CS 4414/5416: Systems Programming

Atomics and Memory Models

David Bindel

2026-10-29

Logistics

Upcoming

- P4 (netsim) revision
  - Shortened assignment due *Wednesday*, 11/4
  - Final part will now be *optional*
- P5 (Tetris) will be released by 11/4
- Lab 10 is Friday: shared memory practice

PollEv Issue of the Week

- Apparently upvote/downvote $\neq$ participation on QA polls
- Will adjust Thursday attendance rubric appropriately

Recap

Where We Are

This module: Concurrent programming with threads

- Message-based concurrent programming
- Shared memory concurrent programming
- Synchronization and deadlock
- Concurrent network programming

This Week

- Tue: Intro + locks and condition variables
- Today: Atomics and memory models

Recommended: Rust Atomics and Locks, Ch 1-3

Reminder: Arc, Mutex, RwLock, Condvar

- `Arc<T>`: Atomically reference-counted allocated `T` object
- `Mutex<T>`: Lock-guarded `T` object
  - `lock` returns a `LockResult<MutexGuard<T>>`
  - Dropping the `MutexGuard` releases the lock
- `RwLock<T>`: Allows multiple readers or one writer
- `Condvar`: Used to signal a condition (e.g. queue empty)

Arc?

On `Rc` or `Arc`:

- `clone()` increments the (strong) ref count
- `drop` decrements the reference count
- On zero count, drop the owned object

Atomic updates avoid leaks or double-drops!

Arc<Mutex>

Common pattern: `Arc<Mutex<T>>`

- `Arc` to share ownership
- `Mutex` for mutual exclusion

Analogous to `Rc<RefCell<T>>` in serial code

Poll Everywhere!

When is a Rust Mutex unlocked?

(a) On a call to the `unlock` method

(b) When the `MutexGuard` is dropped

(c) On the last write reference to the wrapped entity

Atomics!

Definition

Data races (undefined behavior) happen when

- Conflicting accesses (overlapping, >= 1 writer)
- No *happens-before* relation between them
    - More about this later in lecture
- At least one access is *non-atomic*

Atomics

Atomic operations act indivisibly

- Without locks *if hardware allows*
  - Not all HW supports all ops - kernel may assist
- Still not free (e.g. cache coherence costs you)
- But maybe faster than a locking version?

A Few Choice Quotes

Rust pretty blatantly just inherits the memory model for atomics from C++20. This is not due to this model being particularly excellent or easy to understand. Indeed, this model is quite complex and known to have several flaws. Rather, it is a pragmatic concession to the fact that everyone is pretty bad at modeling atomics.

– Rustonomicon

A Few Choice Quotes

> *“It’s the difference between riding a rollercoaster and building one mid-air while it’s on fire – and you’re also the passenger.”*
> *– Julian Goldstein, yeet blog*

Also: “It’s fast, it’s elegant, and it will absolutely kill you if you do it wrong.”

A Few Choice Quotes

> *“Atomics!”*
>
> *– Gurney Halleck (played by Patrick Steward), Dune, 1984*

(The characters all put on some type of helmet, drop a nuke... and then take their helmets off to go riding through the fallout. On the backs of giant carnivorous worms. That’s safe?)

Why Do This Again?

Performance!

- If you've ruled out a message passing option...
- In limited doses, if you think it's really needed...
- And nobody else has done it for you...
- ... or if you just enjoy this type of thing!

Rust Interface

In `std::sync::atomic`:

- Primitive wrappers (`AtomicBool`, `AtomicUsize`, etc)
- Use interior mutability (access through shared references)
- API: store, load, fetch-and-modify, compare-and-exchange
- All methods take an `Ordering` argument (more on this later)
    - `Relaxed` is the least restrictive

(Examples adapted from Rust Atomics and Locks, Ch 2)

Load and Store

```
// Only if target_has_atomic_load_store=8
impl AtomicBool {
    pub fn load(&self, order: Ordering) -> bool;
    pub fn store(&self, value: bool, ordering: Ordering);
}
```

- Load and store will happen (no compiler elision)
- Good for simple cases: stop flags, status reporting

Stop Flag

```
use std::sync::atomic::{self, AtomicBool, Ordering::Relaxed};
use std::thread;

fn main() {
    let stop = AtomicBool::new(false);
    thread::scope(|s| {
        s.spawn(|| { while !stop.load(Relaxed) { bg_work(); } });
        for line in std::io::stdin().lines() {
            match line.unwrap().as_str() {
                "stop" => break,
                cmd => println!("unknown command: {cmd:?}"),
            }
        }
        stop.store(true, Relaxed);
    });
}
```

Racy Lazy Initialization

```
// What happens with two simultaneous calls?
fn get_x() -> u64 {
    static X: AtomicU64 = AtomicU64::new(0);
    let mut x = X.load(Relaxed);
    if x == 0 {
        x = calculate_x();
        X.store(x, Relaxed);
    }
    x
}
```

(Probably better: `std::sync::LazyLock`)

