

17-350 / 17-650

Lecture 08: Encapsulating Unsafe Concurrency

Jonathan Aldrich

Sep 22, 2026

Lecture Plan

- Motivation: better load balancing in the ray tracer
- What you need for dynamic task assignment
 - `UnsafeCell` for interior mutability
 - `unsafe impl Sync` for moving chunks between threads
- Worked example: a shared-array filler, designed so its safety argument holds regardless of careless callers
- Generalizing from the example to HW3
- Benchmarking guidance for measuring speedup
- HW3 recap — due one week from today

Recap: Where a Static Division of Work Breaks Down

- `render_parallel` gives every thread a **fixed, equal** share of rows, decided before any thread starts
- But per-pixel cost in a ray tracer can be very uneven:
 - A pixel hitting a cluster of mirrors/glass recurses many bounces deep
 - A pixel that immediately misses everything into the sky finishes almost instantly
- If the expensive objects cluster in specific rows, whichever thread(s) own those rows finish last — while other threads sit **idle**, done with their cheap rows, with no way to help

Fix: Dynamic Tile Assignment

- Divide the image into **many small tiles** — more tiles than threads
 - Key safety invariant is the same as static division: the tiles must not overlap
- Each thread repeatedly **claims the next unclaimed tile** and renders it, until none remain
- Whichever thread finishes its current tile first grabs the next one — work naturally flows toward whichever threads are actually free
- This is a standard pattern: a **work queue**, consumed by a pool of workers, rather than a **fixed partition** decided up front
- Imposes some synchronization cost, because decisions are made dynamically

Options for Dynamic Tile Assignment

1. A `Mutex` around the whole output buffer. Always safe, proven using Rust's existing type abstractions. But every tile write now serializes on the lock, causing significant contention.
2. Synchronization in a scheduler giving out a safe abstraction
 - 2a. Use an existing safe abstraction such as `chunks_mut`
 - 2b. Design a new safe abstraction, implemented with a small `unsafe` core

Today: how to do (2b) — required for Homework 3

UnsafeCell<T>: Storing a *mut T inside an &T

- Rust's hard rule: &T never coexists with a live &mut T to the same memory — the compiler assumes this and optimizes accordingly
- Normally, an &T can't have mutable state inside it
- UnsafeCell<T> is the **only legal way** in Rust to get a *mut T out of a &T you were handed — and so the only way to mutate data through a shared reference
 - (Later: a design that never needs this, because it takes its pointer *before* any sharing begins)

```
use std::cell::UnsafeCell;

struct RawCell<T> { value: UnsafeCell<T> }

impl<T> RawCell<T> {
    // Safe: handing back a raw pointer cannot cause UB. Dereferencing it
    // can -- and that takes an `unsafe` block the *caller* writes.
    fn get_mut_ptr(&self) -> *mut T {
        self.value.get()      // &RawCell<T> -> *mut T, from nothing but the &.
    }

    // Must be `unsafe fn`: hands safe code a `&mut T` it can alias with a
    // second `&mut T` -- UB, with no `unsafe` of the caller's own.
    unsafe fn get_mut(&self) -> &mut T {
        unsafe { &mut *self.value.get() }
    }
}
```

All Interior Mutability Abstractions Are Built on `UnsafeCell`

Type	Discipline it adds on top of <code>UnsafeCell</code>
<code>Cell<T></code>	never hands out a mutable reference — only <code>replace</code> and a <code>get</code> that copies
<code>RefCell<T></code>	a runtime borrow counter (panics on conflicting borrows)
<code>Mutex<T></code>	an OS-level lock (blocks until available)

- Same foundation every time; the *discipline* enforcing safety is what differs — it's what the safety argument will describe

`unsafe impl Sync`: A New Kind of Unsafe

- HW1/HW2's `unsafe` was about raw pointers and jumps: dereferencing an address, jumping to a handler, trusting `mem::transmute`
- This is different in *kind*:

```
unsafe impl Sync for SomeType {}
```

