Threads & Messaging in Rust

Lecture 12

Slides: Neil Kaushikkar

Goals For Today

- Review thread creation
- Motivation behind message passing
- Basic message passing
- Advanced message passing
- Hopefully a very example heavy lecture

Reminders

- HW9 due 10/11 at 11:59pm
- HW10 releasing today, due 10/13 at 11:50pm
- MP2 due 10/11 at 11:59

Spawning a Single Thread


```rust
use std::thread;

fn main() {
    let handle = thread::spawn(|| {
        println!("Hello from another thread!");
    });

    // Wait for our thread to finish
    match handle.join() {
        Ok(_) => (), // do nothing if our thread finished successfully
        Err(e) => println!("Thread join error {:?}", e)
    }

    println!("Hello from the main thread!");
}
```

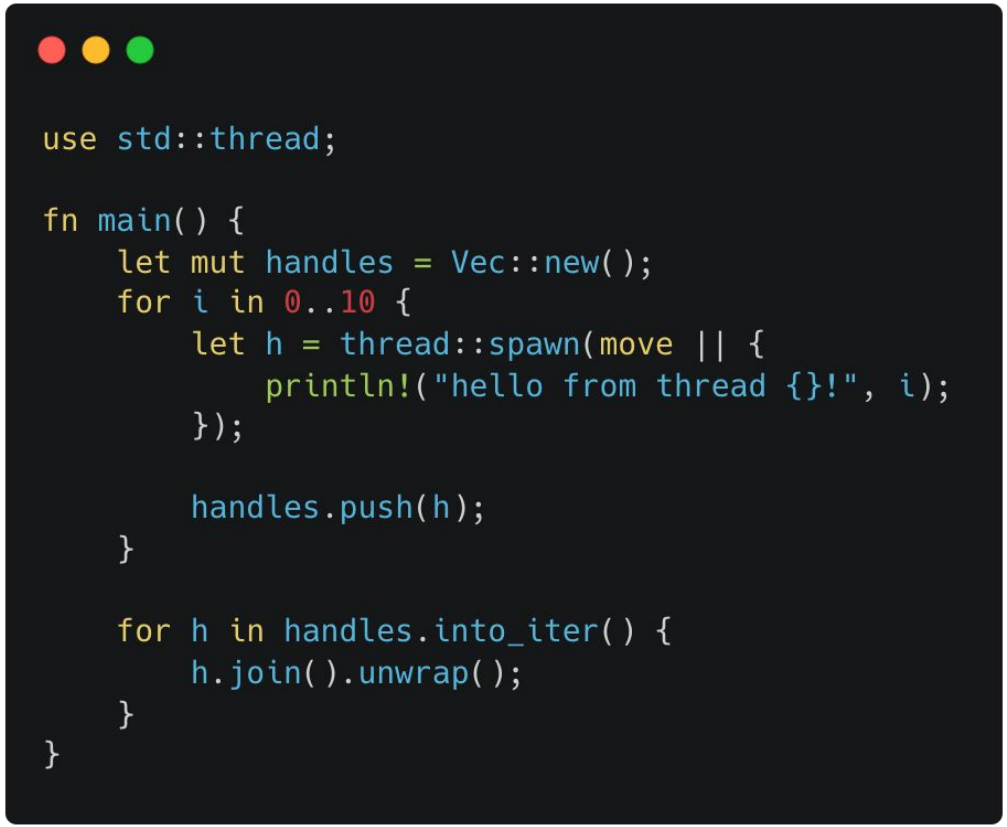
Joining Threads & Blocking Functions

- Blocking Functions: the current thread will stop until the function returns
- **JoinHandle**: A struct that gives you permission to join on a thread
  - Block the program until the thread terminates
- **thread::spawn()**: Spawns a single thread and returns a **JoinHandle** for it
- **join_handle.join()**:
  - Block the current thread until thread associated with the handle finishes
  - If the associated thread panics, **Err** is returned (otherwise **Ok** is returned)

Reference:

- https://doc.rust-lang.org/stable/std/thread/struct.JoinHandle.html
- https://doc.rust-lang.org/std/thread/fn.spawn.html

Spawning Multiple Threads + Moving Primitives



```rust
use std::thread;

fn main() {
    let mut handles = Vec::new();
    for i in 0..10 {
        let h = thread::spawn(move || {
            println!("hello from thread {}!", i);
        });

        handles.push(h);
    }

    for h in handles.into_iter() {
        h.join().unwrap();
    }
}
```

Hint:

