Concurrency

Rust, In Theory and in Practice

CAS CS 392 M1

Outline

The Punchline: Rust's type/borrow system catches many common concurrency bugs at compile time

Today we'll talk about:

- Creating threads to run code at the same time
- Passing messages between threads
- Sharing state across threads

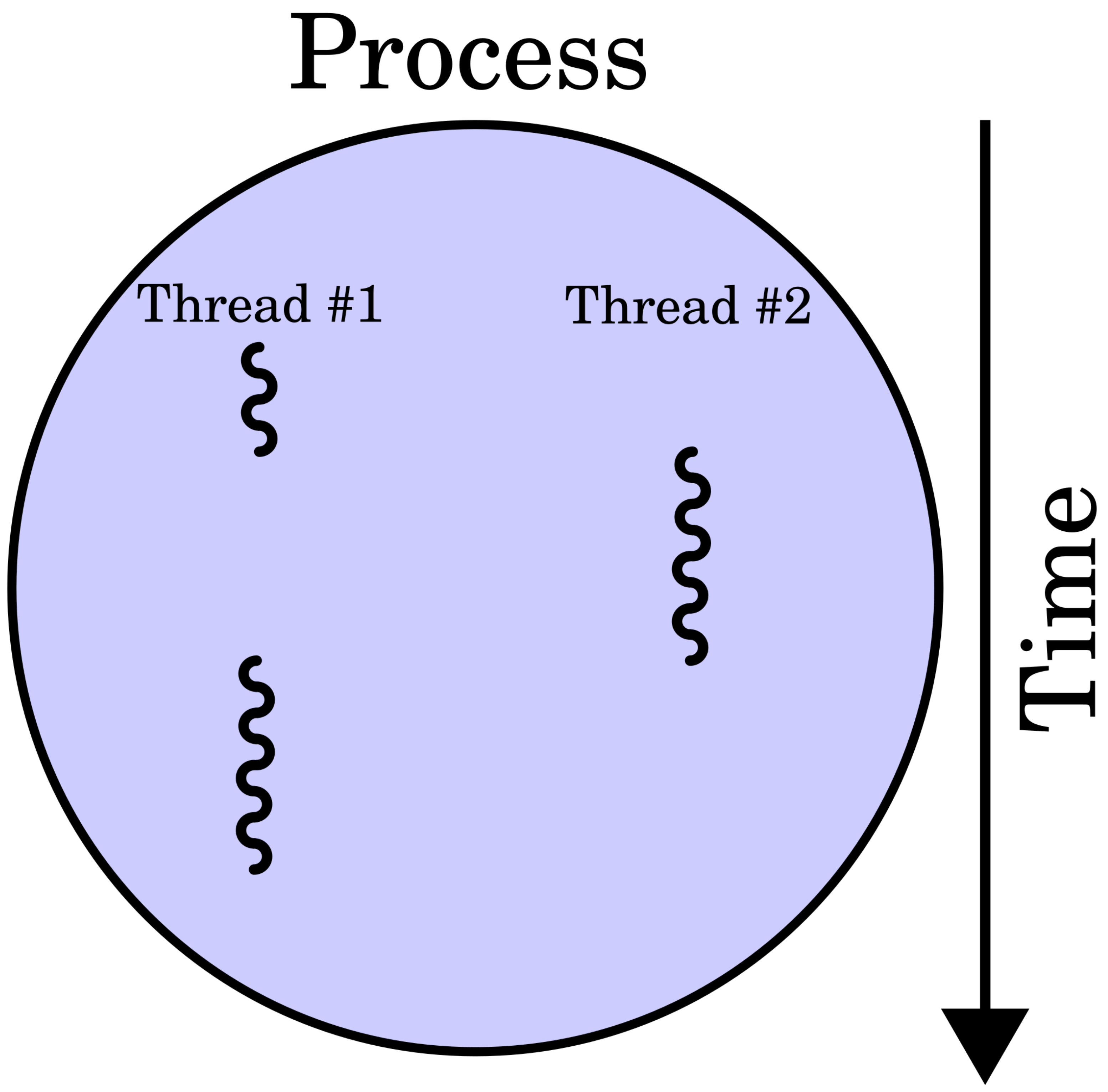
Threads

Processes and Threads

Operating systems run a programs in a **process**

A process can have parts that are run independently using **threads**

Typically an OS exposes an API to spawn threads within a process



https://commons.wikimedia.org/wiki/File:Multithreaded_process.svg

The Challenge

Running multiple tasks at the same time can be great for efficiency, but it introduces *complexity*

There are many bugs that can occur due to interleaved threads or inconsistent access order, i.e., *race conditions*

Note: Safe Rust ensures no **data races** but does not ensure general race condition safety (e.g., deadlocks are "safe")

Data Race

A **data race** occurs when:

- » Multiple threads are accessing the same data
- » At least one is mutating
- » There is no mechanism for synchronization

sequential

$$\{ y \mapsto 1 \}$$

$$\downarrow$$

$$
\begin{aligned}
x1 &\leftarrow y \\
x1 &\leftarrow x1 + 1 \\
y &\leftarrow x1 \\
x2 &\leftarrow y \\
x2 &\leftarrow x2 + 1 \\
y &\leftarrow x2
\end{aligned}
$$

$$\downarrow$$

$$\{ y \mapsto 3 \}$$

concurrent

$$\{ y \mapsto 1 \}$$

$$\downarrow$$

$$
\begin{aligned}
x1 &\leftarrow y \\
x1 &\leftarrow x1 + 1 \\
x2 &\leftarrow y \\
y &\leftarrow x1 \\
x2 &\leftarrow x2 + 1 \\
y &\leftarrow x2
\end{aligned}
$$

$$\downarrow$$

$$\{ y \mapsto 2 \}$$

Deadlock (A Picture)