- This is an **unsafe trait implementation** — a promise about a *property of the type* (`&SomeType` is safe to share across threads) that the compiler cannot verify on its own, because `Sync` isn't auto-derivable once a type contains a raw pointer or an `UnsafeCell`

Applying the Safety Framework to Shared Tiles

The same four-step framework from Lecture 3 — aimed at a new hazard:

- **Hazard:** two threads writing overlapping bytes concurrently (a data race), or a write landing outside the buffer
- **Invariant:** every tile lies inside the buffer, distinct tiles cover non-overlapping regions, and no tile is used by more than one thread — therefore every write is in-bounds and never overlaps a region any other *concurrently executing* write targets
- **Establish:** the scheduler builds the tile grid as a **partition of the image** — every tile in-bounds, no two overlapping — and an **atomic counter** hands out each tile exactly once
- **Maintain:** a tile is **unforgeable** (only the scheduler can mint one) and **linear** (owned, not `Copy` / `Clone`, consumed by the write), so no code can fabricate one or use one twice
- Note: the argument above must hold even if the code driving the threads is buggy — not just for the one call pattern we intend. Let's see how to do this.

Worked Example

Problem: `N` worker threads fill a shared `Vec<f64>` of results, one element each, by repeatedly claiming "the next index" from a shared counter and writing there.

- No ranges, no 2D geometry — just one index at a time
- Small enough to fit in a lecture & demo
- We'll build it in two passes: a first version that *looks* airtight, then a small but essential fix once we see exactly what it's missing

Step 1: The Shared Buffer

```
struct SharedResults {  
    ptr: *mut f64,  
    len: usize,  
    // Never touched again after `new` extracts `ptr` from it. Kept only  
    // so the allocation stays alive until `into_inner` reclaims it.  
    storage: Vec<f64>,  
}
```

- `ptr` is computed **once**, in the constructor, while `storage` is still **uniquely owned** — an entirely ordinary, single-threaded `Vec::as_mut_ptr()` call, no `unsafe` needed for that part at all

Step 2: `new` and `unsafe impl Sync`

```
impl SharedResults {  
    fn new(len: usize) -> Self {  
        let mut storage = vec![0.0; len];  
        let ptr = storage.as_mut_ptr();  
        SharedResults { ptr, len, storage }  
    }  
}  
  
// SAFETY: see the write-up on the next slide.  
unsafe impl Sync for SharedResults {}
```

- A raw pointer field (`*mut f64`) makes `SharedResults` **not** automatically `Sync` — raw pointers carry no compiler-checked aliasing discipline, so the compiler refuses to assume sharing one is safe
- `unsafe impl Sync` is *us* asserting: "I've checked; sharing `&SharedResults` is fine, for reasons the compiler can't see"

Step 3 (First Pass): The "Safe" Public Method

```
impl SharedResults {  
    /// Writes `value` at `index`. Safe to call concurrently from many  
    /// threads, as long as no two calls ever share the same `index`.  
    fn write(&self, index: usize, value: f64) {  
        assert!(index < self.len, "index out of bounds");  
        // SAFETY: `ptr` is valid for `len` elements. The caller  
        // guarantees this call's `index` is never used by any other  
        // concurrently executing call, so this write can't race.  
        unsafe {  
            self.ptr.add(index).write(value);  
        }  
    }  
}
```

- `write` takes `&self` — a *shared* reference — yet mutates. That's the entire point of the exercise.
- Why the scare quotes around "Safe"? The `// SAFETY` comment's requirement ("the caller guarantees...") is not enforced in this design (we'll return to this)

Step 4 (First Pass): The Scheduler

```
use std::sync::atomic::{AtomicUsize, Ordering};

struct SharedCounter { next: AtomicUsize, len: usize }

impl SharedCounter {
    fn new(len: usize) -> Self {
        SharedCounter { next: AtomicUsize::new(0), len }
    }

    /// Claims the next index, or `None` once every index is claimed.
    fn claim(&self) -> Option<usize> {
        let index = self.next.fetch_add(1, Ordering::Relaxed);
        (index < self.len).then_some(index)
    }
}
```