Fetch-and-Modify

```
impl AtomicI32 {
    pub fn fetch_add(&self, v: i32, ordering: Ordering) -> i32;
    pub fn fetch_sub(&self, v: i32, ordering: Ordering) -> i32;
    pub fn fetch_or(&self, v: i32, ordering: Ordering) -> i32;
    pub fn fetch_and(&self, v: i32, ordering: Ordering) -> i32;
    pub fn fetch_nand(&self, v: i32, ordering: Ordering) -> i32;
    pub fn fetch_xor(&self, v: i32, ordering: Ordering) -> i32;
    pub fn fetch_max(&self, v: i32, ordering: Ordering) -> i32;
    pub fn fetch_min(&self, v: i32, ordering: Ordering) -> i32;
    pub fn swap(&self, v: i32, ordering: Ordering) -> i32; // "fetch_store
}
```

Assigning IDs

```
fn allocate_new_id() -> u32 {
    static NEXT_ID: AtomicU32 = AtomicU32::new(0);
    NEXT_ID.fetch_add(1, Relaxed)
}
```

What about overflow?

Assigning IDs

```
fn allocate_new_id() -> u32 {
    static NEXT_ID: AtomicU32 = AtomicU32::new(0);
    let id = NEXT_ID.fetch_add(1, Relaxed);
    assert!(id < MAX_IDS, "Too many IDs!");
    id
}
```

What could go wrong here?

Assigning IDs

```
fn allocate_new_id() -> u32 {
    static NEXT_ID: AtomicU32 = AtomicU32::new(0);
    let id = NEXT_ID.fetch_add(1, Relaxed);
    if id >= MAX_IDS {
        NEXT_ID.fetch_sub(1, Relaxed);
        panic!("Too many IDs!");
    }
    id
}
```

Probably mostly OK.

Compare-Exchange

Much more flexible:

```
impl AtomicI32 {
    pub fn compare_exchange(
        &self,
        expected: i32,
        new: i32,
        success_order: Ordering,
        failure_order: Ordering
    ) -> Result<i32, i32>;
}
```

If `value` is `expected`, change to `new` and return `Ok(expected)`; otherwise return `Err(value)`.

Assigning IDs

```
fn allocate_new_id() -> u32 {
    let mut current = a.load(Relaxed);
    loop {
        let new = current + 1;
        match a.compare_exchange(current, new, Relaxed, Relaxed) {
            Ok(_) => return current,
            Err(v) => current = v,
        }
    }
}
```

ABA Problem

Possible issue in some cases:

- `load` a value A
- Another thread changes to B and back to A
- `compare_exchange` succeeds even with change

Ask Me a Question!

Memory Models

Sequential Consistency

A “natural” mental model: Execution is consistent with a sequential interleaving of thread instructions.

- But Rust statements $\neq$ CPU instructions
- And compiler can usually re-order if no dependencies
- And what about caches?

Sequential consistency is expensive! Cheaper options?

Memory Models

- Memory model gives rules to *reason* about ordering
- Abstracts the hardware and compiler
  - Which may have different rules!
  - E.g., x86 and ARM have different HW models
- Rust follows C++20 for now (may change to C++26?)

Happens Before

Agreed-upon orderings described by *happens-before*:

- In thread, `f(); g();` gives `f()` happens-before `g()`
- Transitive: A happens-before B + B happens-before C means A happens-before C
- Cross-thread
  - On *memory fences*
  - On spawn/join, locking
  - Otherwise, depends on model

Relaxed Memory Model

Assume:

- A thread processes its own reads/writes in order
- All agree on a *total modification order* for atomics

But no other happens-before relations

- Matching load-store does *not* give happens-before

What is the Output?

```
static X: AtomicI32 = AtomicI32::new(0);
static Y: AtomicI32 = AtomicI32::new(0);

fn a() {
    X.store(10, Relaxed); // 1
    Y.store(20, Relaxed); // 2
}

fn b() {
    let y = Y.load(Relaxed); // 3
    let x = X.load(Relaxed); // 4
    println!("{x} {y}");
}
```

Poll Everywhere!

Which output is impossible?

(a) 10 20

(b) 0 0

(c) 0 20

(d) 10 0

(e) None of the above

Acquire-Release

Add happens-before between matched loads/stores

- Store involves `Release`
- Load involves `Acquire`
- Load/store ops involve `AcqRel`

Spinlock Example (Rust docs)

```rust
fn main() {
    let spinlock = Arc::new(AtomicUsize::new(1));
    let spinlock_clone = Arc::clone(&spinlock);
    let thread = thread::spawn(move || {
        spinlock_clone.store(0, Ordering::Release);
    });

    // Wait for other thread to release lock
    while spinlock.load(Ordering::Acquire) != 0 {
        hint::spin_loop();
    }
    thread.join().unwrap();
}
```

Fences

- Can be `Acquire`, `Release`, `AcqRel`, or `SeqCst`
- Not tied to a specific variable

Ask Me a Question!

Wrap-Up

Summary

Shared memory requires synchronization!

- Atomics are a “cheap” low-level option
- But ordering *really complicated*
  - Involves Rust memory models
  - … which are C++ models, which are still evolving

Up Next

- Fri: Actually practicing shared memory code
- Next week: Deadlock and live-lock; multithreaded network programming

Outro

On Ed: What do *you* think would be good final questions for this class?

Also: Happy Halloween weekend!