Threads & Mutual Exclusion in Rust

Lecture 14

Slides: Neil Kaushikkar

Goals For Today

- More Ways to Share Data Between Threads
- Mutex Locks and Shared State
- Atomicity & Atomic Reference Counts

Reminders

- HW10 due 10/18 at 11:59pm
- MP3 due 10/21 at 11:59pm
- Final Project Logistics incoming next week!

Sending/Sharing Data Between Threads

- To communicate between threads, we can:
  - Pass messages between threads
  - Use shared memory
- Pass messages between threads using **mpsc::Sender** and **mpsc::Receiver**
- We can share data between threads using a Mutex Lock
  - Allow threads to share memory taking advantage of some advanced features built into Rust's Type system

Mutex Locks

- Mutex is an abbreviation for mutual exclusion
  - Mutual: Held in common
  - Exclusion: Deny Access
  - A mutex allows only one thread to access some data at any given time
- Example:
  - A panel of speakers at a conference and want 1 person to speak at a time
  - Use a microphone so only 1 person can be heard throughout the conference room at a time
  - The microphone acts as the mutex - it only allows one speaker to access the audience's attention at a given time

Reference:

- https://doc.rust-lang.org/book/ch16-03-shared-state.html

Mutex Locks

- To access the data in a mutex, a thread must first signal that it wants access by asking to acquire the mutex's lock
- The lock is a data structure that is part of the mutex
  - Keeps track of who currently has exclusive access to the data
  - Only 1 thread can hold the lock at a time
- The mutex is described as guarding the data it holds via the locking system
  - Different from other languages where Mutexes guard "critical sections"
  - A critical section is a piece of code that must not be run by multiple threads at once because the code accesses shared resources

Reference:

- https://doc.rust-lang.org/book/ch16-03-shared-state.html

Mutexes and the Critical Section Problem

- A critical section is a piece of code that must not be run by multiple threads at once because the code accesses shared resources
- Rules:
  - You must attempt to acquire the lock before using the shared data
  - When you're done with the data that the mutex guards, you must unlock the data so other threads can acquire the lock

Reference:

- https://doc.rust-lang.org/book/ch16-03-shared-state.html

Mutex in Rust

- **Mutex::new(data: T)** - create a new mutex to provide mutual exclusion to **data**
- **mutex.lock().unwrap()** - Returns a **MutexGuard** for the data in the mutex
  - Attempt to acquire the mutex's lock
  - If another thread holds the lock, wait until it is unlocked and try again
- Mutexes in Rust are automatically unlocked when the **MutexGuard** returned by lock goes out of scope (and is dropped from memory)
  - You do not need to worry about unlocking mutexes in Rust!

Reference:

- https://doc.rust-lang.org/std/sync/struct.Mutex.html

Using Mutexes in Rust (Bad Example)

```rust
use std::sync::Mutex;
use std::thread;

fn main() {
    let mut handles = Vec::new();
    let data = Mutex::new(0);

    for _ in 0..10 {
        let h = thread::spawn(move || {
            for _ in 0..20 {
                *data.lock().unwrap() += 1;
            } // the MutexGuard returned by .lock() goes out of scope
            // Other threads can attempt to lock the mutex now...
        });

        handles.push(h);
    }

    for h in handles.into_iter() {
        h.join().unwrap();
    }

    println!("The final count is {}", data.lock().unwrap());
}
```

```
error[E0382]: use of moved value: `data`
  --> lecture14/src/main.rs:9:31
   |
6  |     let data = Mutex::new(0);
   |         ---- move occurs because `data` has type `Mutex<i32>`, which does not implement the `Copy`
trait
...
9  |         let h = thread::spawn(move || {
   |                               ^^^^^^^ value moved into closure here, in previous iteration of loop
10 |             for _ in 0..20 {
11 |                 *data.lock().unwrap() += 1;
   |                  ---- use occurs due to use in closure
```

REMEMBER!

- Only 1 thread can have ownership of some value

Sharing Data Between Threads

- We need some advanced type to allow us to share our mutex between threads
- We can use the **Arc** smart pointer to help!
  - Atomic Reference Count smart pointer
  - Keeps a count of the number of references to the data inside the pointer
  - Only drop the data when the number of references is 0
  - Atomic???
    - When accessing or mutating a property is atomic, it means that only one read or write operation can be performed at a time
  - Thread safe count of the number of references == thread safe pointer!

Reference:

- https://doc.rust-lang.org/std/sync/struct.Mutex.html

Atomicity

- In your CPU, when you want to modify a value, the CPU...
  - Loads a value from memory into a register
  - Modifies that value
  - Stores it back into memory
- Remember, there's no guarantee of thread execution order!

Atomicity

- Consider the following (non-atomic) sequence of events with 2 threads on an integer **data**:
  - Initially: **data = 0;**
  - T1: load data into register %AX (**data = 0**, %AX = 0)
  - T2: load data into register %BX (**data = 0**, %BX = 0)
  - T1: data += 1 (**data = 0**, %AX = 1)
  - T2: data += 1 (**data = 0**, %BX = 1)
  - T1: write data (the value on register %AX) back into memory (**data = 1**)
  - T2: write data (register %BX) back into memory (**data = 1**)
- Uh Oh! We incremented data twice, but it's value is only 1
- With atomic operations, we want writes (operations that modify some data) to happen instantaneously so that any other readers/writers will always "see" the correct value

Arc Smart Pointers in Rust

- **Arc::new(data: T)** - Create a new Arc smart pointer that has ownership of **data**
- **pointer.clone()** - Create new Arc smart pointer that references the same data the original pointer references!
  - Separate structure, but same reference
  - Increment the number of references to the data
- Pass the cloned pointer to your thread so you can continue to reuse/clone the original **Arc** when passing data to other threads

Reference:

- https://doc.rust-lang.org/std/sync/struct.Mutex.html

Using Mutexes in Rust

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

fn main() {
    let mut handles = Vec::new();
    let data = Arc::new(Mutex::new(0));

    for _ in 0..10 {
        let clone = data.clone();
        let h = thread::spawn(move || {
            for _ in 0..20 {
                *clone.lock().unwrap() += 1;
            } // the MutexGuard returned by .lock() goes out of scope
            // Other threads can attempt to lock the mutex now...
        });

        handles.push(h);
    }

    for h in handles.into_iter() {
        h.join().unwrap();
    }

    println!("The final count is {}", data.lock().unwrap());
}
```

REMEMBER!

- Pass the cloned Arc to each thread

Takeaway

- **Arc::new(data: T)** - Create a new Arc smart pointer that has ownership of **data**
- **pointer.clone()** - Create new Arc smart pointer that references the same data the original pointer references!
  - Separate structure, but same reference
  - Increment the number of references to the data
- Pass the cloned pointer to your thread so you can continue to reuse/clone the original **Arc** when passing data to other threads

Reference:

- https://doc.rust-lang.org/std/sync/struct.Mutex.html

That’s All Folks!