Deadlock occurs when two threads are waiting on each other and the process hangs

sequential

$\{\ y \mapsto 1,\ z \mapsto 1\ \}$

```
x1 ← y.lock()
*x1 ← 2
w1 ← z.lock()
*w1 ← 2
x1.unlock()
w1.unlock()
x2 ← z.lock()
*x2 ← 3
w2 ← y.lock()
*w2 ← 3
x2.unlock()
w2.unlock()
```

$\{\ y \mapsto 3,\ z \mapsto 3\ \}$

concurrent

$\{\ y \mapsto 1,\ z \mapsto 1\ \}$

```
x1 ← y.lock()
*x1 ← 2
x2 ← z.lock()
*x2 ← 3
w1 ← z.lock()
w2 ← y.lock()     ← stuck...
*w1 ← 2
x1.unlock()
w1.unlock()
*w2 ← 3
x2.unlock()
w2.unlock()
```

Thread Model

Rust uses a **1:1 model** for threads, one user thread per one OS thread. There's also:

» **Many:1** (green threads) has many user threads for a single OS thread

» **Many:Many** has many user threads to a pool of OS threads

Spawning Threads

```rust
thread::spawn(|| {
    for i in 0..100 {
        println!("{i}")

    }
});
```

thread::spawn takes a closure, which define what the thread should do

Important. Spawning a thread does not guarantee that the corresponding computation will finish

The main thread (in which the new thread is spawn) may finish and drop any unfinished computation

Joining Threads

```
let handle = thread::spawn(|| {
    for i in 0..100 {
        println!("{i}")

    }
});
let _ = handle.join();
```

We can "wait" for a spawned thread to finish using **.join()**

Joining *blocks* the main thread until the joined thread is done

Note. joining takes ownership of the handle (we can't, for example, extract the underlying thread after we've joined)

Move Closures

```
let v = vec![1, 2, 3];

let handle1 = thread::spawn(move || {
    println!("{}", v[0]);
});
```

We often need to *move* data into closures when working with threads (we need to make sure data doesn't get dropped before the thread is done)

Since closures infer how much borrowing needs to be down we often need the **move** keyword to force closures to take ownership of the values it uses

Type of Spawning

```
pub fn spawn<F, T>(f: F) -> JoinHandle<T>
where
    F: FnOnce() -> T + Send + 'static,
    T: Send + 'static,
```

The lifetime bound on **F** ensures that we can't borrow things that are stack allocated by the main thread

This necessitates **move** in most cases (even with joins)

Message Passing

High-Level

```rust
let (tx, rx) = std::sync::mpsc::channel();
```

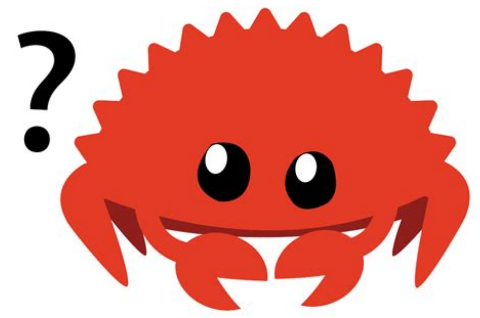
"Do not communicate by sharing memory; instead, share memory by communicating."

In Rust, we can create *multi-producer single-consumer channels* for passing messages between threads

demo

(simple example)

Message passing and Ownership




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

Sending a message *transfers ownership*

The type system can expression that a message should not be used after being sent

Shared-State Concurrency

High-Level

```
let counter = Arc::new(Mutex::new(0));

let counter = Arc::clone(&counter);
let handle = thread::spawn(move || {
    let mut num = counter.lock().unwrap();
    *num += 1;
});
```

- Sometimes we do want shared-state concurrency. We can do this with **Arc** (atomic reference counting)
- If we want *mutable* shared-state, we can use an "internal mutability" pattern with **Mutex**

Comparison with Rc and RefCell

Rc<T> is to **Arc<T>** as **RefCell<T>** is to **Mutex<T>**

RefCell and **Mutex** both allow for "internal mutability"

Rc + RefCell leads to memory leaks and **Arc + Mutex leads** to deadlocks

The Takeaways

The compelling part of concurrency in Rust is not that it handles concurrency better than in other languages, but that *the concerns of concurrency fits into the ownership paradigm very well*

When we pass a value as a message, we shouldn't be able to work with it anymore. That can be represented as transferring ownership once the value is sent

We should be careful and explicit when sharing data across threads, that's built into the way we use Rust

The Takeaways

Concurrency is *hard*

What we've shown is the thread-level interface exposed by Rust. In reality, you probably wouldn't use this unless you *really* needed control

Many folks have thought about this problem, and have built nice libraries, e.g., **rayon** is a popular crate for parallel iterators

Workshop

The Tasks

- Write a function **parallel_sum** that takes the sum of the elements in a vector by breaking the vector into **k** chunks and creating a thread for each chunk to sum. Benchmark this against the sequential sum and sum with rayon
- Work on assignment 6. If you finish extend with thread pool according to the tutorial given in the book
- Write a safe Rust program with deadlock (For an additional challenge, write an unsafe Rust program with a data race)