

17-350 / 17-650

Lecture 07: Safe Concurrency in Rust

Jonathan Aldrich

Sep 17, 2026

Lecture Plan

- What is a data race? How does Rust rule them out?
- Spawning threads, closures, and ownership transfer
- Passing data among threads with `Send` and `Sync`
- `std::thread::scope` : borrowing local data across threads
- Proving disjointness statically: `split_at_mut` / `chunks_mut`
- `Arc` , `Mutex` , and channels
- Demo, and testing concurrent code via determinism

Next lecture: encapsulated unsafe concurrency

Rust and Concurrrency

- Recall Rust's approach from Lecture 1
 - Type-driven concurrency - prevents data races by construction
- Ray tracing is **embarrassingly parallel**: every pixel's color depends on nothing but the scene and that pixel's own coordinates
 - No pixel needs any other pixel's result
 - No shared mutable state *has* to exist between them at all
- Today's focus: safe parallelism in Rust
 - Relevant to Phase 2 of Homework 3

What is a Data Race?

- Two threads access the same memory without synchronization, and at least one of the accesses is a write
- In Rust (as in C/C++), this is **undefined behavior**
 - Worse than just getting an incorrect answer
 - The compiler is permitted to assume data races don't happen, so if one does, it can result in arbitrary behavior, including corrupted memory, memory leaks, or crashes

Rust vs Other Languages

- Java and Go are type safe and provide synchronization constructs, but few guarantees
 - Unsynchronized access is caught only at run time, if at all
 - e.g. with a `ConcurrentModificationException` or an external race detector
- Rust's philosophy: "make illegal states unrepresentable"
 - Code with data races should not compile
- Mechanisms
 - The same borrow-checking machinery you already know
 - Two marker traits (`Send` , `Sync`) that extend reasoning across thread boundaries

std::thread::spawn

```
use std::thread;

let handle = thread::spawn(|| {
    println!("hello from a new thread");
    42
});

let result = handle.join().unwrap();
assert_eq!(result, 42);
```

- `thread::spawn` takes a closure and runs it on a new OS thread
- Returns a `JoinHandle<T>`, where `T` is the closure's return type
- `.join()` blocks until the thread finishes, returning its result (or an `Err` if the thread panicked)

`move` Closures: Ownership Transfer

```
let data = vec![1, 2, 3];

let handle = thread::spawn(move || {
    println!("{data:?}");    // the thread now OWNS `data`
});

// data is no longer usable here -- it moved into the closure
handle.join().unwrap();
```

- `move` forces the closure to capture each variable by value (moving it, or copying it if `Copy`) instead of letting the compiler create an implicit borrow
- After the `move`, the parent function can't use `data` anymore — it belongs to the spawned thread now

Why Not Just Borrow?

```
let data = vec![1, 2, 3];

let handle = thread::spawn(|| {    // no `move` -- tries to borrow `data`
    println!("{data:?}");
});
// error[E0373]: closure may outlive the current function,
// but it borrows `data`, which is owned by the current function
```

- Nothing guarantees the spawned thread finishes before this function returns and `data` is dropped
- If the borrow were allowed, the spawned thread could end up reading `data` *after* it's gone — a dangling reference, which is what borrowing rules are designed to prevent
- `thread::spawn` thus requires that the closure's data can live for the lifetime of the program
 - a `'static` lifetime

Send and Sync: Two New Marker Traits

- `T: Send` — safe to *move* a value of type `T` to another thread
- `T: Sync` — safe to *share* `&T` between threads (equivalently, `&T: Send`)
- Both are **marker traits**: no methods, nothing to implement
 - They allow the compiler to track "is this safe to hand to another thread" in the type system
- Both are **auto-derived structurally**
 - a type is `Send` / `Sync` exactly when all of its fields are
 - the compiler works this out for you, with zero code and no runtime cost
 - you don't have to declare them, except as an `unsafe` exception to the rule above

Zero-Cost Data Race Checking - Like Many Rust Checks

- `Send` / `Sync` are checked at compile time
 - No run time enforcement, no overhead
 - By the time your program runs, the compiler has already proven no data race can occur
- Rust theme: "compile-time guarantee, pay nothing at run time"
 - See also ownership/borrowing, generics, and non-`dyn` trait dispatch

Passing Smart Pointers Among Threads

Type	Send ?	Sync ?	What and Why
<code>Rc<T></code>	no	no	Sharing with unknown lifetime via reference counting. Refcount increments/decrements aren't atomic — races corrupt the count
<code>RefCell<T></code>	yes	no	Holds a mutable object in an immutable (sharable) container. One mutable (or several immutable) borrows at a time, checked non-atomically — fine on one thread
<code>Arc<T></code>	yes	yes	Like <code>Rc<T></code> but refcount uses atomic operations
<code>Mutex<T></code>	yes	yes	Like <code>RefCell<T></code> but borrowing synchronizes, waiting for other threads' borrows to be returned

* `Rc` / `RefCell` are OK to use, just keep to a single thread; `Arc` / `Mutex` are their thread-safe counterparts, at the cost of atomic/lock overhead.

