Lecture 12

Intro to threads

What we will cover today

Some OS concepts

- Processes and threads

Rust

- Spawning and Joining threads in Rust
- Using the move keyword in threads

Optional Reading:

The Rust Book Chapter 16.1 – Using Threads to Run Code Simultaneously

Operating Systems

Common OSes

- Windows
- MacOS
- Linux

Roles of the OS

- Resource management – E.g. CPU time, Main memory, Disks
- Abstraction – Work on different hardware

Processes and threads

What are processes?

- The abstraction of a single running program
- Multiple processes can be running the same program
- E.g. Firefox, Discord, Text editor, Rust program

What are threads?

- Multiple threads in one program
- Threads share the address space of the program
- E.g. In Discord, you can have
    - A thread for listening to new messages
    - A thread for processing keyboard presses and then displaying it on screen
    - A thread to draw out the different UI components

OSes and Processes/Threads

OS is responsible for scheduling processes and threads

- If we don’t do anything special, we cannot guarantee that threads are run in a certain order

Will make sense as we see how the different threads interleave

Writing threads are hard…

Since information can be shared between threads, we have to be careful to make sure that the information stays consistent

E.g.

- Deadlock: Thread A waits for Thread B, Thread B waits for Thread A. Program is now stuck
- Bugs are difficult to reproduce, which make them harder to find

A taste of race conditions!

Say you have a bank account with $100, and two friends decide to deposit $80 each into your bank account. (You expect to have $260 at the end)

$100

Friend A

Friend B

A terrible bank

They go to the ATM at the same time...

$100

How much does he have now?

How much does he have now?

Friend A

Friend B

A terrible bank

The bank tells the ATM that the balance is $100

$100

$100

$100

Friend A

Friend B

A terrible bank

Friend A's ATM calculates that the new balance is $180

Okay, his new balance is $180

~~$100~~ $180

Friend A

Friend B

A terrible bank

They go to the ATM at the same time...

~~$180~~ $180

Okay, his new balance is $180

Friend A

Friend B

A terrible bank

Bank steals your $80 :D

$180

Friend A

Friend B

Writing threads are hard…

Previous slides was an example of a race condition

Rust uses the ownership and type system to help prevent concurrency problems as well. This aspect of Rust is called fearless concurrency!

In this lecture, we will go through how to make new threads. In subsequent lectures, we will talk about different techniques to deal with concurrency problems.

Spawning new threads

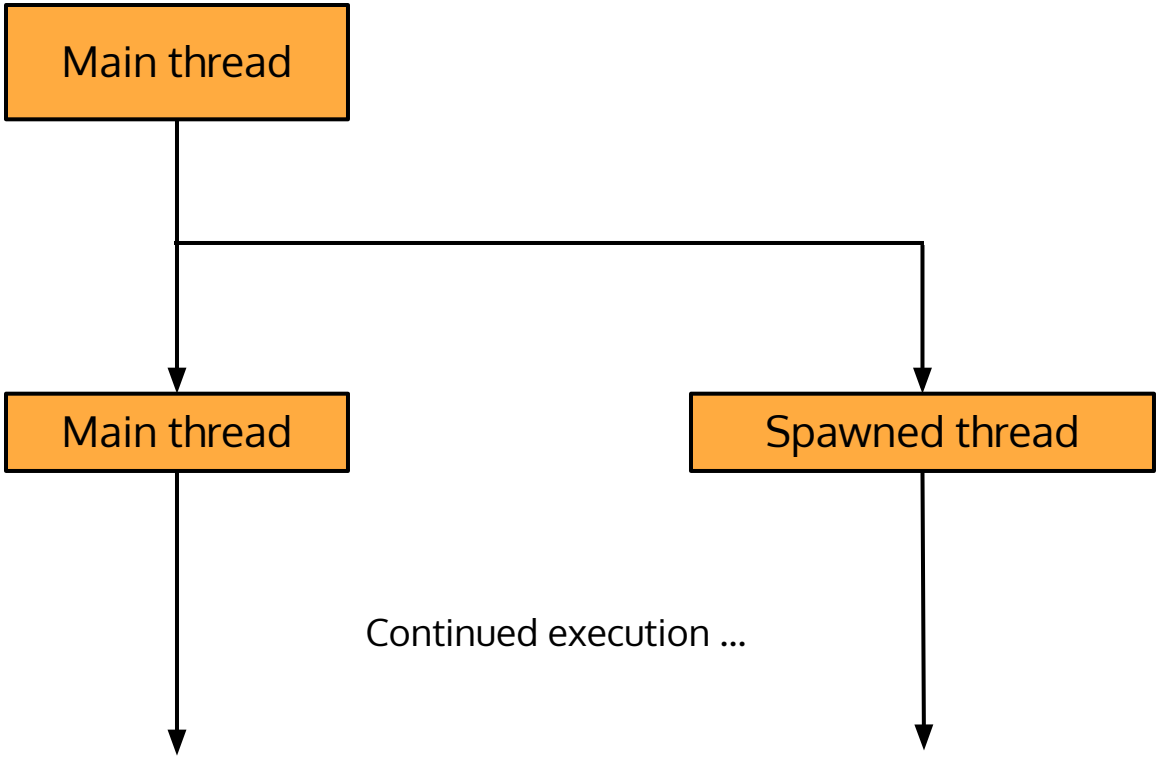
```
use std::thread;
use std::time::Duration;
...
thread::spawn(|| {
    for i in 1..10 {
        println!("hi number {i} from the spawned thread!");
        thread::sleep(Duration::from_millis(1));
    }
});
```

Import the thread library

Closure

Put thread to sleep for 1ms

Split into two “lanes” of execution



Demo!

Child thread doesn’t finish :(

In the demo, we noticed that the spawned thread never gets to finish

This is because the main threads finishes before the spawned thread. Since the parent thread terminates, all child threads terminate as well.

Solution: Make the parent wait for child thread to finish!

JoinHandle

```
let handle: JoinHandle<_> = thread::spawn(|| {
    ...
});
...
handle.join();
```

Wait for thread to terminate

thread::spawn returns a JoinHandle type

Calling handle.join() will cause the parent thread to block till the child thread finished

Demo!

Using data from main thread in spawned thread

```
let vector = vec![1,2,3,4,5];

let handle: JoinHandle<_> = thread::spawn(|| {
   println!("{:?}", vector);
});
```

Rust will try to borrow vector to use in the closure. However, the thread may outlive the function in which it is called! vector might not always be a valid reference

Example where vector becomes invalid

```rust
let vector = vec![1,2,3,4,5];

let handle: JoinHandle<_> = thread::spawn(|| {
    println!("{:?}", vector);
});

drop(vector);
```

Use the move keyword!

```
let vector = vec![1,2,3,4,5];

let handle: JoinHandle<_> = thread::spawn(move || {
    println!("{:?}", vector);
});

handle.join();
```

Use the move keyword

Move date into closure instead of borrowing it

Recap

Processes vs threads – A process can have multiple threads

Spawning threads – thread::spawn

Wait for child to finish – handle.join()

Ownership rules – Using move when passing data into closure

Announcements

HW10 released today on PrairieLearn

Due 1 week from now — Next Friday 03/14 23:59

Final project release – Form groups and think of project ideas