- `fetch_add` is an **indivisible** read-modify-write: two concurrent callers are *guaranteed* to receive two *different* values, never the same one twice

Step 5 (First Pass): Putting It Together

```
let results = SharedResults::new(20);
let counter = SharedCounter::new(20);

std::thread::scope(|s| {
    for _ in 0..4 {
        s.spawn(|| {
            while let Some(index) = counter.claim() {
                results.write(index, (index * index) as f64);
            }
        });
    }
});

let output = results.into_inner(); // Vec<f64>, now safe to use normally
```

- Four threads, twenty indices, one atomic operation per claim — every index written exactly once, by exactly one thread. **This code is *correct* — but the API it leans on is not yet *robust*.**

Discussion: Can You Break This API?

```
// You are a different module. You hold these two values, and nothing else:  
let results = SharedResults::new(20); // fn write(&self, index: usize, value: f64)  
let counter = SharedCounter::new(20); // fn claim(&self) -> Option<usize>
```

- Write code that causes a **data race** — using no `unsafe` anywhere
- How many lines did it take? What did you have to know about the design to write it?
- Design question: what is the smallest change to `write`'s signature that would turn your attack into a *compile error*?

Sample Answer: Yes — `write` Takes a Bare `usize`

```
// Two lines. Different module, no `unsafe`, compiles clean:  
s.spawn(|| results.write(0, 1.0));  
s.spawn(|| results.write(0, 2.0)); // <- race on index 0. `write` can't see it.
```

- `write`'s safety comment says "the caller guarantees `index` is never reused concurrently" — but **any** code holding `&results` can call `write` with **any** `usize`, as many times as it likes, from as many threads as it likes
- The *intended* discipline — "always go through `counter.claim()`" — is a comment and a convention. Nothing in the type signature enforces it
- `write` should be declared `unsafe`, like any function whose safety relies on unchecked assumptions about how it is used. Calling it can cause UB with no `unsafe` written by the caller. As a safe `fn`, this type is **unsound**, not just risky
- `unsafe fn write` *would* be honest and sound — but it moves the obligation to every call site, so the driver loop joins the trusted code. The better fix: make the **signature** carry the promise
- HW3 requires this kind of design: sound *no matter what the caller does*, not just safe *if the caller behaves*

The Fix: An Unforgeable, Linear `Claim`

```
/// A capability to write exactly one slot, obtained only from
/// `SharedCounter::claim`.
struct Claim { index: usize }

impl Claim {
    fn index(&self) -> usize { self.index }
}
```

- **Unforgeable:** `index` is a private field, and no function outside this module can ever construct a `Claim` — the only place in the program a `Claim { .. }` literal appears is inside `claim`, next slide. Code elsewhere can *read* `claim.index()`, never invent a `Claim`.
- **Linear:** no `#[derive(Copy, Clone)]`. A `Claim` can be *moved* (into a thread's closure) but never *duplicated*.

Step 3 (Hardened): `write` Consumes a `Claim`

```
impl SharedResults {  
    /// Writes `value` at the slot `claim` was issued for. Consumes  
    /// `claim` – it cannot be presented to `write` again, by this call  
    /// or any other, because nothing else can obtain it.  
    fn write(&self, claim: Claim, value: f64) {  
        assert!(claim.index() < self.len, "index out of bounds");  
        unsafe {  
            self.ptr.add(claim.index()).write(value);  
        }  
    }  
}
```

- `claim: Claim`, not `claim: &Claim` — taking it by value is what makes it get consumed. After this call, the `claim` that was passed in no longer exists anywhere; the borrow checker will not let you write it twice, or hand it to two threads, even if you try

Step 4 (Hardened): `claim` Mints a `Claim`

```
impl SharedCounter {  
    /// Claims the next slot as a `Claim`, or `None` once every slot is  
    /// claimed.  
    fn claim(&self) -> Option<Claim> {  
        let index = self.next.fetch_add(1, Ordering::Relaxed);  
        (index < self.len).then_some(Claim { index })  
    }  
}
```

