LECTURE 23

Theory and Design of PL (CS 538)

April 15, 2020

SHARED-STATE CONCURRENCY IN RUST: MUTEX

IN RUST: MUTEX

- Most common operations
  - `new`: make a new mutex
  - `lock`: acquire lock, blocks if other thread has lock

```
let my_mutex = Mutex::new(5);

// wait for the lock
let mut data_inside = my_mutex.lock().unwrap();

// have the lock, write to value inside
*data_inside = 6;

// check what value is now
println!("my_mutex = {}", *data_inside);
```

WHY LOCK RETURNS RESULT?

- Error handling is tricky in the presence of threads
- If a thread panics, no good solutions
  - Kill all threads? But threads should be separate.
  - Keep going? Panicking thread might have been in the middle of some complicated operation.

LOCK POISONING

- Rust: if thread panics holding Mutex, it is *poisoned*
- Later calls to `lock()` will return `Err(ptr)`
- Signal: someone panicked while holding this lock
  - Were in critical section, but didn't finish
  - May not be safe to enter critical section now
- Usually: you should just call `.lock().unwrap()`
  - "If someone else panicked, I'm panicking too"
- Can get pointer into lock with `ptr.into_inner`.

WHAT ABOUT UNLOCKING?

- Most languages: need to unlock once done with lock
- Common bug: forgetting to unlock
  - No one else can get the lock!
- In Rust: locking a Mutex gives smart pointer
  - Automatically unlocks when it is dropped

Ownership to the rescue!

FORCING UNLOCKING

- Sometimes, want to unlock a lock “early”
- Either close the scope, or call std::mem::drop

```
let my_mutex = Mutex::new(5);

{ // start new scope
  // wait for the lock
  let mut data_inside = my_mutex.lock().unwrap();

  // holding the lock, write to value inside
  *data_inside = 6;

  // explicit unlock: std::mem::drop(data_inside);
} // or: scope ends, automatically unlocked here

// no longer holding lock here
```

WHO OWNS THE MUTEX?

- Mutex shared between threads, no single owner

```
let counter = Mutex::new(0);
let mut handles = vec![];
for i in 0..10 {
  let handle = thread::spawn(move || {      // move lock in
    let mut num = counter.lock().unwrap(); // acquire lock
    *num += 1;
  });
  handles.push(handle);
}

for handle in handles { handle.join(); }
```

- Fails: Mutex can’t be owned by multiple threads!

ANOTHER TRY

- Use `Rc` to allow multiple owners of Mutex

```
let counter = Rc::new(Mutex::new(0)); // allow mutex to be shared
let mut handles = vec![];
for i in 0..10 {
  let rc_count = Rc::clone(&counter);  // get a ref to the mutex
  let handle = thread::spawn(move || {
    let mut num = rc_count.lock().unwrap();
    *num += 1;
  });
  handles.push(handle);
}

for handle in handles { handle.join(); }
```

- Compiler is not happy: “`Rc` is not `Sync`”

SOLUTION: USE ARC

- Common pattern for using/sharing Mutex in Rust

```
let counter = Arc::new(Mutex::new(0)); // use atomic Rc
let mut handles = vec![];
for i in 0..10 {
  let rc_count = Arc::clone(&counter);  // get a ref to the mutex
  let handle = thread::spawn(move || {
    let mut num = rc_count.lock().unwrap();
    *num += 1;
  });
  handles.push(handle);
}

for handle in handles { handle.join(); }
```

A BANK ACCOUNT, LOCKS

- Tweak bank account code from previous lecture

```
struct Account { balance: i32 };

impl Account {
  fn deposit(&mut self, amt: i32) { self.balance += amt; }

  fn try_withdraw(&mut self, amt: i32) -> Result<i32, &str> {
    if self.balance < amt {
      Err("Insufficient funds.")
    } else {
      self.balance -= amt;
      Ok(self.balance)
    }
  }
}
```

A BANK ACCOUNT, LOCKS

- Wrap account in Mutex, share between clients

```
let acct = Account { balance: 100 };
let mutex = Mutex::new(acct);          // wrap account in lock
let rc_mutex = Arc::new(mutex);        // owner 1 of mutex
let rc_copy = Arc::clone(rc_mutex);    // owner 2 of mutex

thread::spawn(move || {
  let acct_ptr = rc_mutex.lock().unwrap();   // try to get lock
  acct.try_withdraw(100);   // got lock: try withdrawl
})

thread::spawn(move || {
  let acct_ptr = rc_copy.lock().unwrap();   // try to get lock
  acct.try_withdraw(100);   // got lock: try withdrawl
})
```