- Use the **move** keyword to move data from the original thread to a new thread

Spawning Threads + Cloning Non-Primitives

```rust
use std::thread;

fn main() {
    let mut handles = Vec::new();
    let data = String::from("Here's a random message");

    for i in 0..10 {
        let data_clone = data.clone();

        let h = thread::spawn(move || {
            let msg = format!("{} -- from thread {}", data_clone, i);
            println!("{msg}");
        });

        handles.push(h);
    }

    for h in handles.into_iter() {
        h.join().unwrap();
    }
}
```

Hint:

- Use the **move** keyword to move data from the original thread to a new thread

Joining Threads & Blocking Functions

- Use the **move** keyword to move data from the original thread to a new thread
- The **move** keyword means:
  - Either Transfer ownership of data to the new thread
  - Or copy (NOT Clone) any data that can be copied into the new thread
    - i.e. primitive types
- Remember to call **.clone()** on data types that cannot be copied
  - Pass the clone of the original data into the thread

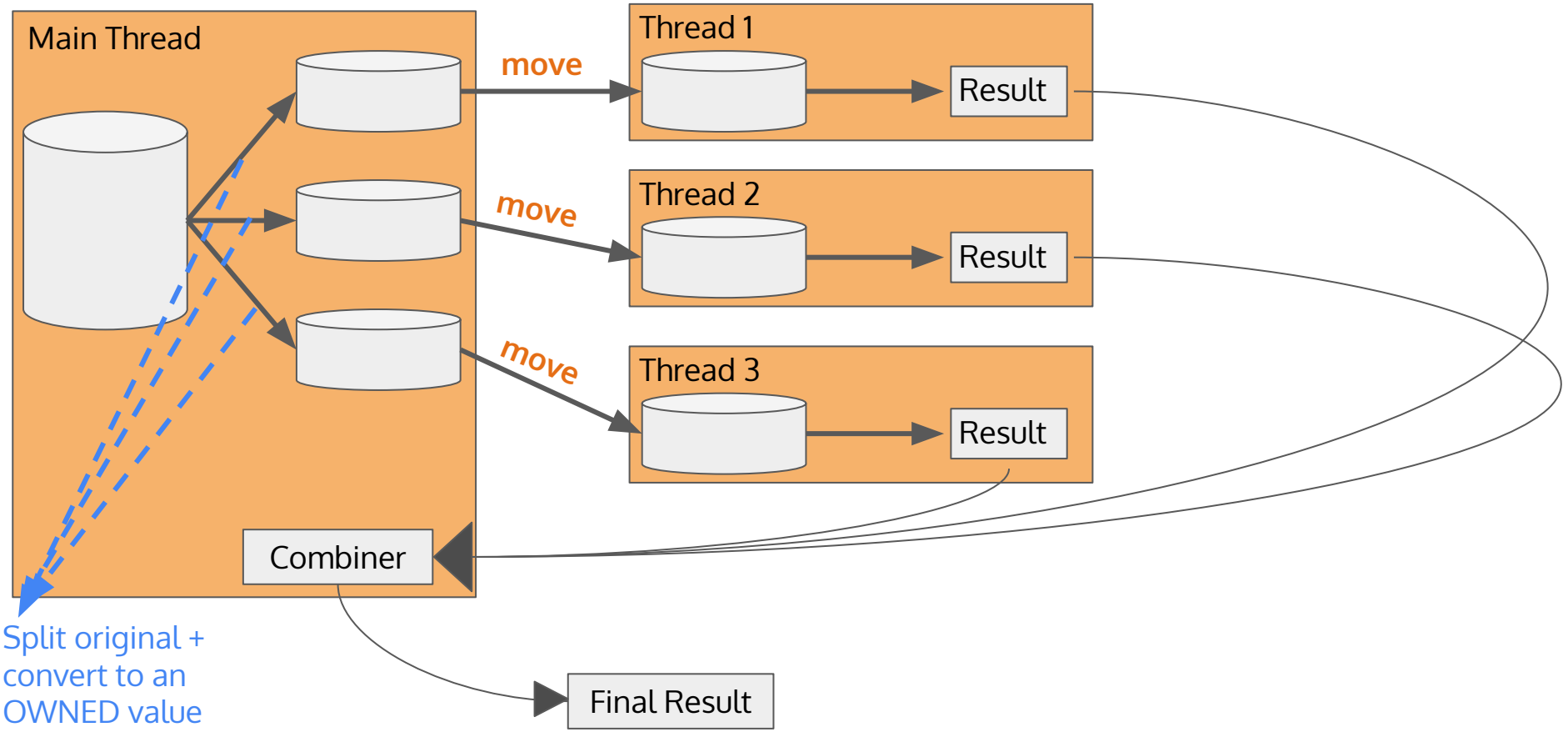
Reference:

- https://doc.rust-lang.org/stable/std/thread/struct.JoinHandle.html
- https://doc.rust-lang.org/std/thread/fn.spawn.html