HW3's Traits - Send + Sync

```
pub trait Hittable: Send + Sync {  
    fn hit(&self, ray: &Ray, t_min: f64, t_max: f64) -> Option<HitRecord>;  
}  
  
pub trait Material: Send + Sync {  
    fn scatter(&self, ray_in: &Ray, hit: &HitRecord, rng: &mut SmallRng)  
        -> Option<(Vec3, Ray)>;  
}
```

- A scene must be safely **shareable, read-only**, across every render thread at once
- Declaring `Hittable: Send + Sync` means that every type that implements `Hittable` must also be `Send` and `Sync`
 - Thus `dyn Hittable` (and any nested `Material`) in a `Scene` is *guaranteed*, by the compiler, to be safe to share this way — you cannot accidentally put a non-thread-safe shape in the scene

`thread::spawn` - Going Beyond `'static`

- `thread::spawn` requires `'static` data because nothing ties the spawned thread's lifetime to the calling function's stack frame
- But `render_parallel` wants to borrow `scene: &Scene` and `camera: &Camera` — local to the calling function, not `'static`
- Cloning or `Arc`-wrapping the whole scene for every render call would work, but it's unnecessary overhead for something that's only ever *read*, never mutated, during the render

`std::thread::scope`: Borrowing Across Threads

```
let scene = build_scene();           // a local, not 'static'

std::thread::scope(|s| {
    for _ in 0..4 {
        s.spawn(|| {
            // borrows `scene` by reference -- no `move`, no `Arc`
            render_some_rows(&scene);
        });
    }
}); // <- every spawned thread is guaranteed joined by here
```

- The `scope` method provides a `Scope` value `s` whose lifetime is bound to the method invocation
- Everything captured by the closures passed to `s.spawn` must live at least this long
- `scope` keeps a count of live threads spawned within the scope
- After calling the provided closure, `scope` waits for all those threads to finish, ensuring no thread is still using the captured values by the time `scope` returns

This mechanism makes borrowing a local across a scoped thread sound — no dangling reference is possible, so no `'static` bound is needed

Writing to an Array or Buffer in Parallel

- In the ray tracer, each thread must each write to a **different** part of one output image buffer
- We want **no runtime synchronization** at all for this — no lock, no atomic per pixel — just each thread owning its own region outright
- Rust provides an abstraction that can prove, at compile time, that the regions don't overlap

`split_at_mut` Creates Disjointness Verifiable at Compile-Time

```
let mut data = [0u8; 8];
let (left, right) = data.split_at_mut(4);
// left:  &mut [u8] over data[0..4]
// right: &mut [u8] over data[4..8]

left[0] = 1;
right[0] = 2;    // no conflict -- the borrow checker knows these don't overlap
```

- `split_at_mut(&mut self, index) -> (&mut [T], &mut [T])` splits **one** `&mut [T]` array slice into **two** at the given index, and the borrow checker guarantees the two halves do not overlap — purely from the types, no runtime check at all
- `chunks_mut(n)` generalizes this to *many* equal-sized, non-overlapping `&mut` slices in one call

Discussion: How Could `split_at_mut` Be Implemented?

```
fn split_at_mut(&mut self, index: usize) -> (&mut [T], &mut [T]);
```

- Try sketching an implementation yourself, using only what you already know
 - Hint: you can take a sub-slice with range syntax, e.g. `&mut self[a..b]`
- Can this be done in **safe** Rust at all?
 - If not — what exactly does the borrow checker refuse to allow?
- If it requires `unsafe`, what would *you* have to guarantee by hand that the compiler can no longer check for you?

Sample Answer: Implementing `split_at_mut`

- Not possible in safe Rust. `&mut self[..index]` and `&mut self[index..]` are both mutable borrows of `self` — the borrow checker only sees "two `&mut` borrows of the same slice," it can't reason about *which indices* each one touches
- The real standard library implementation uses `unsafe`:

```
fn split_at_mut(&mut self, index: usize) -> (&mut [T], &mut [T]) {
    let len = self.len();
    let ptr: *mut T = self.as_mut_ptr();
    assert!(index <= len);
    unsafe {
        (
            std::slice::from_raw_parts_mut(ptr, index),
            std::slice::from_raw_parts_mut(ptr.add(index), len - index),
        )
    }
}
```

