Concurrency I

CIS 198 Lecture 10

Misc.

Rust 1.7 releasing on Thursday, 3/03!

- If using multirust: `multirust update stable`
- If not: just run the installer again, it will provide the option to update a toolchain rather than re-installing it.

What is Concurrency?

- One program with multiple threads of execution running at the same time.
- Threads can share data without communication overhead.
    - (networking, inter-process communication channels, etc).
- Threads are more lightweight than individual processes.
    - No large OS context switch when switching between threads.

What is a Thread?

- A context in which instructions are being executed.
- References to some data (which may or may not be shared).
- A set of register values, a stack, and some other information about the current execution context (low-level).

Threads

- Conceptually, every program has at least one thread.
- There is a thread scheduler which manages the execution of these threads.
  - It can arbitrarily decide when to run any thread.
- Programs can create and start new threads, which will be picked up by the scheduler.

Concurrent Execution

- Take these two simple programs, written in pseudo-Rust (ignoring ownership semantics):

```
let mut x = 0;

fn foo() {
    let mut y = &mut x;
    *y = 1;
    println!("{}", *y); // foo expects 1
}

fn bar() {
    let mut z = &mut x;
    *z = 2;
    println!("{}", *z); // bar expects 2
}
```

Instruction Interleaving

- Imagine two threads: one executing `foo`, one executing `bar`.
- The scheduler can interleave instructions however it wants.
- Thus, the above programs may be executed like this:

```
/* foo */ let mut y = &mut x;
/* foo */ *y = 1;
/* foo */ println!("{}", *y); // foo expects 1
          // => 1
/* bar */ let mut z = &mut x;
/* bar */ *z = 2;
/* bar */ println!("{}", *z); // bar expects 2
          // => 2
```

- ...and everything works as expected.

Instruction Interleaving

- However, there is no guarantee that execution happens in that order every time, or at all!
- We need some mechanisms to ensure that events happen in an order that produces the expected results.
- Otherwise, `foo` and `bar` may be interleaved arbitrarily, causing unexpected results:

```
/* bar */ let mut z = &mut x;
/* bar */ *z = 2;
/* foo */ let mut y = &mut x;
/* foo */ *y = 1;
/* bar */ println!("{}", *z); // bar expects 2
          // => 1
/* foo */ println!("{}", *y); // foo expects 1
          // => 1
```

Why is concurrency hard?

- **Sharing data:** What if two threads try to write to the same piece of data at the same time?
  - Writing to `x` in the previous example.
- **Data races:** The behavior of the same piece of code might change depending on when exactly it executes.
  - Reading from `x` in the previous example.

Why is concurrency hard?

- **Synchronization:** How can I be sure all of my threads see the correct world view?
  - A series of threads shares the same buffer. Each thread `i` writes to `buffer[i]`, then tries to read from the entire buffer to decide its next action.
  - When sending data between threads, how can you be sure the other thread receives the data at the right point in execution?
- **Deadlock:** How can you safely share resources across threads and ensure threads don't lock each other out of data access?

Deadlock

- A deadlock occurs when multiple threads want to access some shared resources, but end up creating a state in which no one is able to access anything.
- There are four preconditions for deadlock:
  - Mutual exclusion: One resource is locked in a non-sharable mode.
  - Resource holding: A thread holds a resource and asks for more resources, which are held by other threads.
  - No preemption: A resource can only be released voluntarily by its holder.
  - Circular waiting: A cycle of waiting on resources from other threads exists.
- To avoid deadlock, we only have to remove *one* precondition.

Dining Philosophers

- An oddly-named classical problem for illustrating deadlock.
- N philosophers sit in a circle and want to alternate between eating and thinking.
- Each philosopher needs two forks to eat. There are `N` forks, one between every pair of philosophers.
- Algorithm:
  - A philosopher picks up their left fork. (Acquire a resource lock)
  - They pick up their right fork. (Acquire a resource lock)
  - They eat. (Use the resource)
  - They put both forks down. (Release resource locks)

Dining Philosophers

- What happens when we do this?
  - Let N = 3, for simplicity.
- Philosopher 1 picks up their left fork.
- Philosopher 2 picks up their left fork.
- Philosopher 3 picks up their left fork.
- Philosophers 1, 2, and 3 all try to pick up their right fork, but get stuck, since all forks are taken!

