Intro to Multithreading II

CS110L
February 23, 2022

Logistics

- **Channels and event-driven programming content will be optional** + use recorded content (rather than live lectures)
  - Using three **videos from 2021** posted on Canvas.
  - Optional **extensions to project 2**: use this material in your implementation!
  - This is really good content! I highly recommend it, and I'm happy to talk about it + answer questions now or after the quarter ends!
  - It's definitely a bit beyond 110 material and the core goals of this class.
- *This class is still relatively new; I appreciate your ongoing feedback so much, and I'm adjusting based on it!*

Logistics

- Plan:
  - **Monday:** scalability, availability, distributed systems, + **project 2 intro**
  - **Wednesday 3/2: project 2 work** / discussion
    - Or think of this as the "**channels**" lecture (and feel free to come talk + ask questions about channels!)
  - **Monday 3/7: project 2 work** / discussion
    - Or think of this as the "**event-driven programming**" lecture (and feel free to come talk + ask questions about event-driven programming)
- *Note: project 2 is much more open-ended than project 1, so **I recommend working together** + discussing your design decisions with others in the class!*

Logistics

- Last class (March 9th): **guest talk with Ryan Eberhardt**
  - Mark your calendars!!!!!
  - Co-designed this course, taught 110, and has lots of experience
  - In person with joinable Zoom link
  - Come chat about how this stuff applies in the real world, what you’ve liked + disliked about the course, what you’ve learned, etc.!
    - and/or come to boost your participation grade :)

Review: Multithreading in Rust

Spawning + joining

- Syntax for spawning and joining threads in Rust is similar to in C++

```
threads.push(thread::spawn(|| { /* Function for thread to run*/ });
```

Spawn the thread
Returns a thread object

Closure (lambda function in C++):
- What the thread will do

https://doc.rust-lang.org/std/thread/

Spawning + joining

- Syntax for spawning and joining threads in Rust is similar to in C++

`join` returns a *Result*
If the thread exited normally, this will be Ok
- Threads can return a value when they exit; that value will be stored in the Ok

If the thread panicked, this will be Err
- If some value was given to the *panic* (e.g., a string indicating an error message), that value will be stored in the Err.

```
for handle in threads {
    handle.join().expect("Panic occurred in thread!");
}
```

https://doc.rust-lang.org/std/thread/

More on spawning

- A *closure* is the function that the thread runs

|| indicates to Rust that this is a closure

Note: closures are sometimes used as parameters to a function (ex: a custom 'sum' method), or defined and stored in a variable (you can store a closure object in a variable to be invoked later). In these cases, the || will indicate general, required parameters to the closure. If this doesn't make sense, don't worry about it! You won't need it for this class.

More on spawning

- A *closure* is the function that the thread runs
- Like lambdas in C++, closures can "capture" their environments. This means that a closure can use a variable that was defined outside of it.
  - Unlike in C++, capturing happens implicitly: if you use a variable, Rust will assume you want to capture it.

Example: `i` is initialized in the main thread, then used in the closure by the new, spawned thread.

```
let i = 10;
threads.push(thread::spawn(|| {
    println!("Hello from {}!", i);
}));
```

More on spawning

- When a closure captures a value, that value is **borrowed** (immutably) by default.
- That can lead to errors like this (from last class)
- why?
  - **i** is owned by the main thread. It has some *lifetime* in that thread.
  - **i** is borrowed by the spawned thread.
  - The Rust compiler can't guarantee that's ok: what if **i** is dropped — goes out of scope — in the main thread before the spawned thread is done using it? That would lead to the spawned thread using a reference that no longer refers to valid memory!

```
error[E0373]: closure may outlive the current function, but it borrows `i`, which is owned by the current function
  --> src/main.rs:11:36
   |
11 |         threads.push(thread::spawn(|| {
   |                                    ^^ may outlive borrowed value `i`
12 |             println!("Hello from extrovert {}!", NAMES[i]);
   |                                                        - `i` is borrowed here
```

More on spawning

- To force a closure to take ownership, we use the **move** keyword.
- \`Move\` acts like the \`=\` operator:
  - If the value's type is copy (primitive types like ints, booleans, etc.), a new variable is created with a copy of the value.
  - Otherwise, the closure takes ownership of the value, and it can no longer be accessed in the original thread.
    - *For a refresher on copy vs. move, see these **notes**.*

```
let mut threads = Vec::new();
for i in 0..6 {
    threads.push(thread::spawn(move || {
        println!("Hello from extrovert {}!", NAMES[i]);
    }));
}
```

In this example from last time, \`i\` is an integer, which is copy, so \`move\` gives each spawned thread gets its own copy of \`i\`.

