CIS 1905

Lecture 07: Intro to Concurrency

03.12.2024

Overview

Fill out the Office Hours poll!

pollev.com/cis1905rust915

Review: Smart Pointers

Box<T>

Single ownership on the heap

Cow<'a, B>

Lazy copies

Rc<T>

Multiple ownership (without mutation)

Vec<T>/ …

Dynamically sized collections

Rc<RefCell<T>>/ …

Multiple ownership (with mutation)

&[T], &dyn T

Wide pointers

Multithreaded Smart Pointers

```
Arc<T>
Arc<Mutex<T>>
Arc<RwLock<T>>
```

Concurrency

01

Threads
Concurrency

02

Message Passing
Channel sends/receives messages between threads

03

Shared State
Threads accessing shared data

Threads

In Rust

What is a Thread?

- A thread is a worker that will do work for you in the background
- Threads are independent parts of your program that run concurrently

Spawning threads

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

fn main() {
    let handle = thread::spawn(|| {
        for i in 1..10 {
            println!("{}, spawned thread", i);
            thread::sleep(Duration::from_millis(1));
        }
    }); // Returns JoinHandle

    for i in 1..5 {
        println!("{}, main thread", i);
        thread::sleep(Duration::from_millis(1));
    }

    handle.join().unwrap(); // Waits for thread(s) to finish
}
```

Spawning threads

What's the output?

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

fn main() {
    let handle = thread::spawn(|| {
        for i in 1..10 {
            println!("{}, spawned thread", i);
            thread::sleep(Duration::from_millis(1));
        }
    }); // Returns JoinHandle

    for i in 1..5 {
        println!("{}, main thread", i);
        thread::sleep(Duration::from_millis(1));
    }

    handle.join().unwrap(); // Waits for thread(s) to finish
}
```

Spawning threads

When does handle.join() return Error?

```
use std::thread
use std::time::

fn main() {
    let handle = thread::spawn(|| {
        for i in 1..10 {
            println!("{}, spawned thread", i);
            thread::sleep(Duration::from_millis(1));
        }
    }); // Returns JoinHandle

    for i in 1..5 {
        println!("{}, main thread", i);
        thread::sleep(Duration::from_millis(1));
    }

    handle.join().unwrap(); // Waits for thread(s) to finish
}
```

Spawning threads

When does handle.join() return Error?

If thread panics

```
use std::thread
use std::time::
fn main() {
    let handle = thread::spawn(|| {
        for i in 1..10 {
            println!("{}, spawned thread", i);
            thread::sleep(Duration::from_millis(1));
        }
    }); // Returns JoinHandle

    for i in 1..5 {
        println!("{}, main thread", i);
        thread::sleep(Duration::from_millis(1));
    }

    handle.join().unwrap(); // Waits for thread(s) to finish
}
```

Spawning threads

What happens here?

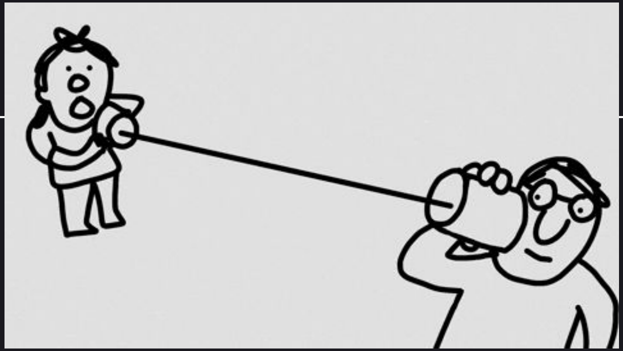
```rust
use std::thread;
use std::time::Duration;

fn main() {
    let handle = thread::spawn(|| {
        for i in 1..10 {
            println!("{}, spawned thread", i);
            thread::sleep(Duration::from_millis(1));
        }
    }); // Returns JoinHandle
    handle.join().unwrap(); // Moved, waits for thread before loop

    for i in 1..5 {
        println!("{}, main thread", i);
        thread::sleep(Duration::from_millis(1));
    }
}
```

Rayon Crate

- Data Parallelism library
- Abstracts away explicit thread management!
  - Map Reduce
  - Sort a vector
  - Testing any or all matches

Got an `.iter()`? Swap it with `.par_iter()`

```
use rayon::prelude::*;
fn main() {
    let mut arr = [0, 7, 9, 11];
    arr.par_iter_mut().for_each(|p| *p -= 1);
    println!("{:?}", arr);
}
```



Message Passing

In Rust

Message Passing Concurrency

- Exchanging information between threads
- **Channel**: a general programming concept by which data is sent from one thread to another.
    - Sender/Transmitter (TX)
    - Receiver (RX)