Dining Philosophers

- A better algorithm?
- We'll revisit this at the end of the lecture.

Rust Threads

- Rust's standard library contains a threading library, `std::thread`.
  - Other threading models have been added and removed over time.
  - The Rust "runtime" was been removed.
- Each thread in Rust has its own stack and local state.
- In Rust, you define the behavior of a thread with a closure:

```rust
use std::thread;

thread::spawn(|| {
    println!("Hello, world!");
});
```

Thread Handlers

- `thread::spawn` returns a thread handler of type `JoinHandler`.

```
use std::thread;

let handle = thread::spawn(|| {
    "Hello, world!"
});

println!("{:?}", handle.join().unwrap());
// => Ok("Hello, world!")
```

- `join()` will block until the thread has terminated.
- `join()` returns an `Ok` of the thread's final expression (or return value), or an `Err` of the thread's `panic!` value.

`std::thread::JoinHandler`

- A thread is detached when its handler is dropped.
- Cannot be cloned; only one variable has the permission to join a thread.

`panic!`

- Thread panic is unrecoverable from *within* the panicking thread.
- Rust threads `panic!` independently of the thread that created them.
  - *Only* the thread that panics will crash.
  - The thread will unwind its stack, cleaning up resources.
  - The message passed to `panic!` can be read from other threads.
- If the main thread panics or otherwise ends, all other threads will be shut down.
  - The main thread can choose to wait for all threads to finish before finishing itself.

`std::thread::Thread`

- The currently running thread can stop itself with `thread::park()`.
- `Thread`s can be unparked with `.unpark()`.

```
use std::thread;

let handle = thread::spawn(|| {
    thread::park();
    println!("Good morning!");
});
println!("Good night!");
handle.thread().unpark();
```

- A `JoinHandler` provides `.thread()` to get that thread's `Thread`.
- You can access the currently running `Thread` with `thread::current()`.

Many Threads

- You can create many threads at once:

```
use std::thread;

for i in 0..10 {
    thread::spawn(|| {
        println!("I'm first!");
    });
}
```

Many Threads

- Passing ownership of a variable into a thread works just like the rest of the ownership model:

```rust
use std::thread;

for i in 0..10 {
    thread::spawn(|| {
        println!("I'm #{}!", i);
    });
}
// Error!
// closure may outlive the current function, but it borrows `i`,
// which is owned by the current function
```

- ...including having to obey closure laws.

Many Threads

- The closure needs to own `i`.
- Fix: Use `move` to make a movable closure that takes ownership of its scope.

```
use std::thread;

for i in 0..10 {
    thread::spawn(move || {
        println!("I'm #{}!", i);
    });
}
```

Send and Sync

- Rust's type system includes traits for enforcing certain concurrency guarantees.
- Send: a type can be safely transferred between threads.
- Sync: a type can be safely shared (with references) between threads.
- Both Send and Sync are marker traits, which don't implement any methods.

Send

```
pub unsafe trait Send { }
```

- A `Send` type may have its ownership tranferred across threads.
- Not implementing `Send` enforces that a type *may not* leave its original thread.
    - e.g. a C-like raw pointer, which might point to data aliased by another (mutable) raw pointer that could modify it in a thread-unsafe way.

Sync

```
pub unsafe trait Sync { }
```

- A `Sync` type cannot introduce memory unsafety when used across multiple threads (via shared references).
- All primitive types are `Sync`; all aggregate types containing only items that are `Sync` are also `Sync`.
  - Immutable types (`&T`) and simple inherited mutability (`Box<T>`) are `Sync`.
  - Actually, all types without interior mutability are inherently (and automatically) `Sync`.

`Sync`

- A type `T` is `Sync` if `&T` is thread-safe.
  - `T` is thread safe if there is no possibility of data races when passing `&T` references between threads
- Consequently, `&mut T` is also `Sync` if `T` is `Sync`.
  - An `&mut T` stored in an aliasable reference (an `& &mut T`) becomes immutable and has no risk of data races.
- Types like `Cell` are not `Sync` because they allow their contents to be mutated even when in an immutable, aliasable slot.
  - The contents of an `&Cell<T>` could be mutated, even when shared across threads.

