Lecture 13

Message passing with mpsc

Happy pi day! (3.14159265358979323846264338327…)

Credits: ChatGPT / DALL-E

“Please help me generate an image that integrates the Rust programming language with a pie”

What we will cover today

Topics

- Message passing vs Shared memory
- std::sync::mpsc

Optional Reading:

The Rust Book Chapter 16.2 – Using Message Passing to Transfer Data Between Threads

What we talked about last time

- Multithreading
- Spawning multiple threads
- Using join to wait for child threads to finish

We haven't discussed communication!

How do threads to talk to each other?

Talking between threads

Two main ways in Rust:

- Shared data
  - Different threads access the same copy of the data
  - Faster, but easier to create bugs due to synchronization
- Message passing
  - Communicate by sending messages to each other
  - Slower, but no need to worry about synchronization bugs

Talking between threads

Two main ways in Rust:

- Shared data
  - Different threads access the same copy of the data
  - Faster, but easier to create bugs due to synchronization
- Message passing
  - Communicate by sending messages to each other
  - Slower, but no need to worry about synchronization bugs

Message passing in Rust

Rust implements **channels** – Data sent from one thread to another

Can imagine the channel as a river with a flow direction, if you put a rubber ducky into the water, it will flow downstream

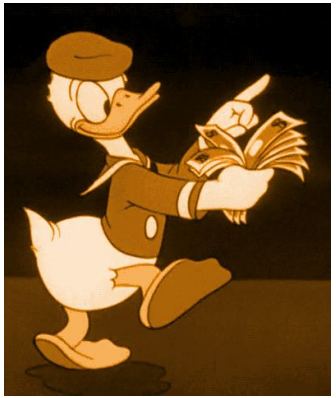
2 halves – Transmitter and receiver.
Transmitter puts the duck in, receiver picks up the duck

Message passing in Rust

Can image the channel as a river with a flow direction, if you put a rubber ducky into the water, it will flow downstream

Transmitter puts the duck in

receiver picks up the duck



std::sync::mpsc

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

fn main() {
    // tx = transmitter, rx = receiver
    let (tx, rx) = mpsc::channel();
}
```

mpsc = Multiple producer, single consumer

`mpsc::channel` returns a tuple of transmitter and receiver

tx.send()

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

thread::spawn(move || {
    let test = String::from("henlo");
    tx.send(test);
});
```

Ownership of the tx is moved into the thread!

`send<T>()` returns a `Result<(), SendError<T>>`

`Result` of the `tx.send()`

`send<T>()` returns a `Result<(), SendError<T>>`

Err means that data will ***never*** be received.

Ok ***does not mean*** that the data will be received

- Receiver side can close after the message is sent out but before receiving it

Let's look at how it's received!