Sender
`send` - Returns result. Error if receiver has been dropped

Receiver
`recv` - Blocks execution until a message is received
`try_recv` - Return none if nothing is available

Example

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

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);
}
```

MPSC Channel

MPSC (Multiple Producer, Single Consumer), Unbounded

- You can clone the sender

```
use std::sync::mpsc;

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

    tx.send(10).unwrap();

    println!("Received: {:?}", rx.recv());

    let tx2 = tx.clone();
    tx2.send(30).unwrap();
    println!("Received: {:?}", rx.recv());
}
```

MPSC Sync Channel

MPSC (Multiple Producer, Single Consumer), Bounded (Synchronous)

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

let (sender, receiver) = mpsc::sync_channel(1);

sender.send(1).unwrap(); // this returns immediately

thread::spawn(move|| {
    // this will block until the previous message has been received
    sender.send(2).unwrap();
});
```

Tokio Crate

Channels for nonblocking, async Rust

- `mpsc` - Similar to Standard Library
- `oneshot` - single-producer, single consumer channel. A single value can be sent.
- `broadcast` - multi-producer, multi-consumer. Many values can be sent. Each receiver sees every value.
- `watch` - single-producer, multi-consumer. Many values can be sent, but no history is kept. Receivers only see the most recent value.

Summary of Channels

Channels are a programming concept where you send data across threads (messages). There is a sender (TX) and a receiver (RX).

Different Rust channels have different underlying implementation depending the type of communication you want.

Shared State

Arc<Mutex<...>>

How do we manage shared information across entities?

- Databases
- Distributed Systems
- What we learned from Rust ownership

Only one element can “write” it at a time

Mutex

Mutex means mutual exclusion, which means "only one at a time".

You call `.lock()` to obtain a MutexGuard that ensures only you have access to the value. Then, when that guard goes out of scope or is explicitly dropped with `std::mem::drop(var_name)`, it's no longer locked.

You can update the internal value using `*` symbol:

```
*mutex_guard = 6
```

Mutex

Mutex means mutual exclusion, which means "only one at a time".

```
use std::sync::Mutex;
fn main() {
    let my_mutex = Mutex::new(5); // A new Mutex<i32>. We don't need to say mut
    let mut mutex_changer = my_mutex.lock().unwrap();
                                 // mutex_changer is a MutexGuard
                                 // Now it has access to the Mutex

    println!("{:?}", my_mutex); // This prints "Mutex { data: <locked> }"
                                // So we can't access the data with my_mutex
                                // now, only with mutex_changer
    println!("{:?}", mutex_changer); // This prints 5.

    *mutex_changer = 6; // mutex_changer is a MutexGuard<i32>

    println!("{:?}", mutex_changer); // Now it says 6
}
```

Note that there's no concurrency in this example!

Mutex

What happens when you call lock on an already locked value?

Mutex

What happens when you call lock on an already locked value?

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

fn main() {
    let my_mutex = Mutex::new(5);
    let mut mutex_changer = my_mutex.lock().unwrap();
    let mut other_mutex_changer = my_mutex.lock().unwrap();
    println!("This will never print...");
}
```

Arc‹Mutex‹…››

How do we use Mutex in an async context?

First, why can't we directly use Mutex as is?

- Directly using Mutex violates rust ownership rules

```
use std::sync::{Arc, Mutex};
use std::thread;
fn main() {
    let counter = Arc::new(Mutex::new(0));
    let mut handles = vec![];
    for _ in 0..10 {
        let counter = Arc::clone(&counter);
        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());
}
```

Arc<Mutex<..>>

pollev.com/cis1905rust915 async / .await

Rust Asynchronous Book

https://rust-lang.github.io/async-book/

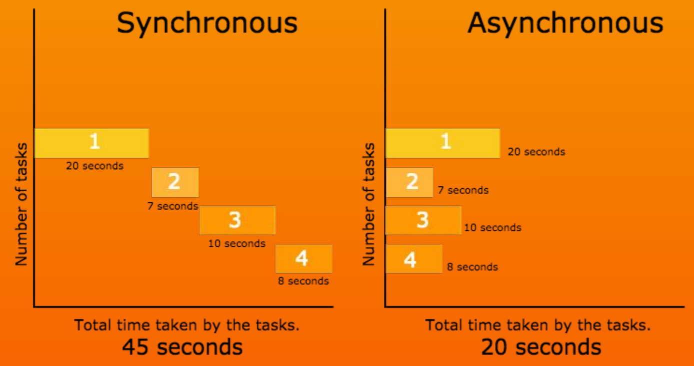
Why async?

Advantages

- Threads are nice, but they come with a performance overhead
- Async uses “virtual threads” (not “real” OS threads)
- Dedicated syntax
- *Feels* like synchronous programming

Disadvantages

- It is usually simpler to use actual threads



Don’t waste time waiting!

What is a **Future**?