- The `Claim { index }` literal is only legal inside this module, because `claim`'s field is private to it — this one line is the *only* place in the program a `Claim` ever comes into existence

Step 5 (Hardened): Putting It Together

```
std::thread::scope(|s| {  
    for _ in 0..4 {  
        s.spawn(|| {  
            while let Some(claim) = counter.claim() {  
                let value = (claim.index() * claim.index()) as f64;  
                results.write(claim, value);    // `claim` moved, consumed  
            }  
        });  
    }  
});
```

- Identical shape to the first pass — read the index, compute a value, write it — except the *type* of what's passed to `write` now makes the bug on the "Gap" slide a **compile error**, not just against the rules

The Safety Argument, First Attempt

- **Hazard:** two threads write the same index (a data race on that element), or an out-of-bounds index is used
- **Invariant:** every write targets an index in `0..len`, and each such index is written by **at most one** thread
- **Establish:** the counter starts at `0`; `claim` stops minting `Claim`s once `next >= len`
- **Maintain:** `fetch_add` never returns the same value twice, so no two `Claim`s ever carry the same index — and because `claim` is unforgeable and linear, **no caller, however written**, can present the same index to `write` twice or invent an index `claim` never issued

Verified under Miri (both its default and Tree Borrows modes) — no undefined behavior detected.

Post-Lecture: One Buffer, Two Counters

```
let results = SharedResults::new(20);
let (c1, c2) = (SharedCounter::new(20), SharedCounter::new(20));
std::thread::scope(|s| {
    s.spawn(|| results.write(c1.claim().unwrap(), 1.0)); // Claim { index: 0 }
    s.spawn(|| results.write(c2.claim().unwrap(), 2.0)); // Claim { index: 0 }: race
});
```

- No `unsafe`, compiles clean — and Miri reports `Data race detected`
- The broken line is **Maintain**: "`fetch_add` never returns the same value twice" is true *per counter*, and `SharedCounter::new` lets any caller make as many counters as it likes
- The hidden promise about the caller came back in a new form: not "get indices only from `claim`," but "make only one counter per buffer"
- Moving the counter *into* `SharedResults` is not enough by itself: with two buffers, a `claim` minted by A could still be handed to B's `write`

Added after lecture, prompted by a question in class.

The Fix: Each `Claim` Knows Its Buffer

```
struct SharedResults { ptr: *mut f64, len: usize, next: AtomicUsize, storage: Vec<f64> }
struct Claim<'a> { results: &'a SharedResults, index: usize }

impl SharedResults {
    fn claim(&self) -> Option<Claim<'_>> {
        let index = self.next.fetch_add(1, Ordering::Relaxed);
        (index < self.len).then_some(Claim { results: self, index })
    }
}

impl Claim<'_> {
    fn write(self, value: f64) { // consumes the claim, as before
        assert!(self.index < self.results.len, "index out of bounds");
        unsafe { self.results.ptr.add(self.index).write(value); }
    }
}
```

- One counter per buffer: `next` is a field that `new` sets to `0`;
`SharedCounter` is gone
- `write` moved onto `Claim`: a claim writes only to the buffer that minted it, so neither attack can even be *expressed*. The driver loop becomes `while let Some(claim) = results.claim() { claim.write(v) }`

The Safety Argument, Revised

- **Hazard:** unchanged — two threads write the same element (a data race), or an out-of-bounds index is used
- **Invariant:** every `claim` ever minted by `r` has an index `< min(r.next, r.len)`, and no two share an index (assuming `next` never overflows). Each `claim` writes once, so every write is in bounds and no index is written by two threads
- **Establish:** `new` creates `r` with its **only** counter, `next = 0`, and no claims
- **Maintain:** `claim`'s atomic `fetch_add` returns the old `next` and increments it. Every earlier claim's index is below the old `next`, so the new index is fresh, and it is minted only if it is `< len`. Nothing else can create a `claim` (unforgeable) or change `next` (private). A `claim` writes only to the buffer it borrows, and only once (linear). The borrow also stops `into_inner` from running while any `claim` is alive (E0505)
- **Lesson:** a token is only as unique as whatever mints it. Check that the minter can't be duplicated, too

Verified under Miri (default and Tree Borrows modes): no undefined behavior in the fix; a data race reported for the two-counter attack on the previous design.

