Concurrency

CAS CS 392: Rust, in Theory and in Practice

March 4, 2025 (Lecture 12)

Outline

An Overview of Concurrency in Rust

Workshop

Introduction

```rust
let counter = Arc::new(Mutex::new(0));
//...
let counter = Arc::clone(&counter);
let handle = thread::spawn(move || {
    let mut num = counter.lock().unwrap();
    *num += 1;
});
```

Concurrency is, in some sense, core to the design of Rust

Rust offers both **message passing** and **shared-state** concurrency

High Level

Threads are abstractions for units of a process which can be handled independently

The trick: order of operations between multiple threads is not guaranteed, which can lead to:

- Race conditions: threads accessing data in inconsistent order
- Deadlocks: threads waiting on each other

1:1 Threading Model

A **thread** (or kernel thread) is an OS-level abstraction. Each thread is dealt with by the *scheduler* of the OS

It's not uncommon to have user-level thread abstractions (e.g., in a VM)
Rust does not do this

There is a 1-1 correspondence betweenn user-threads and OS-threads in Rust

Spawning Threads

```
thread::spawn(|| {
    for i in 1..10 {
        println!("{i}");
    }
});
```

`thread::spawn` takes a closure, which defines what is done by the thread

Important. Spawning a thread does not guarantee that it's corresponding computation will finish

The main thread (in which the new thread was spawned) may finish first and drop any computation

Joining Threads

We can "wait" for a spawned thread to finish by using `.join()`:

```
let handle = thread::spawn(|| {
  for i in 1..10 {
      println!("{i} (from spawned)");
  }
});
handle.join().unwrap();
```

If we don't do this, the spawned thread may not run at all

Note: The the joiner *owns* and is consumed:

```
pub fn join(self) -> Result<T>
```

Joining *blocks* the owning thread, code put after isn't run until the spawned thread is done

Move Closures

We often need `move` closures when working with threads:

```
// THIS DOES NOT COMPILE
fn main() {
    let v = vec![1, 2, 3];

    // should replace with `move || {...}`
    let handle = thread::spawn(|| {
        println!("{v:?}");
    });
    drop(v);
    handle.join().unwrap();
}
```

It's possible for a value to get dropped before the thread is done!
(**Question:** *What if we put the drop after the join?*)

Another Look at Spawning

```
pub fn spawn<F, T>(f: F) -> JoinHandle<T>
where
    F: FnOnce() -> T + Send + 'static,
    T: Send + 'static,
```

The lifetime bound on `F` ensures that only global things can live as long as the input closure

This necessitates `move` is *most* cases

Message passing

"Do not communicate by sharing memory; instead, share memory by communicating."

In Rust we can create *multi-producer single-consumer channels* which can be used for passing messages between threads:

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

//...
let (tx, rx) = mpsc::channel();
```

Example

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

fn main() {
    let (tx, rx) = mpsc::channel();

    thread::spawn(move || {
        let val = String::from("hi");
        tx.send(val).unwrap(); // tx is moved here
    });

    let received = rx.recv().unwrap();
    println!("Got: {received}");
}
```

Recieving messages blocks the main thread (so no need for joining)

Message Passing and Ownership

```rust
// THIS DOES NOT COMPILE
thread::spawn(move || {
    let val = String::from("hi");
    tx.send(val).unwrap();
    println!("val is {val}");
});
```

Sending a message *transfers ownership*:

```rust
pub fn send(&self, t: T) -> Result<(), SendError<T>>
```

The *type system* can express that a message should not be used after being sent

Shared State Concurrency

We can also do standard shared state concurrency with `Mutex` in Rust:

```
let counter = Arc::new(Mutex::new(0));
//...
let counter = Arc::clone(&counter);
let handle = thread::spawn(move || {
    let mut num = counter.lock().unwrap();
    *num += 1;
});
```

If we're sharing state then we must use a reference counter

It *has* to be atomic (which basically means updating the reference count can't be interrupted partway through the update)

Concurrent vs. Sequential

`Rc<T>` is to `Arc<T>` as `RefCell<T>` is to `Mutex<T>`

`RefCell` and `Mutex` both allow for internal mutability

`Rc` + `RefCell` leads to memory leaks, `Arc` + `Mutex` leads to deadlocks

Takeaways

The compelling part of concurrency in Rust is not that it handles concurrency better than in other languages, but that *the concerns of concurrency fits into the ownership paradigm very well*

- When we pass a value as a message, we shouldn't be able to work with it anymore. That can be represented as transferring ownership once the value is sent
- We should be careful and explicit when sharing data across threads, that's built into the way we use Rust

Next Steps

There isn't much interesting we can do with concurrency until we have something like a **thread pool**, which is a structure for spawning threads and giving them jobs when they're ready

There also isn't that much interesting we can do until we have an interesting application. . .

There are a lot of interesting libraries for concurrency in Rust, it's worthwhile to checkout the ecosystem (e.g., Rayon is a popular and simple library for parallel iterators)

There are other models of concurrency! Rust also support asynchronous computation

Outline

Tasks

- **Practice Problem.** Implement a function which determines whether the sum of elements of a `Vec<i32>` is even or odd. *Implement a version which spawns a fixed number of threads to process chunks of the input and combines the answers at the end*
- Read through RPL 21.2 on building a thread pool
- **(Long) Practice Problem.** Design a `Set` data structure using binary search trees (don't worry about balancing). Implement an interface with: `empty`, `insert`, and `mem` (membership). Finally, design an iterator structure for sets and implement the `IntoIterator` trait for `Set` (*Challenge.* Implement an `iter` method for `Set` and use this to implement `IntoIterator` for `&Set`)
- Finish the Gödel numbering practice problems