CS 4414/5416: Systems Programming

Intro to Shared State Concurrency

David Bindel

2026-10-29

Logistics

Upcoming

- P4 (netsim) revision
  - Shortened assignment due Monday, 11/2
- Lab 10 is Friday: shared memory practice

Final Exam

- Final is Dec 16 at 9 AM
  - Shooting for alt 6-8 on Dec 7
- Proposal: Exams (60%) as $\max(P + F, 1.5F)$
  - P as prelim score (40%)
  - F as final score (20%)

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

- Today
  - Races, ordering, deadlock, contention
  - Arc, Mutex, and RwLock
  - Condition variables and parking
- Thu: Atomics and memory models

Recommended: Rust Atomics and Locks, Ch 1-3

This Week

Shared memory programming is

(a) Something you've seen before (2110, 3410)
(b) Deceptively easy to get wrong

There is a reason we repeat this stuff!

Poll Everywhere!

A quarter of the time in some code is spent on serial work, and the remainder can be parallelized on a p-core machine. Which of these is *not* true of the total runtime?

(a) It might get slower with more cores
(b) Amdahl's law says speedup is less than 4
(c) It will be p times faster than the serial case
(d) None of the above

Shared State Blues

Channels to Shared Data

- Rust Book summary:
  - Channels favor single ownership
  - Shared-memory favors multiple ownership
- Why people think they like shared state
  - Sharing more intuitive from serial case?
  - “Shared nothing” makes consistency harder?
- But sharing nicely is hard…

Write About Now

- Easy: Shared read-only data
- Easy: Non-shared read-write data
- Harder: One writer, many readers
- Hard: Multiple writers

Counter Example

Consider a language like C:

- Consider a shared counter c
- Each thread increments c on events
- What could go wrong?

Data Races

Thread 1	Thread 2
`t1 = load(c)`	
	`t2 = load(c)`
`t1 += 1`	
	`t2 += 1`
store `t1`	
	store `t2`

One update gets lost!

Data Race

- Conflicting memory operations
  - Two threads access same data
  - At least one is a write
- Conflict is a *data race* if no ordering
  - We'll talk about "happens-before" on Thu

Caches

Thread 1	Thread 2
`t1 = load(c)`	
`t1 += 1`	
store `t1`	
	`t2 = load(c)` *cached*
	`t2 += 1`
	store `t2`

Might still lose update!

Caches and Ordering

Sequential consistency Execution is consistent with a sequential interleaving of thread instructions.

- Cache breaks sequential consistency!
- Thu: sequential consistency and memory models

Synchronization

Synchronization is coordination needed to

- Ensure consistency in data access
- Ensure consistent interpretation of time

Really about preserving understood *invariants*.

But synchronization has performance implications...

Coherency and contention

- Caching makes sharing harder!
- So... keep caches coherent?
  - Yes, can do this in various ways
  - But extra traffic slows things down
  - Compulsary, capacity, conflict, *coherency* misses!
  - Talk about *contention* for shared lines

Contention continued

```
let mut counters = Counter(0,0); // Two-field struct
thread::scope(|s| {
    s.spawn(|| update_counter(&mut counters.0));
    s.spawn(|| update_counter(&mut counters.1));
});
let total = counters.0 + counters.1;
```

No logical sharing – but maybe shared cache line!

Deadlock

Thread 1	Thread 2
`lock(x)`	
	`lock(y)`
`lock(y)`	
	`lock(x)`
Uh-oh	Uh-oh

- Deadlock! T1 waits for `y` while T2 waits for `x`
- Not unique to shared memory; more in one week

A Rusty Note

- Rust disallows simultaneous writers
  - Usually statically enforced, except RefCell
  - Our example data race is undefined behavior
  - Lack of synchronization here means compile error
- *But* non-deterministic behavior is allowed!
- Too little synchronization may give *unintended* behavior
  - Deadlock, violation of data structure invariants, etc

Ask Me a Question!

Standard Synchronization

Basic constructs

- Atomics
- Locks / mutexes
- Semaphores / condition variables
- Critical sections and monitors
- Barriers

Atomics

- Protected *indivisible* updates on one variable
  - Compare-and-swap
  - Maybe increment/decrement, other simple ops
- Usually hardware supported (no syscalls)

More on this on Thu

Locks / Mutexes

- Piece of data with `lock` and `unlock` operations
- Only one thread at a timeholds a lock
- Lock = mutex = **Mut**ual **Ex**clusion variables
- Forces threads to wait to get lock

Condition Variables

A common pattern:

- Obtain a lock
- Realize there is nothing to do
- Release lock and sleep until work maybe ready

Use *condition* variable to signal work ready

- Ex: signal when work is available in a queue

Critical Sections and Monitors

- *Critical section*: only one at a time allowed
  - Code requiring exclusive access to resources
  - Can implement with statically scoped locking
  - Bakery approach: “Wait until your number is called”
- *Monitor*: higher-level org around data + methods (a la Java)
  - Synchronized methods provide mutual exclusion
  - Condition variable(s) control queueing

Barrier

Bulk synchronous model: phases separated by *barriers*

- All threads stop at barrier, then proceed together
- Writes should only be read by other threads *after* a barrier

Ask Me a Question!