Writing H-I-E-M: Choosing the Invariant

A good Invariant is:

- **(A) Inductive:** if it holds before any operation the safe API allows, it still holds after
- **(B) Sufficient:** whenever it holds, the Hazard cannot occur

The safety condition itself ("no index written twice") is usually (B) but **not** (A): it says nothing about what `claim` will hand out *next*

- **Generalize** it until it is inductive, often by relating it to the module's state: here, "every claim's index is `< next`"

Establish and **Maintain** are then the proof that it is inductive:

- **Establish** (base case): it holds when the object is created
- **Maintain** (inductive case): *every* operation the safe API allows, called by *any* caller, preserves it — so list them all

Today's gaps, seen this way: a caller promise inside Maintain (first pass); an operation Maintain never considered (a second `SharedCounter::new`); and a case that only shows up once the Invariant mentions `next` (overflow). A small tweak to the code to test for `next` overflow could address the remaining assumption.

This Argument is Robust to a Careless Caller

- The first-pass argument's `Maintain` step had a silent hidden clause: "...for as long as every caller gets its index only from `claim`." That's a **promise about the caller**, not a fact the module itself guarantees
- `Claim` removes the clause entirely: **unforgeable** rules out fabricating an index out of thin air; **linear** (consumed by `write`) rules out reusing or duplicating one; **tied to its buffer** rules out a second counter or a claim from another buffer — all compiler-enforced, not conventions
- So the invariant holds for *any* code holding `&results` — correct, careless, or actively adversarial — not just the one driving loop we happened to write
- **This is the requirement in Phase 3 of hw3.md:**
your safety argument must hold even if the code spawning threads and calling your API (`render_dynamic`) is buggy

The Trusted Code Is Deliberately Tiny

- Everything the safety argument depends on lives in two small items: `SharedResults` (with its counter) and `claim` — call this the module's **trusted computing base**
- The driving loop (the `thread::scope` block, standing in for HW3's `render_dynamic`) is *not* part of that base — it's an ordinary, entirely safe caller that only ever touches the trusted pair through `new`, `claim`, `write`, `into_inner`
- Consequence: that loop can spawn the wrong number of threads, claim a slot and never write it, loop in a strange order — every one of those is either a compile error, a wrong/incomplete result, or a bounds-check panic. **Never** a data race.
- This is why the assignment insists the safety argument name *one small module* and show it holds even if the surrounding caller has bugs — a small trusted base is one you can actually finish checking

Where the Boundary Actually Is

- Concretely, this means one Rust module: `SharedResults` and `Claim` live together, and **nothing else about them is `pub`** — no free-standing way to build a `Claim`, no field access, nothing
- The driving loop lives in a **different** module, and reaches the trusted pair only through their safe, `pub(crate)` signatures — same relationship HW3's sample solution actually uses: the unsafe types (`FrameBuffer`, `TileScheduler`, a `Tile` capability) live in one file, and `render_dynamic` — the thread-spawning driver — lives in a *different* file, calling in only through that safe API
- Your Phase 3 safety argument should say, explicitly, which module this is for your code, and that `render_dynamic` is not in it

Why Not Just `UnsafeCell<Vec<f64>>`?

