ECE459: Programming for Performance Winter 2019

Lecture 38 — Rust Concurrency

Patrick Lam *2019-04-05*

Functions and Return Values

Move semantics also applies to function calls and return values, e.g.

```
fn main() {
  let s = String::from("moving to callee");
  callee(s); // afterwards, s is invalid
}
fn callee(param:String) {
  println!("got {}, param");
} // param goes out of scope, object dropped
```

If you return something, then the ownership passes back to the caller. Tuples help pass ownership of multiple objects, but this is still quite high-overhead for developers.

"Can I please borrow your object?"

```
fn main() {
  let s = String::from("459");
  let len = calculate_length(s);
  println!("string {} has length {}", s /* we still have it! */,
           len);
}
fn calculate_length(s:&String) -> usize { // note the & for borrow
  s.len() // last expr is return value
} // s is ref so nothing goes out of scope
```

Borrowing and mutation

Like other variables, references are immutable by default. We can have mutable references, though.

```
fn change(s:&mut String) {
  s.push_str("more");
}
fn main() {
  let mut main_str = String::from("some ");
  change(&mut main_str); // create mutable ref to main_str
}
```

There can be only one (mutable). The following code won't compile:

```
let mut s = String::from("one");
let r1 = &mut s;
let r2 = &mut s; // rustc complains!
```

In fact, while `r1` is in scope, you can't do anything with the original `s`. The only way to access the string is through `r1`. Once `r1` goes out of scope, you can create `r2`.

Since there is only one way to access `r1`, then there will be no race conditions.

This is OK:

```
let mut s = String::from("one");
let r1 = &s;
let r2 = &s; // no problem!
```

But you can't then do `let r3 = &mut s;`.

How many? You can have as many outstanding immutable refs as you want. If there are any immutable refs, you can't have *any* mutable refs. The immutable refs must go out of scope first.

You also can't commit use-after-free errors: you can't return a ref that outlives its value. Would've been useful on A4!

```
fn dangle() -> &String {
  let s = String::from("hello");
  &s // rustc complains: s goes out of scope with active refs
}
```

Rust also has *smart pointers*, which may be reference counted. This is like C++'s smart pointers, specifically `shared_ptr` (but Rust can tell you about some things at compile time which C++ will tell you at runtime). Normal Rust objects are more like `unique_ptr`.

We need reference counted heap objects e.g. to implement graphs. We don't have enough time to talk about smart pointers, but Chapter 15 of the Rust book is good.

Fearless Concurrency

As with many other aspects of Rust, we trade compiler errors for runtime errors; in this case, runtime concurrency errors like race conditions. That is, the type system ensures concurrency safety!

Rust uses a fork/join model like pthreads. It delegates to the operating system's threads support and hence implements 1:1 threads.

```
let handle = thread::spawn(|| { // closure (can put args between ||)
  // thread code goes here
});
// main thread continues here
handle.join.unwrap(); // unwrap: panic in case of error
```

This is not too different from C++.

OK, how do we share data between threads? We can move it from main to thread:

```
let v = vec![1,2,3];
let handle = thread::spawn(move || { // move: everything accessed inside closure is moved
  println!("vector␣{:?}", v);
}); // no longer have access to v in main
handle.join().unwrap();
```

Rust is saving you from being able to concurrently access `v` in main and thread.

But that's only one way! This isn't quite enough.

Message Passing

One way to share data is message passing. We've seen this before, when we talked about OpenMPI. In this case, each value still only has one owner. We use *channels*. The ownership passes from the sender, through the channel, to the receiver.

```
use std::thread;
use std::sync::mpsc; // multi producer, single consumer

fn main() {
  let (tx, rx) = mpsc::channel(); // tx is cloneable
  thread::spawn(move || { // here, tx goes to closure
    let val = String::from("april");
```

```
    tx.send(val).unwrap(); // val moved from sender
  });
  let received = rx.recv().unwrap();
  println!("got: {}", received);
}
```

Note the send/receive pair. There is also `try_recv` to do nonblocking receives.

Shared State

People debated for a long time which was better: shared state (like pthreads) or channels. Rust supports both. Of course, the problem with shared state is race conditions. Like manual memory management, we can manually acquire and release mutexes. What could possibly go wrong? Rust's ownership system will help.

We'll need to talk about multiple ownership. But let's talk about mutexes first.

```
use std::sync::Mutex;
fn main() {
  let m = Mutex::new(5); // mutex guards access to an i32
  {
    let mut num = m.lock.unwrap();
    // unwrap: maybe some other thread panicked while holding lock;
    //         then we panick too.
    *num = 6; // "deref" the mutex (is actually a smart pointer)
  } // release lock when num goes out of scope
  println!("m={:?}", m);
}
```

Well, that's fine, but it's just one thread. We really do need multiple ownership to share data. The shared data needs to be owned by all threads, and a naive solution will get rejected by the borrow checker. Instead, we have to use *reference counted cells*, implemented by `Arc`.

```
use std::sync::{Mutex, Arc};
use std::thread;
fn main() {
  let counter = Arc::new(Mutex::new(0)); // atomic reference cell
  let mut handles = vec![];

  for _ in 0..10 {
    let counter = Arc::clone(&counter); // clone the Arc
    let handle = thread::spawn(move || {
      let mut num = counter.lock().unwrap();
      *num += 1;
    });
    handles.push(handle);
  }
  for handle in handles {
    handle.join.unwrap();
  }
  println!("result: {}", *counter.lock().unwrap());
}
```

Rust guarantees that you have the appropriate lock, using ownership (possibly multiple ownership). Rust does not guarantee lack of deadlocks.