Advanced Rust - Lab 1: Thread Safety and Synchronization

Lukáš Hozda

2025

Introduction

Let's do some uhh, thread safety paralellization stuff

Setup

1. Make sure you have Rust installed (you probably do)
2. Create a new project for this lab: `cargo new rust_concurrency_lab`
3. You'll implement separate binaries for each exercise

Exercise 1: Thread-Safe Counter Implementations (25 minutes)

Objective

Implement a thread-safe counter using three different synchronization mechanisms, then benchmark their performance.

Instructions

Create a program that spawns multiple threads, each incrementing a shared counter multiple times. Implement this using:

1. `Arc<Mutex<T>>` for shared mutable state
2. `AtomicUsize` for lock-free operations
3. Channels (`mpsc`) for message passing

Compare the performance of each approach using simple timing measurements.

Requirements

- Use 8 threads
- Each thread should increment the counter 100,000 times
- Verify that the final count is correct (8 * 100,000)
- Report timing results for each implementation

Getting Started

Here's a skeleton to begin with:

```rust
use std::sync::{Arc, Mutex};
use std::sync::atomic::{AtomicUsize, Ordering};
use std::sync::mpsc;
use std::thread;
```

```rust
use std::time::{Duration, Instant};

// Configuration parameters
const NUM_THREADS: usize = 8;
const INCREMENTS_PER_THREAD: usize = 100_000;

fn main() {
    println!("Testing different counter implementations");

    // TODO: Implement and test the three counter versions
}

fn test_mutex_counter() -> Duration {
    // TODO: Implement mutex-based counter
}

fn test_atomic_counter() -> Duration {
    // TODO: Implement atomic counter
}

fn test_channel_counter() -> Duration {
    // TODO: Implement channel-based counter
}
```

Questions to Consider

1. Which implementation is fastest? Why?
2. How does the performance gap change as you vary the number of threads?
3. What are the trade-offs between these different synchronization approaches?

Exercise 2: Thread Pool Implementation (25 minutes)

Objective

Implement a simple thread pool that can accept tasks and execute them on a fixed number of worker threads.

Instructions

Create a `ThreadPool` struct that:

1. Initializes with a specified number of worker threads
2. Accepts closures to be executed (`FnOnce() + Send + 'static`)
3. Distributes the work among the threads
4. Provides a clean shutdown mechanism

Requirements

- The pool should properly handle multiple concurrent tasks
- Tasks should be distributed across all worker threads
- The pool should shut down cleanly when dropped
- Each worker should process tasks until told to terminate

Getting Started

Here's a skeleton to begin with:

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

// TODO: Define Message enum for job management

type Job = Box<dyn FnOnce() + Send + 'static>;

struct Worker {
    id: usize,
    thread: Option<thread::JoinHandle<()>>,
}

impl Worker {
    fn new(id: usize, receiver: Arc<Mutex<mpsc::Receiver<Message>>>) -> Worker {
        // TODO: Implement worker thread logic
    }
}

pub struct ThreadPool {
    workers: Vec<Worker>,
    sender: mpsc::Sender<Message>,
}

impl ThreadPool {
    /// Create a new ThreadPool with the specified number of threads
    pub fn new(size: usize) -> ThreadPool {
        // TODO: Implement thread pool initialization
    }

    /// Submit a job to be executed by the thread pool
    pub fn execute<F>(&self, f: F)
    where
        F: FnOnce() + Send + 'static,
    {
        // TODO: Implement job submission
    }
}

impl Drop for ThreadPool {
    fn drop(&mut self) {
        // TODO: Implement clean shutdown
    }
}

fn main() {
    // TODO: Test your thread pool implementation
}
```

Questions to Consider

1. How does your design handle task distribution?
2. What happens if a worker panics?

Exercise 3: Identifying and Fixing Data Races (20 minutes)

Objective

The following code examples have issues and either do not compile, or do something suspicious (yet not memory unsafe) at runtime

Instructions

For each code sample below:

1. Identify the potential data races or synchronization issues
2. Explain why they are problematic

Code Sample 1

```rust
use std::thread;

fn main() {
    let mut counter = 0;

    let handle = thread::spawn(move || {
        for _ in 0..1000 {
            counter += 1;
        }
    });

    for _ in 0..1000 {
        counter += 1;
    }

    handle.join().unwrap();
    println!("Final counter: {}", counter);
}
```

Code Sample 2

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

fn main() {
    let data = vec![1, 2, 3, 4, 5];
    let index = Arc::new(AtomicUsize::new(0));
    let index_clone = Arc::clone(&index);

    let handle = thread::spawn(move || {
        index_clone.fetch_add(10, Ordering::Relaxed);
    });

    if index.load(Ordering::Relaxed) < data.len() {
        println!("Value: {}", data[index.load(Ordering::Relaxed)]);
    }
```

```
    handle.join().unwrap();
}
```

Code Sample 3

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

fn main() {
    let resource1 = Arc::new(Mutex::new(0));
    let resource2 = Arc::new(Mutex::new(0));

    let r1 = Arc::clone(&resource1);
    let r2 = Arc::clone(&resource2);

    let handle1 = thread::spawn(move || {
        let mut lock1 = r1.lock().unwrap();
        thread::sleep(std::time::Duration::from_millis(100));
        let mut lock2 = r2.lock().unwrap();

        *lock1 += 1;
        *lock2 += 1;
    });

    let handle2 = thread::spawn(move || {
        let mut lock2 = resource2.lock().unwrap();
        thread::sleep(std::time::Duration::from_millis(100));
        let mut lock1 = resource1.lock().unwrap();

        *lock2 += 1;
        *lock1 += 1;
    });

    handle1.join().unwrap();
    handle2.join().unwrap();
}
```