Unsafety

- Both `Send` and `Sync` are `unsafe` to implement, even though they have no required functionality.
- Marking a trait as `unsafe` indicates that the implementation of the trait must be trusted to uphold the trait's guarantees.
  - The guarantees the trait makes must be assumed to hold, regardless of whether it does or not.
- `Send` and `Sync` are unsafe because thread safety is not a property that can be guaranteed by Rust's safety checks.
  - Thread unsafety can only be 100% prevented by *not using threads*.
- `Send` and `Sync` require a level of trust that safe code alone cannot provide.

Derivation

- `Send` is auto-derived for all types whose members are all `Sync`.
- Symmetrically, `Sync` is auto-derived for all types whose members are all `Send`.
- They can be trivially `impl`ed, since they require no members:

```
unsafe impl Send for Foo {}
unsafe impl Sync for Foo {}
```

Derivation

- If you need to remove an automatic derivation, it's possible.
  - Types which *appear* `Sync` but *aren't* (due to `unsafe` implementation) must be marked explicitly.
  - Doing this requires so-called "OIBITs": "opt-in builtin traits".

```
#![feature(optin_builtin_traits)]

impl !Send for Foo {}
impl !Sync for Foo {}
```

> *The acronym "OIBIT", while quite fun to say, is quite the anachronism. It stands for "opt-in builtin trait". But in fact, Send and Sync are neither opt-in (rather, they are opt-out) nor builtin (rather, they are defined in the standard library). It seems clear that it should be changed.* —nikomatsakis

Sharing Thread State

- The following code looks like it works, but doesn't compile:

```
use std::thread;
use std::time::Duration;

fn main() {
    let mut data = vec![1, 2, 3];

    for i in 0..3 {
        thread::spawn(move || {
            data[i] += 1;
        });
    }

    thread::sleep(Duration::from_millis(50));
}

// error: capture of moved value: `data`
//        data[i] += 1;
//        ^~~~
```

Sharing Thread State

- If each thread were to take a reference to `data`, and then independently take ownership of `data`, `data` would have multiple owners!
- In order to share `data`, we need some type we can share safely between threads.
  - In other words, we need some type that is `Sync`.

`std::sync::Arc<T>`

- One solution: `Arc<T>`, an **A**tomic **R**eference-**C**ounted pointer!
  - Pretty much just like an `Rc`, but is thread-safe due to atomic reference counting.
  - Also has a corresponding `Weak` variant.
- Let's see this in action...

Sharing Thread State

- This looks like it works, right?

```rust
use std::sync::Arc;
use std::thread;
use std::time::Duration;

fn main() {
    let mut data = Arc::new(vec![1, 2, 3]);

    for i in 0..3 {
        let data = data.clone(); // Increment `data`'s ref count
        thread::spawn(move || {
            data[i] += 1;
        });
    }

    thread::sleep(Duration::from_millis(50));
}
```

Sharing Thread State

- Unfortunately, not quite.

```
error: cannot borrow immutable borrowed content as mutable
                   data[i] += 1;
                   ^~~~
```

- Like `Rc`, `Arc` has no interior mutability.
- Its contents cannot be mutated unless it has one strong reference and no weak references.
  - Cloning the `Arc` naturally prohibits doing this.

Many Threads

- `Arc<T>` assumes its contents must be `Sync` as well, so we can't mutate anything inside the `Arc`. :(
- What could we do to solve this?
  - We can't use a `RefCell`, since already know these aren't thread-safe.
  - Instead, we need to use a `Mutex<T>`.

Mutexes

- Short for **Mut**ual **Ex**clusion.
- Conceptually, a mutex ensures that a value can only ever be accessed by one thread at a time.
- In order to access data guarded by a mutex, you need to acquire the mutex's lock.
- If someone else currently has the lock, you can either give up and try again later, or block (wait) until the lock is available.

