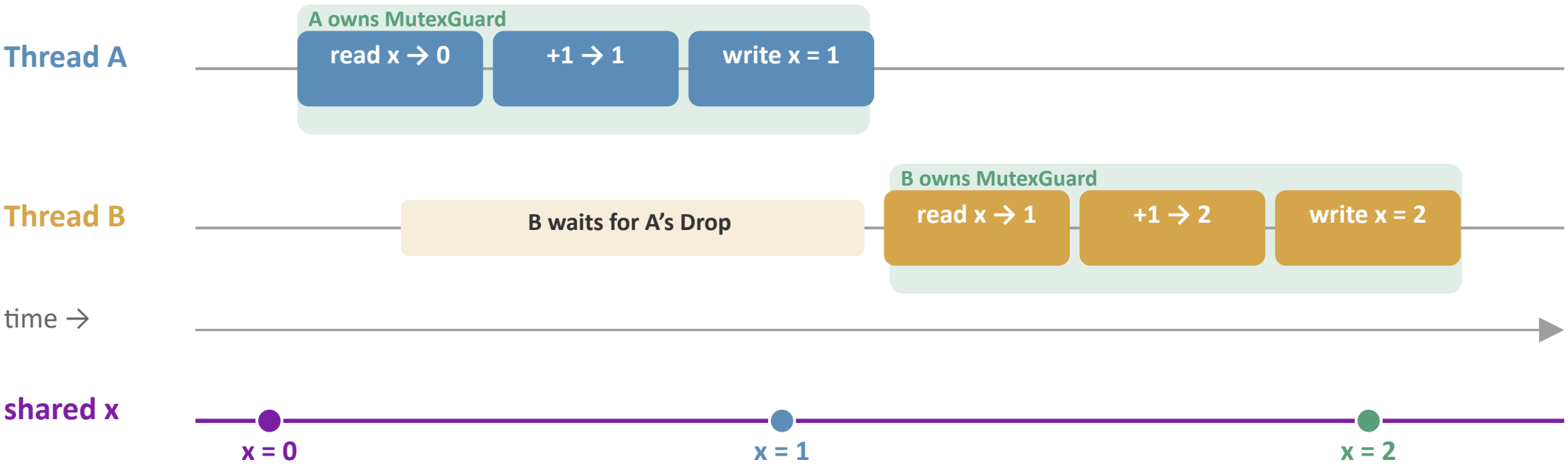
MutexGuard makes the critical-section boundary visible

```
fn increment(x: &Mutex<i32>) {
    let mut guard = x.lock().unwrap();  // acquire; guard owns lock
    *guard = *guard + 1;                // protected x = x + 1
}                                       // Drop guard; release lock
```

- Acquire constructs a MutexGuard; the protected value is reached through the guard.
- Scope controls release: every normal exit runs Drop, including an early return.
- The type prevents an unlocked reference from casually escaping the critical section.

The guard makes the timeline exclusive

Same lanes and events; Thread B waits until Thread A's guard drops



Disjoint guard lifetimes force the two updates to take effect one after the other.

Coarse locks are simple; fine locks can scale

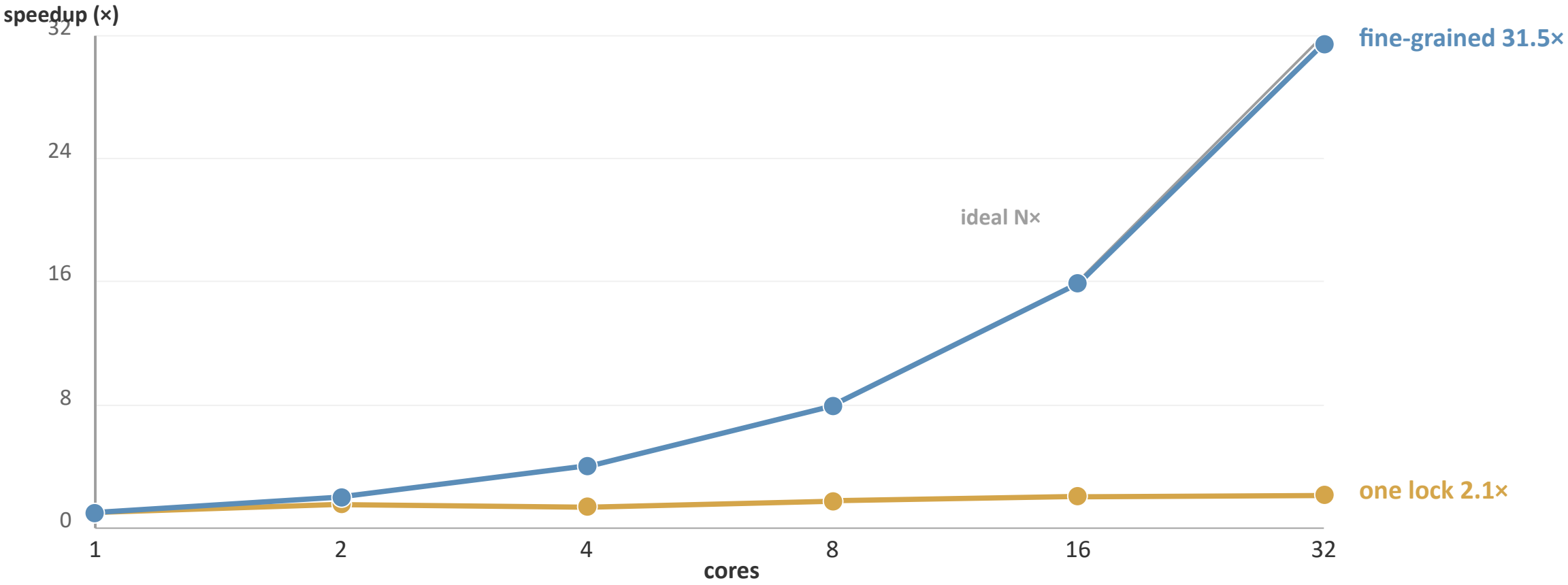
Follow the original rule of thumb — then demand measurement before adding complexity