A tempting alternative: skip the raw-pointer field, wrap the `Vec` directly, and re-derive the pointer inside `write`:

```
struct SharedResults { data: UnsafeCell<Vec<f64>>, next: AtomicUsize }

fn write(self, value: f64) {    // on Claim<'_>, as before
    unsafe {
        let ptr = (*self.results.data.get()).as_mut_ptr();    // <- forms &mut here
        ptr.add(self.index).write(value);
    }
}
```

- This momentarily forms a `&mut Vec<f64>` on **every call** — doing that from multiple threads at once (even to touch disjoint elements) is a genuinely debated pattern under Rust's stricter aliasing rules (Miri's *stacked borrows* rejects this, while *tree borrows* allows it)
- Extracting the pointer **once**, before any sharing begins, sidesteps the question entirely — prefer the strictly simpler, unambiguous option

Generalizing to Homework 3

- The shape of the solution is similar — distribute non-overlapping, unforgeable, linear capabilities, with a small `unsafe` isolated in one module
- The toy's `claim` carries **one array index**; HW3 needs a **2D image** — your capability type has to carry a **rectangle** (a tile), and your safety argument has to show the index-to-rectangle mapping is a bijection (every tile claimed, none twice, none out of bounds)
- You must design that capability type yourself — private fields, no `Copy / Clone`, minted only by your scheduler — the *pattern* is given, the *shape* (what fields, what accessors) is your Phase 3 design work
- The toy writes **one** `f64`; a tile write is **many bytes across several rows** — the offset arithmetic is yours to get right
- The toy's threads invent their own values; HW3's threads call `render_pixel` many times per tile before writing it

Benchmarking Guidance: Measure Honestly

- Time `render_sequential` and your parallel/dynamic renderers on the **same fixed scene, same machine**, ideally with nothing else running
- Check `std::thread::available_parallelism()` — and if your numbers look smaller than expected, check your actual core topology
- On Linux: `lscpu` — look at **Core(s) per socket** and **Thread(s) per core**. A machine can report more *logical* CPUs than *physical* cores (SMT/hyperthreading), which caps how much a compute-bound workload like this can speed up, no matter how good your parallel code is

A Real Example: What Happened Building the Sample Solution

- The reference solution was benchmarked on a cloud sandbox reporting 2 CPUs — but `lscpu` showed `Core(s) per socket: 1`, `Thread(s) per core: 2`: **one physical core**, two SMT siblings
- Confirmed with a raw microbenchmark (a tight integer loop, nothing to do with ray tracing): **2 threads took *longer* than 1** (38ms vs. 32ms for the same total work) — the classic signature of SMT contention on an ALU-bound workload
- Lesson: report what you measure, **and say what hardware you measured it on** — a small speedup isn't necessarily a bug in your code

The Real Numbers (Sample Solution, 1 Physical Core / 2 SMT Threads)

Phase 2 — `render_parallel` vs. `render_sequential`, benchmark scene:

Threads	Time	Speedup
1	20.92s	1.00x
2	17.60s	1.19x

Phase 3 — `render_dynamic` vs. `render_parallel`, uneven-cost scene:

Threads	static	dynamic	Improvement
2	4.94s	4.48s	9.2% faster
8 (oversubscribed)	5.12s	4.35s	15.1% faster

Why the Phase 3 Comparison Is Still Fair

- Both `render_parallel` and `render_dynamic` are **equally constrained** by this hardware's SMT ceiling — so the *relative* comparison between them is meaningful even though the *absolute* speedup is modest
- Dynamic scheduling wins more as thread count grows past the core count — more, smaller work items give the scheduler finer-grained opportunities to keep both SMT threads equally busy
- On genuine multi-core hardware: expect Phase 2's speedup to approach the core count, and Phase 3's improvement to be **more dramatic**, since idle *whole cores* (not just an underused SMT sibling) are a bigger loss than what this sandbox could even expose

HW3 Recap: Submission Checklist

Everything needed for all three phases has now been covered:

- **Phase 1:** `Triangle / Dielectric` passing the provided unit tests
- **Phase 2:** `render_parallel` exact-match at every thread count, plus a reported speedup with methodology
- **Phase 3:** `render_dynamic` exact-match, plus a reported improvement over Phase 2 on the uneven-cost scene, **plus** the four-heading safety argument in `README.md`
- Due **Sep 29** — one week from today

Wrap-up

- You now have every tool this assignment needs: traits, safe concurrency, and one carefully-argued `unsafe` module
- Questions? Piazza is the place for anything Phase 3-specific this week
- Next up on the syllabus: Rust's asynchrony and distributed systems