```rust
pub trait Future {
    type Output;

    // Required method
    fn poll(self: Pin<&mut Self>, cx: &mut Context<'_>) -> Poll;
}
```

- *"A future is a value that might not have finished computing yet"*
- When a `fn` returns a `Future<Output = T>`, you don't actually *get* the `T`, instead you get a *promise* of some future value
- The future doesn't do any **work** until you `.poll()` it (futures are **lazy**)
  - The future will either by `Poll::Ready(result)` or `Poll::Pending`

async / .await = syntax sugar

async

Marks a function or scope as asynchronous.
The return type is a lie!

For example, this function:

```
async fn inc(x: i32) -> i32
```

…is equivalent to:

```
fn inc(x: i32) -> impl Future<Output = i32>
```

.await

Polls the future.

- If it is ready, then extract the T
- Otherwise, early return `Poll::Pending`

For example, this statement inside an async function:

```
let x = inc(5).await;
```

…is equivalent to:

(it's complicated)

Desugaring async

```rust
async fn patrol(unit: UnitRef, poses: [i32; 2]) {
    loop {
        goto(unit.clone(), poses[0]).await;
        goto(unit.clone(), poses[1]).await;
    }
}
```

Becomes

```rust
fn patrol(unit: UnitRef, pos: [i32; 2]) -> impl Future<Output = ()> {
    PatrolFuture::Start(unit.clone(), pos)
}
```

https://www.eventhelix.com/rust/rust-to-assembly-async-await-nested/

Desugaring async

```
async fn patrol(unit: UnitRef, poses: [i32; 2]) {
    loop {
        goto(unit.clone(), poses[0]).await;
        goto(unit.clone(), poses[1]).await;
    }
}
```

Becomes

```
fn patrol(unit: UnitRef, pos: [i32; 2]) -> impl Future<Output = ()> {
    PatrolFuture::Start(unit.clone(), pos)
}
```

What is PatrolFuture?

https://www.eventhelix.com/rust/rust-to-assembly-async-await-nested/

Async functions are *state machines*

```rust
// The state machine enum
enum PatrolFuture {
    // Initial state
    Start(UnitRef, [i32; 2]),
    // Waiting for goto to complete
    WaitingToReachPosition0(UnitRef, [i32; 2], impl Future<Output = ()>),
    WaitingToReachPosition1(UnitRef, [i32; 2], impl Future<Output = ()>),
    // Final state (which is unreachable)
    Done,
}
```

https://www.eventhelix.com/rust/rust-to-assembly-async-await-nested/

```rust
impl Future for PatrolFuture {
    type Output = ();

    fn poll(mut self: Pin<&mut Self>, cx: &mut Context<'_>) -> Poll<Self::Output> {
        loop {
            match *self {
                // In the start state, create a goto future and move to the waiting state
                PatrolFuture::Start(unit, poses) => {
                    let fut = goto(unit.clone(), poses[0]);
                    *self = PatrolFuture::WaitingToReachPosition0(unit, poses, fut);
                }
                // In the waiting state for goto1, poll the goto future and move to the next waiting state if it's ready
                PatrolFuture::WaitingToReachPosition0(unit, poses, ref mut fut) => {
                    match Pin::new(fut).poll(cx) {
                        Poll::Ready(()) => {
                            let fut = goto(unit.clone(), poses[1]);
                            *self = PatrolFuture::WaitingToReachPosition1(unit, poses, fut);
                        }
                        Poll::Pending => return Poll::Pending,
                    }
                }
                // In the waiting state for goto2, poll the goto future and move back to the start state if it's ready
                PatrolFuture::WaitingToReachPosition1(unit, poses, ref mut fut) => {
                    match Pin::new(fut).poll(cx) {
                        Poll::Ready(()) => *self = PatrolFuture::Start(unit.clone(), poses),
                        Poll::Pending => return Poll::Pending,
                    }
                }
                // In the done state (which is unreachable), return ready
                PatrolFuture::Done => return Poll::Ready(()),
            }
        }
    }
}
```

https://www.eventhelix.com/rust/rust-to-assembly-async-await-nested/

`async` / `.await` summary

If you are inside an `async` function or `async` block, you unlock the power to use `.await`

It behaves like a normal function, but the return type is `Future<Output = T>`

As soon as you `.await` something, one of two things will happen:

1. The future is ready, and you get the value
2. The future is *not* ready, and you early return `Poll::Pending`

Your "function" is actually a state machine enum that implements `Future`, and can itself be `.await`ed

The `async` **executor** decides when to poll your futures again run your code

Summary of everything

Diagram time

- Thread
- Channel
- Arc<Mutex<..>>
- Tokio / Rayon

thE midDle

The “End” (Part 2, continued next Tuesday)