ANOTHER PRIMITIVE: CONDVARS

CONDITION VARIABLES

- Used for waiting and signalling threads
- This is how condvars work:
  1. T1 holds lock L and waits on condvar C
  2. T1 sleeps, L is auto unlocked
  3. T2 can notify (one or all) threads waiting on C
  4. T1 wakes up and tries to grab L

SIGNALING A CONDVAR

- From `std::sync::Condvar` docs...

```
let p = Arc::new((Mutex::new(false), Condvar::new()));
let q = Arc::clone(&p);

// Spawn a new thread, which will signal when it starts
thread::spawn(move || {
    let my_lock = q.0;
    let my_cvar = q.1;
    let mut started = my_lock.lock().unwrap();  // grab lock
    *started = true;

    // We notify the condvar that the value has changed.
    my_cvar.notify_one();
});  // lock released here (started out of scope)
```

WAITING ON A CONDVAR

- From `std::sync::Condvar` docs...

```
let p = Arc::new((Mutex::new(false), Condvar::new()));
let q = Arc::clone(&p);

// Spawn a new thread, which will signal when it starts
thread::spawn(move || { ... });

let my_lock = p.0;
let my_cvar = p.1;

let mut started = my_lock.lock().unwrap();  // grab lock

// Spin: sleep-wake until flag is true
while !*started { started = my_cvar.wait(started).unwrap(); }
```

CONDVAR: PITFALLS

- Don't assume thread wakes up "right after" signal
  - Maybe: many threads signaled, you are not first
  - Maybe: "spurious" wakeups
  - Always check if the wakeup is "for your thread"
- Waiting with multiple locks
  - In Rust: you probably don't want to do this
  - One lock will be released, but others *still locked*
  - Can easily lead to deadlocks

A BANK ACCOUNT, CONDVARS

- Setup: one account under Mutex, one Condvar

```
let acct = Account { balance: 50 };

let rc_acct1 = Arc::new(Mutex::new(false));
let rc_acct2 = Arc::clone(rc_acct1);
let rc_cvar1 = Arc::new(Condvar::new());
let rc_cvar2 = Arc::clone(rc_cvar1);
```

A BANK ACCOUNT, CONDVARS

- First client: try to withdraw, wait if it can't

```
thread::spawn(move || {
  let mut acct_ptr = rc.acct1.lock().unwrap();  // try get lock
  loop {
    if acct_ptr.try_withdraw(125).is_ok() {      // try withdraw
      break;
    } else {
      // not enough funds: release lock and sleep until notified
      acct_ptr = rc_cvar1.wait(acct_ptr).unwrap();
    }
  }
});
```

A BANK ACCOUNT, CONDVARS

- Second client: make deposit, notify condvar

```
thread::spawn(move || {
  // try to get lock
  let acct_ptr = rc_acct2.lock().unwrap();

  // got the lock, do deposit
  acct_ptr.deposit(100);

  // notify (all) waiters
  rc_cvar2.notify_all();
});
```

- Question: why notify all, instead of notify one?

MESSAGE PASSING IN RUST

RECALL THE IDEA

- Threads interact by sending/receiving messages
- Make threads as modular as possible
  - Limit all interaction to specific places
  - No shared state, no data races
- Simplify error handling
  - No mutexes, no poisoning
  - Restart threads after errors

COMMUNICATE ONLY THROUGH CHANNELS

- Main abstraction: *channels* between threads
- Threads send/receive messages along channels
  - Wait on messages (synchronous)
  - Send and continue (asynchronous)

(A)SYNCHRONOUS CHANNELS

- Receiving messages: blocking or not?
  - Receive/try-receive
- Sending messages: blocking or not?
  - Nonblocking: asynchronous channels
  - Blocking: synchronous channels
- Also known as *unbounded* and *bounded*

ASYNCHRONOUS CHANNELS

- Found in `mpsc::channel`
  - Multiple producers
  - Single consumer
- Pair of objects: *transmitter* and *receiver*
- Synchronous channels in `mpsc::sync_channel`

CREATING A CHANNEL

- Construct a pair of endpoints
  - Typically: `tx` for transmit, `rx` for receive
- Spawn thread and pass it one endpoint
  - Use `move` to transfer ownership of endpoint

```
fn main() {
  let (tx, rx) = mpsc::channel();     // set up channel pair

  thread::spawn(move || {             // move tx end to child
    let val = String::from("hi");
    tx.send(val);                     // child sends message
  });
}
```

TRANSMITTING END SENDS

- Sending returns a `Result` type
  - Error if something goes wrong
  - Example: transmit end already dropped (closed)
- Error handling: use `unwrap` to stop program if error