- There is one raw pointer and two slices built from disjoint sub-ranges of it — the programmer (not the compiler) now guarantees `index <= len` and that the ranges don't overlap
- The `unsafe` block is *encapsulated*: everything outside this function still gets a fully safe, checked API — **more on this pattern next lecture**

Arc<T>: Shared Ownership Across Threads

Not needed for Phase 2, but worth knowing:

```
use std::sync::Arc;
use std::thread;

let shared = Arc::new(vec![1, 2, 3]);

let mut handles = vec![];
for _ in 0..3 {
    let shared = Arc::clone(&shared); // bumps the atomic refcount
    handles.push(thread::spawn(move || {
        println!("{shared:?}");
    }));
}
for h in handles { h.join().unwrap(); }
```

- Same interface as `Rc<T>`: `Arc::clone` gives another owner, the value drops once the last owner does — no `.lock()`, nothing to release
- The only difference is *how* the refcount is updated: atomically, so `Arc<T>` is `Send / Sync` where `Rc<T>` is neither
- Use `Arc` exactly when you'd use `Rc` --when lifetimes are determined by the last user of an object--but need to hand the same data to more than one thread

Mutex<T>: Runtime-Checked Exclusive Access

```
use std::sync::{Arc, Mutex};
use std::thread;

let counter = Arc::new(Mutex::new(0));

let mut handles = vec![];
for _ in 0..4 {
    let counter = Arc::clone(&counter);
    handles.push(thread::spawn(move || {
        let mut guard = counter.lock().unwrap(); // blocks until available
        *guard += 1;
    })); // guard dropped -> unlocked
}
for h in handles { h.join().unwrap(); }
```

- `.lock()` plays the same role as `RefCell`'s `.borrow_mut()`: both hand back a guard that derefs to `&mut T` and enforces exclusive access
- The difference is what happens on conflict: `RefCell` panics immediately; `Mutex::lock()` instead *blocks* the calling thread until the lock is free
- `Arc<Mutex<T>>` is a standard combo — shared ownership *and* safe mutation, together

Channels: Handing Off Ownership, Not Sharing It

```
use std::sync::mpsc;
use std::thread;

let (tx, rx) = mpsc::channel();

thread::spawn(move || {
    let result = compute_something();
    tx.send(result).unwrap(); // `result` moves down the channel
});

let result = rx.recv().unwrap();
```

- No shared memory at all: `send` asynchronously moves a value into the channel, `recv` moves it back out — only one side ever owns it at a time
- `mpsc` = *multi-producer, single-consumer*; clone `tx` to give more threads a sender, `rx` stays the one receiver
- Fits work that gets handed off to completion, rather than state threads need to keep touching — no `Arc` / `Mutex` required

A Problem These Don't Solve

- What if the *assignment* of work to threads isn't decided until run time — e.g., whichever thread finishes first grabs the next chunk of work, rather than every thread getting a fixed, equal share up front?
- `split_at_mut` / `chunks_mut` need to know the split points **before** any thread starts — they can't help once "who gets what" is decided dynamically
- A `Mutex` around the whole buffer would work, but serializes every write — exactly the contention dynamic work-splitting is meant to avoid
- **More in the next lecture!**

Deterministic Ray Tracing, with Concurrency

```
fn pixel_seed(opts: &RenderOptions, x: u32, y: u32) -> u64 {  
    opts.seed.wrapping_add((y as u64) * (opts.width as u64) + (x as u64))  
}
```

- Ray tracing includes (pseudo-)randomness, e.g. sampling rays within a pixel or sampling transmit/reflect off of a dielectric (glass-like) surface
- Every pixel's random-number generator is seeded from **its own coordinates** and the run's base seed — never from a shared global RNG, never from wall-clock time
- Which thread happens to compute pixel `(x, y)` can never change what `(x, y)`'s color value is — the seed doesn't depend on thread identity or timing at all

Determinism as a Testing Strategy

- Because of the seeding scheme above: `render_parallel` / `render_dynamic` can (and must, for full credit) match `render_sequential` **exactly**, pixel for pixel, at *every* thread count — no floating-point tolerance needed anywhere in HW3
- The payoff: a mismatch is always a data race or bug, not noise. Two threads that raced on a shared byte will (eventually, if not immediately) produce a result that depends on scheduling — and scheduling is exactly what your seeding scheme was supposed to make irrelevant
- Treat a failing exact-match test as **a bug to fix**, never as a threshold to loosen

Wrap-up

- You now have everything needed for Phase 2 of Homework 3: `thread::scope` , `Send` / `Sync` , `split_at_mut` / `chunks_mut` , and a testing strategy for concurrent code
- Next lecture: the load-balancing problem that static chunking can't solve, and the one place this assignment requires `unsafe`