If \`i\` were a string, a vector, or some other type that isn't copy, this code wouldn't compile. Why?

More on spawning

- To force a closure to take ownership, we use the **move** keyword.
- `Move` acts like the `=` operator:
  - If the value's type is copy (primitive types like ints, booleans, etc.), a new variable is created with a copy of the value.
  - Otherwise, the closure takes ownership of the value, and it can no longer be accessed in the original thread.
    - *For a refresher on copy vs. move, see these **notes**.*

```
let mut threads = Vec::new();
for i in 0..6 {
    threads.push(thread::spawn(move || {
        println!("Hello from extrovert {}!", NAMES[i]);
    }));
}
```

In this example from last time, `i` is an integer, which is copy, so `move` gives each spawned thread gets its own copy of `i`.

If `i` were a string, a vector, or some other type that isn't copy, this code wouldn't compile. Why?

What if we want to share data?

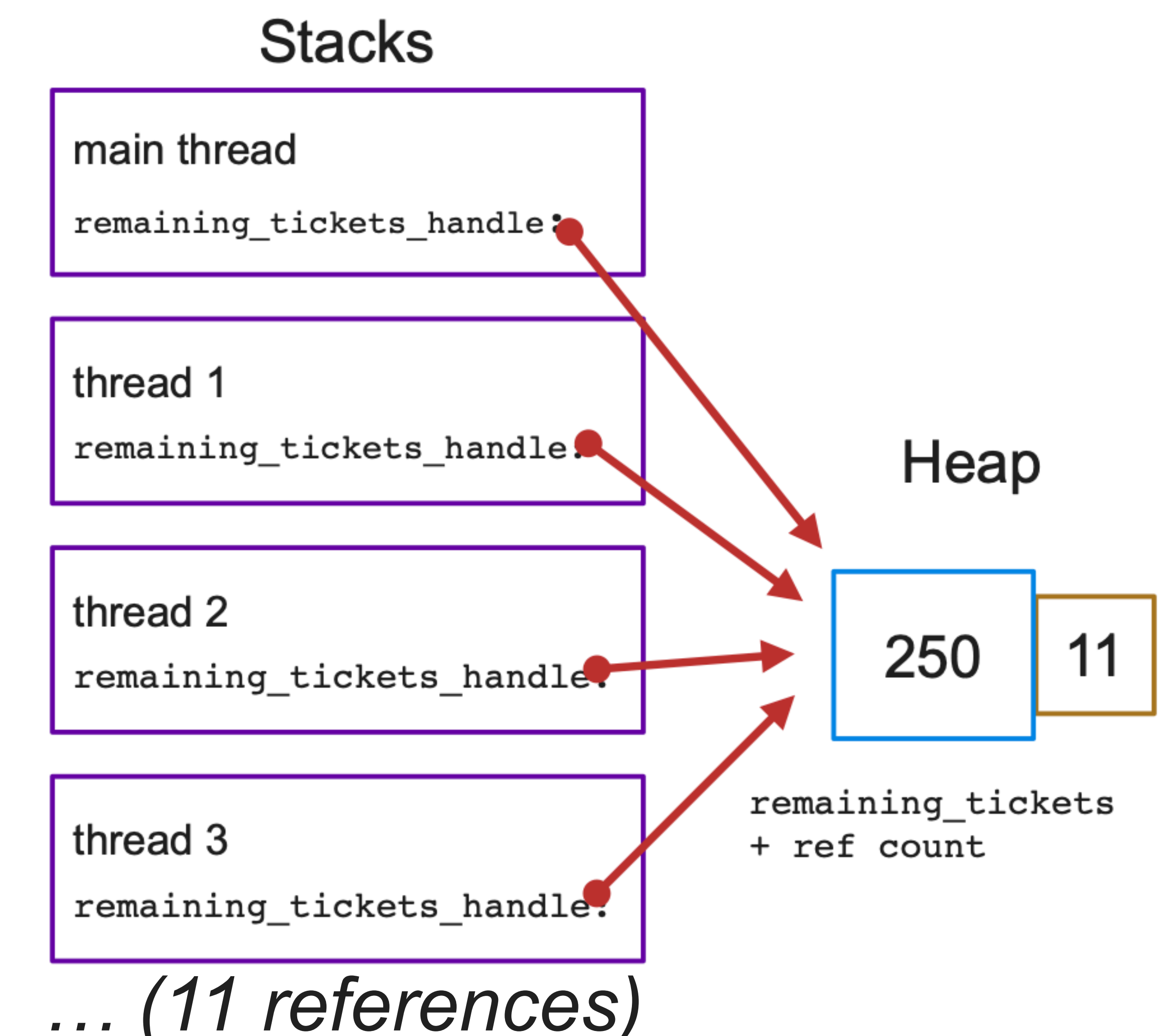
- Let's start with immutable data
- The problem: lifetimes
  - We've talked about how every value in Rust can only have one *owner.*
  - If we want to share data across threads, who should own the value?
  - Say that thread A owns X and thread B borrows X. What happens if A exits before B does? (This is totally possible!)