	Coarse-grained concurrency	Fine-grained concurrency
Design	Easy: one broad ownership rule	Hard: many ownership and ordering rules
Safety	Easily made safe	Difficult to make safe
Performance	Often poor: independent work serializes	Can scale: independent work overlaps

Start with a safe coarse baseline; keep finer locks only when measured speedup earns their proof cost.

Speedup exposes the cost of one global lock

speedup(N) = throughput(N) / throughput(1); ideal fixed-work scaling follows N



At 32 cores, the one-lock design reaches only ≈2.1×; fine-grained locking reaches ≈31.5×.

Source: concurrent-index scaling-host experiment, Aug. 2026; controlled uniform-key point operations; rounded median speedups; not a grading curve.

Lock correctness has more than one axis

A mutually exclusive lock can still hang or starve a caller

Obligation	Question to ask	Failure shape
Safety	Can two threads enter together?	Broken invariant / data corruption
Progress	Does some waiter eventually enter?	Deadlock or livelock
Bounded waiting	Can one caller wait forever?	Starvation / unfairness
Failure behavior	What if the holder panics or dies?	Poisoned or abandoned state

Test-and-set changes ownership atomically

Hardware changes ownership in one indivisible read-modify-write operation

Teaching pseudocode

```
fn lock() {
    while test_and_set(&LOCK) == LOCKED {
        spin();
    }
}

fn unlock() {
    clear(&LOCK);
}
```

- test_and_set returns the old value and writes LOCKED as one atomic action.
- Exactly one caller can observe FREE and become the owner.
- A real mutex also adds sleeping, wakeups, fairness policy, and failure handling.

Instapoll

Instapoll · Review · Oct 6 C

Eight threads want one lock. Seven of them spin on test-and-set until it succeeds. The lock is correct. What are those seven doing to the machine?

- A. Little — a failed test-and-set is one instruction, and it runs on its own core
- B. Each failed attempt pulls the memory holding the lock away from the other cores
- C. Each waiter is parked by the kernel, so the cost is context switches, not memory
- D. They slow only each other; the thread that holds the lock is unaffected

Correct test-and-set can still be expensive

The waiting protocol decides how often contenders request write ownership

TAS: every spin is an atomic RMW

```
while test_and_set(&LOCK) == LOCKED {
    spin();
}

// every failed attempt writes
```

TTAS: read until it looks free

```
loop {
    while LOCK == LOCKED { spin(); }
    if test_and_set(&LOCK) == FREE {
        break;
    }
}
```

Next: cache-line ownership explains why RMW storms and false sharing bend the scaling curve.

Key Takeaways

1. A single source line is not a single machine action. x = x + 1 reads, adds, and writes, so two threads can both read 0 and both write 1, and one increment disappears. Correctness must hold for every legal interleaving, not the lucky one.

2. Mutual exclusion keeps two threads out of one critical section, and the lock must cover the whole read-modify-write, not the read or the write alone. Rust's MutexGuard makes that interval visible: it acquires on construction and releases on Drop.

3. One global lock caps speedup, because independent work still serializes through it. Finer locks can scale, but each adds safety, progress, and fairness obligations, so start coarse and let a measured speedup justify the extra complexity.