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

  thread::spawn(move || {
    let val = String::from("hi");
    tx.send(val).unwrap();            // panic if send fails
  });
}
```

BLOCKING RECEIVE

- Blocking `recv` waits for a message to be delivered

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

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

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

NON-BLOCKING RECEIVE

- Non-blocking `try_recv` returns immediately
- Returns error in `Result` if there was no message

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

  thread::spawn(...);

  let maybe_recv = rx.try_recv();        // don't panic if error
  match maybe_recv {
    Err(e) => println!("Got nothing so far!");
    Ok(v)  => println!("Got something: {}", v);
  }
}
```

ITERATOR RECEIVE

- Can treat receiving end as an iterator

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

  thread::spawn(move || {
    let vals = vec![ String::from("hi"), String::from("from"),
                     String::from("the"), String::from("thread"),
                   ];
    for val in vals { tx.send(val).unwrap(); }
  });

  for received in rx {
    println!("Got: {}", received);
  }
}
```

CHANNELS AND OWNERSHIP

- Channels transfer ownership of data
  - Can’t use sent data after sending it across
- Only types implementing `Send` can be sent

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

  thread::spawn(move || {
    let val = String::from("hi");
    tx.send(val).unwrap();                 // transfer ownership
    println!("val is {}", val);            // Not OK: can't use val!
  });

  let received = rx.recv().unwrap();   // receive ownership
  println!("Got: {}", received);       // OK: can use val
}
```

CLONING CHANNELS

- Clone transmit end to let multiple threads send
- Receiver will see all messages (in some order)

```
let (tx, rx) = mpsc::channel();
let tx_copy  = mpsc::Sender::clone(&tx); // copy transmit end
thread::spawn(move || {    // make thread with tx
  // ...
  tx.send(val).unwrap();
});
thread::spawn(move || {    // make thread with tx_copy
  // ...
  tx_copy.send(val).unwrap();
});

// receive messages
for received in rx { println!("Got: {}", received); }
```

DROPPING CHANNELS

- Sending to dropped receiver returns None
- Receiving from dropped sender returns None
- Channel deallocated when both ends dropped

FANCIER CHANNELS

- Multiple producer, multiple consumer
- Selection (sending and receiving)
- Mostly in external crates (crossbeam)

EVEN MORE PRIMITIVES

ATOMICS

- Usually small cells (single int, bool, etc.)
- Operations guaranteed to be atomic
  - Cannot be interruptible by other threads
  - Load, store, compare-and-swap, ...
- No need to lock when accessing
  - In fact, often used to implement locks

EXAMPLE: ATOMICS

- From `std::sync::atomic` docs...

```
fn main() {
  let spinlock = Arc::new(AtomicUsize::new(1));
  let spinlock_clone = spinlock.clone();

  // child "has lock" ==> spinlock = 1
  let thread = thread::spawn(move|| {
    // child "releases lock"
    spinlock_clone.store(0, Ordering::SeqCst);
  });
  // spin: wait for child to "release lock"
  while spinlock.load(Ordering::SeqCst) != 0 {}

  // continue onwards
}
```

BARRIERS

- Allows multiple threads to sync and continue
- Specify number of threads when constructing
- Each thread calls wait, blocks until all have called

EXAMPLE: BARRIERS

- From `std::sync::Barrier` docs...

```
// Barrier that waits for 10 threads
let barrier = Arc::new(Barrier::new(10));

for i in 0..10 {
  let c = barrier.clone();
  thread::spawn(move|| {
    println!("here");
    c.wait();
    println!("there");
  }));
}
```

- No interleaving: all “here” before “there”

READER-WRITER LOCKS

- Souped-up version of Mutex
- Like golden rules for references
  - Multiple threads can read at the same time
  - Only one thread can write (and no readers)
- Checked at runtime: panics if rules violated

EXAMPLE: RWLOCK

- From `std::sync::RwLock` docs...

```
let lock = RwLock::new(5);

// many reader locks can be held at once
{
  let r1 = lock.read().unwrap();
  let r2 = lock.read().unwrap();
} // read locks are dropped at this point

// only one write lock may be held at a time
{
  let mut w = lock.write().unwrap();
  *w += 1;
} // write lock is dropped here
```

CRATES TO KNOW

STDLIB PROVIDES THE BASIC

- Aim is to keep stdlib small
- Many other crates relate to concurrency
  - Often much fancier than stdlib
  - Some will eventually be put into stdlib

CROSSBEAM

- Utilities for general-purpose concurrency
- Lock-free (non-blocking) concurrent collections
  - Memory management for concurrent collections
- Better channels, better performance
  - Multiple producer, *multiple* consumer
  - Select between channels
- Scoped threads: use regular borrows instead of `Arc`

TOKIO AND ASYNC-STD

- Libraries for "asynchronous" concurrency
  - Concurrency on a single thread
- Running, switching, and waking up jobs
- Highly sophisticated libraries
- Few months ago: compiler support ("async/await")
- We'll go into a lot more detail next week…