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

... // Spawned some thread that sends data

rx.recv(); // Will block!
```

Receiver end will call `recv()`

Be careful! This call will block the thread!

Will only wake up if a message is received or if the sender is disconnected

`Result` of the `rx.recv()`

`recv<T>()` returns a `Result<T, RecvError>`

recv() will block the thread till *something* happens.

Something = Receive a message OR sender is disconnected

Err means the Sender side has disconnected

Ok returns the data we are waiting for

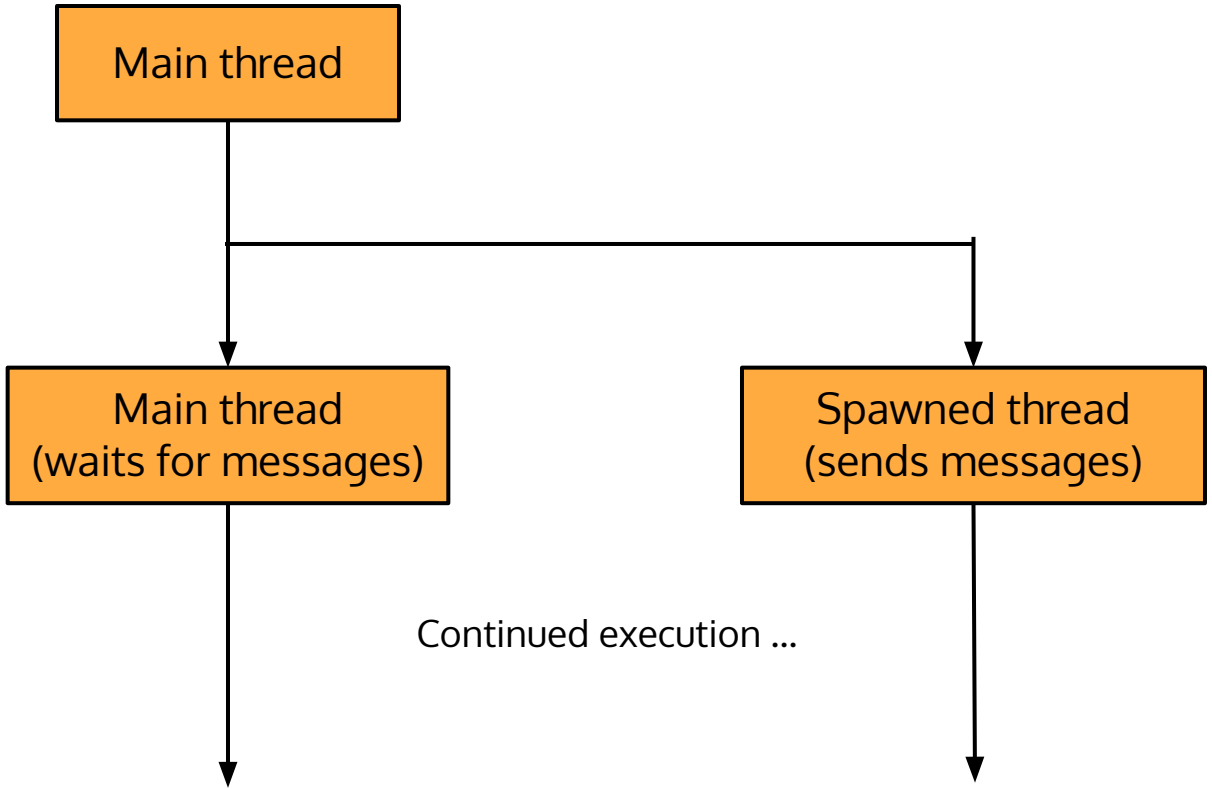
Put it all together...

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

thread::spawn(move || {
    let test = String::from("henlo");
    tx.send(test);
});

rx.recv();
```

Split into two “lanes” of execution



Demo!

What to take note of…

Ownership of message gets transferred from sender to receiver.

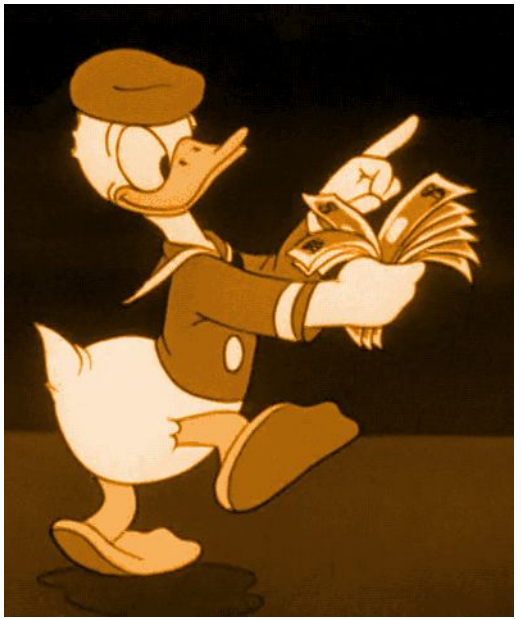
- Cannot use an object after sending it!

Wait… Isn't it called *multiple producers*, single consumer?

We can create multiple senders! They will all send to the same receiver





Multiple senders!

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

for i in 0..5{
    let tx_clone = tx.clone();
    thread::spawn(move ||{
        let message = String::from("Hello from thread") +
                        i.to_string().as_str();
        tx_clone.send(message);
    });
}

for message in rx{ println!("{}", message); }
```

Demo!

Announcements

MP3 released today on PrairieLearn
Due 3 weeks from now — 04/05 23:59

Remember to form project groups, it's due today (today = 03/14)!
Get the link in the Discord or the email I sent out.