Solving Problems with Parallelism

- Parallel Execution:
  - Split up your computation into different parts
  - Create threads to solve each part and solve at the same time
  - Combine the results at the end (if need be)
- We want our threads to communicate results back to the main thread
  - Combine the solution to the problem in the main thread

Solving Problems with Parallelism



Sending Data Between Threads

- To communicate between threads, we can:
  - Use shared memory (this can be very hard)
  - Pass messages between threads
- In Rust, we can pass messages with a **mpsc::Sender** and **mpsc::Receiver**
  - Created using **mpsc::channel()** – opens up a communication channel

Reference:

- https://doc.rust-lang.org/std/sync/mpsc/

Sending Data Between Threads

```rust
use std::sync::mpsc::{Sender, Receiver};
use std::sync::mpsc;
use std::thread;

fn main() {
    let (tx, rx): (Sender<String>, Receiver<String>) = mpsc::channel();

    thread::spawn(move || {
        tx.send("hello there!".to_string()).unwrap();
        println!("sent message from thread!")
    });

    let message = rx.recv().unwrap();
    println!("Main thread got message: {}", message);
}
```

MPSC???

- MPSC == Multiple Producer Single Consumer
  - The **mpsc** module in Rust provides message-based communication over channels
- Senders are **clone**-able (multi-producer) such that many threads can send simultaneously to one receiver (single-consumer)
  - **Clone** the sender/transmitter (tx) and **move** the cloned tx to a thread
  - Send messages within your threads
  - Receive messages on the main thread with your receiver (rx)
- Sidenote: you'll often see tx used as a shorthand for the sender (transmitter) and rx used for receiver in any communication/messaging context

Reference:
- https://doc.rust-lang.org/std/sync/mpsc/

No JoinHandle?

- **rx.recv()** blocks the current thread of execution until it receives either...
  - A message – returns **Ok(m)** where **m** is your message
  - All transmitters go out of scope (aka are dropped)
- When all transmitters go out of scope...
  - There are no more messages to receive since there are no senders to send messages
- Why no JoinHandles?
  - All transmitters go out of scope when the thread finishes
  - **rx.recv()** receives messages and keeps listening until all transmitters go out of scope
  - So we have code that blocks until all threads finish (and tx's are dropped)!!!

Sending Multiple Messages From 1 Thread

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

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

    thread::spawn(move || {
        tx.send("hello").unwrap();
        thread::sleep(Duration::from_millis(1000));
        tx.send("from").unwrap();
        thread::sleep(Duration::from_millis(1000));
        tx.send("another").unwrap();
        thread::sleep(Duration::from_millis(1000));
        tx.send("thread").unwrap();
    }); // tx goes out of scope here

    // this will block until it receives either Ok or Err
    while let Ok(msg) = rx.recv() {
        println!("Main thread got message: {}", msg);
    }
}
```

while let Loops + recv

- **while let <pattern> = <expression> { … }**
  - Allows you to loop while **<expression>** matches the pattern **<pattern>**
  - **<pattern>** DOES NOT have to be exhaustive
    - It is a single pattern
  - Break when the **<pattern>** does not match **<expression>**

Receiving Messages from Many Threads (BUGGY)

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

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

    for thread_idx in 0..10 {
        let tx_clone = tx.clone();
        thread::spawn(move || {
            for i in 0..10 {
                let msg = format!("message {i} from thread {thread_idx}");
                tx_clone.send(msg).unwrap();
            }
        });
    }

    while let Ok(msg) = rx.recv() {
        println!("Main thread got message: {msg:?}");
    }
}
```

Receiving Messages from Many Threads

Transmitters and Drop (Remember Ownership?)

- **rx.recv()** returns **Err()** when
  - All transmitters have been dropped and …
  - All messages have been received
- Transmitters are be dropped when…
  - They go out of scope
  - You call **drop()** on the transmitter
- Note: Rust automatically calls **drop()** when a variable goes out of scope, but you can do it ahead of time if you wish

Performing Some Real Computation


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

fn main() {
    let chunk_size = 10_000;
    let num_threads = 10;
    let max_data = chunk_size * num_threads;
    let data = (0..max_data).collect::<Vec<usize>>();
    let (tx, rx) = mpsc::channel();

    for i in 0..num_threads {
        let start = i * chunk_size;
        let end_excl = start + chunk_size;
        let owned_subvec = data[start..end_excl].to_vec();
        let tx_clone = tx.clone();

        thread::spawn(move || {
            let sub_vec_sum: usize = owned_subvec.into_iter().sum();
            tx_clone.send(sub_vec_sum).unwrap();
        });
    }

    drop(tx);
    let mut total = 0;
    while let Ok(value) = rx.recv() {
        println!("Receiver got {value}!");
        total += value;
    }
}
```

Important!!!

That’s All Folks!