`std::sync::Mutex<T>`

- When a value is wrappted in a `Mutex`, you must call `lock` on the `Mutex` to get access to the value inside. This method returns a `LockResult`.
- If the mutex is locked, the method will block until the mutex becomes unlocked.
  - If you don't want to block, call `try_lock` instead.
- When the mutex unlocks, `lock` returns a `MutexGuard`, which you can dereference to access the `T` inside.

Mutex Poisoning 🐍

- If a thread acquires a mutex lock and then panics, the mutex is considered *poisoned*, as the lock was never released.
- This is why `lock()` returns a `LockResult`.
  - `Ok(MutexGuard)`: the mutex was not poisoned and may be used.
  - `Err(PoisonError<MutexGuard>)`: a poisoned mutex.
- If you determine that a mutex has been poisoned, you may still access the underlying guard by calling `into_inner()`, `get_ref()`, or `get_mut()` on the `PoisonError`.
  - This may result in accessing incorrect data, depending on what the poisoning thread was doing.

Sharing Thread State

- Back to our example:

```rust
use std::sync::Arc;
use std::thread;
use std::time::Duration;

fn main() {
    let mut data = Arc::new(Mutex::new(vec![1, 2, 3]));

    for i in 0..3 {
        let data = data.clone(); // Increment `data`'s ref count
        thread::spawn(move || {
            let mut data = data.lock().unwrap();
            data[i] += 1;
        });
    }

    thread::sleep(Duration::from_millis(50));
}
```

Sharing Thread State

- At the end of this example, we put the main thread to sleep for 50ms to wait for all threads to finish executing.
  - This is totally arbitrary; what if each thread takes much longer than that?
- We have no way to synchronize our threads' executions, or to know when they've all finished!

Channels

- Channels are one way to synchronize threads.
- Channels allow passing messages between threads.
- Can be used to signal to other threads that some data is ready, some event has happened, etc.

`std::sync::mpsc`

- **M**ulti-**P**roducer, **S**ingle-**C**onsumer communication primitives.
- Three main types:
  - `Sender`
  - `SyncSender`
  - `Receiver`
- `Sender` or `SyncSender` can be used to send data to a `Receiver`
- Sender types may be cloned and given to multiple threads to create *multiple producers*.
  - However, Receivers cannot be cloned (*single consumer*).

`std::sync::mpsc`

- A linked `(Sender<T>, Receiver<T>)` pair may be created using the `channel<T>()` function.
- `Sender` is an *asynchronous* channel.
- Sending data across the channel will never block the sending thread, since the channel is asynchronous.
  - `Sender` has a conceptually-infinite buffer.
- Trying to receive data from the `Receiver` will block the receiving thread until data arrives.

std::sync::mpsc

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

fn main() {
    let (tx, rx) = channel();
    for i in 0..10 {
        let tx = tx.clone();
        thread::spawn(move|| {
            tx.send(i).unwrap();
        });
    }
    drop(tx);

    let mut acc = 0;
    while let Ok(i) = rx.recv() {
        acc += i;
    }
    assert_eq!(acc, 45);
}
```

`std::sync::mpsc`

- A linked `(SyncSender<T>, Receiver<T>)` pair may be created using the `sync_channel<T>()` function.
- `SyncSender` is, naturally, synchronized.
- `SyncSender` *does* block when you send a message.
  - `SyncSender` has a bounded buffer, and will block until there is buffer space available.
- Since this `Receiver` is the same as the one we got from `channel()`, it will obviously also block when it tries to receive data.

`std::sync::mpsc`

- All channel send/receive operations return a `Result`, where an error indicates the other half of the channel "hung up" (was dropped).
- Once a channel becomes disconnected, it cannot be reconnected.

So, is concurrency still hard?

- **Sharing data is hard:** Share data with `Send`, `Sync`, and `Arc`
- **Data races:** `Sync`
- **Synchronization:** Communication using channels.
- **Deadlock:** ??

Dining Philosophers

- This illustrates a problem with the concurrency primitives we've seen so far:
  - While they can keep our code safe, they do not guard against all possible logical bugs.
  - These are not "magic bullet" concepts.
- Solutions?
- There are many:
  - The final philosopher pick up their forks in the opposite order.
  - A token may be passed between philosophers, and a philosopher may only try to pick up forks when they have the token.