What if we want to share data?

- Solution: the Arc (atomically reference counted) type
- A way to “share ownership” across threads
- Arc::new: puts the value (here, an integer, 250) on the heap and gives it a reference count (initially, 1). The returned value is an Arc that can be “dereferenced” like a pointer to the memory.
- Arc::clone increments the reference count and returns a new “pointer” (Arc) to the same memory
- Dropping an Arc decrements the reference count. If the reference count is 0, frees the memory.

```
let remaining_tickets = Arc::new(250);

let mut threads = Vec::new();
for i in 0..10 {
    let remaining_tickets_handle = remaining_tickets.clone();
    threads.push(thread::spawn(move || {
        ticket_agent(i, remaining_tickets_handle)
    }));
}
```



What if we want to share data?

- The reference count stays in sync with the number of active references to it
- Memory is freed by the last thread using it
- A way to “share ownership” across threads while (largely) guaranteeing that:
  - Every value will get cleaned up (prevent memory leaks)
  - Every value will *only* get cleaned up once *all* threads are done with it.

What if we want to share and modify data?

- Mutex!
- In Rust, data goes *inside* the mutex

```
let remaining_tickets: Arc<Mutex<usize>>
    = Arc::new(Mutex::new(250));
```

Arc allows us to share the mutex across threads without worrying about lifetime/use-after-free
Mutex helps us make sure that only one thread can access it at the same time.

What if we want to share and modify data?

- You need to lock the mutex before accessing the data inside
- Locking a mutex returns a Result with (on success) a MutexGuard:
  - You can use this like a pointer to access the data.
  - Throughout the lifetime of the MutexGuard — until it's dropped — the mutex is locked.
  - Dropping the MutexGuard unlocks the mutex.
  - Dropping can happen implicitly (e.g., variable goes out of scope) or explicitly (call to `drop`).

```
let mut remaining_tickets_ref =
    remaining_tickets.lock().unwrap();
if *remaining_tickets_ref == 0 {
    break;
}
handle_call();
*remaining_tickets_ref -= 1;
println!("Agent #{} sold a ticket! ({}
    id, *remaining_tickets_ref);
drop(remaining_tickets_ref);
if should_take_break() {
    take_break();
}
```

What if we want to share and modify data?

- Note: locking a mutex will fail if the mutex has been “poisoned” — if a thread panicked while holding a lock on the mutex.
- In this case, Rust assumes that the encapsulated data may be corrupted, and you probably shouldn’t access it.
- In this class, you can just `unwrap` the call to `lock`.

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

Safer Multithreading

- Safe Rust does not prevent all race conditions, but it does prevent *data races*
  - *"Multiple threads access a value, where at least one of them is writing"*
  - Significantly: it prevents dangling pointers/use-after-free errors that can emerge from multithreading.
- What else can go wrong in Rust with multithreading?
  - Dropping a lock in the wrong place
    - Inadvertently serializing your code
    - Race conditions / unexpected behavior due to interactions across multiple lines of code
  - Deadlock, livelock, and starvation
    - *Cool project in development: "Avoiding Rust Deadlocks via Visualizing Lifetime" — an IDE tool to highlight critical sections + help people avoid deadlock!*

More recommended reading

- Why threads are a bad idea (for most purposes)
- Understanding Memory and Thread Safety Practices and Issues in Real-World Rust Programs
- Search for “race condition” on CVE (publicly-disclosed vulnerabilities) database
    - Search for “rust race condition”
- Data Races and Race Conditions (from the Rust book)
- Optional end-of-quarter 110L content: channels and event-driven programming

Notes

- Rust also has implementations for *semaphores* and *condition variables*.
  - Sema: one implementation of a semaphore (there isn't one in the standard library)
  - CondVar: this is in the standard library
- You shouldn't need them in this class, and they're not that different from CS110 material, so I'm not covering them directly.
  - *There are multithreading examples from 2020 that use both; check those out if you're interested, and I'm happy to answer questions!*

Let’s practice!

Link Explorer

- You and your friends are bored, so you decided to play a game (as one does) where you go to a random Wikipedia page and try to find a link to another wikipedia page that is the longest (by length of the html)
- You decide to enlist Rust (along with the reqwest and select crates) to help you.

[code]
see lecture notes :)