Arc, Mutex, and RwLock

Reminder: Rc and RefCell

```
struct LoggedCounter {
    log: Rc<RefCell<File>>, // Same log OK for many counters
    name: String,
    data: usize,
}

impl LoggedCounter {
    fn increment(&mut self) -> io::Result<()> {
        writeln!(self.log.borrow_mut(), "Updated {}", self.name)?;
        self.data += 1;
        Ok(())
    }
}
```

`Rc` for shared ownership, `RefCell` for interior mutability

Rc and RefCell

- `!Send`: unprotected reference count/mutability flags
- `Arc` (atomic Rc) synchronizes ref count access
- Not sensible to exactly have `ARefCell`
    - `panic!` on concurrent writes is undesireable!

Mutex

Rust `Mutex<T>` type protects a data object of type `T`

- Wrapper with interior mutability (like `RefCell`)
- `lock` method returns `LockResult<MutexGuard<'_, T>>`
- `MutexGuard` implements `Deref` and `DerefMut`
- Unlock when `MutexGuard` is dropped
- Result error type is `Poison<T>`
    - Lock is *poisoned* if a thread holding the lock panics

Mutex

Ex: From Comprehensive Rust

```
// Copyright 2024 Google LLC
// SPDX-License-Identifier: Apache-2.0
use std::sync::Mutex;

fn main() {
    let v = Mutex::new(vec![10, 20, 30]);
    println!("v: {:?}", v.lock().unwrap());
    {
        let mut guard = v.lock().unwrap();
        guard.push(40);
    }
    println!("v: {:?}", v.lock().unwrap());
}
```

Arc and Mutex

Common to use `Arc<Mutex<T>>` for shared data

- `Arc` owns data, satisfies `'static` (unlike raw ref)
- Atomic updates make reference counting thread safe
- Wrapped mutex guards data inside

Similar to `Rc<RefCell<T>>` pattern in single-thread case

RwLock

- `Mutex<T>`: Only one thread can hold lock
- Maybe overkill if most accesses are reads!
- `RwLock<T>` replaces `lock()` with `read()/write()`
- At any time, allow one writer xor multiple readers
    - Acts like a multi-threaded `RefCell`

Condition Variables and Parking

Condition Variable

Methods on a condition variable (`Condvar`):

- `wait(m)`: Unlock mutex `m` and go to sleep
  - Wait for a wake signal before return
  - Mutex is re-locked on return
- `wait_timeout(m,t)`: Wait with a timeout `t`
- `notify_one()`: Unlock + wake one waiting thread
- `notify_all()`: Unlock + wake all waiting threads

Producer-Consumer Example

```
use std::collections::VecDeque;
use std::sync::Condvar;

fn handle_queued_work(qmutex: &Mutex<VecDeque>, not_empty: &Condvar) {
    loop {
        {
            let mut q = queue.lock().unwrap();
            let item = loop {
                if let Some(item) = q.pop_front() {
                    break item;
                } else {
                    q = not_empty.wait(q).unwrap();
                }
            };
        }
        handle(item);
    }
}
```

Producer-Consumer Example

```
fn main() {
    let queue = Mutex::new(VecDeque::new());
    let not_empty = Condvar::new();
    thread::scope(|s| {
        // Consuming thread
        let t = s.spawn(|| handle_queued_work(&queue));

        // Producing thread
        for i in 0.. {
            queue.lock().unwrap().push_back(i);
            not_empty.notify_one();
            thread::sleep(Duration::from_secs(1));
        }
    });
}
```

Thread Parking

Usually use `Condvar`, but can also call:

- `thread::park()` - Put thread to sleep
- `thread::park_timeout(t)` - Park with timeout `t`
- `handle.unpark()` - Wake thread back up

Poll Everywhere!

What was most confusing today?

Wrap-Up

Summary

Shared memory requires synchronization!

- `Mutex<T>` and `Condvar` in Rust
- Atomics and memory models Thu
- There are other possibilities (e.g. barriers)

Up Next

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

Outro

Reminder for US folks

- Early voting is this week
- Election day is in one week