Fearless Concurrency — Threads

Performant Software Systems with Rust — Lecture 12

Baochun Li, Professor
Department of Electrical and Computer Engineering
University of Toronto

Rust's Fearless Concurrency

- Rust's **ownership** and **type checking** features helped manage both memory safety and concurrency problems
- Many runtime concurrency errors becomes compile-time errors!
- Other programming languages, such as Erlang and Pony, may impose limits or performance tradeoffs
  - They implemented the Actor Model, which has elegant functionality for message-passing concurrency, but only obscure ways to share state between threads

Concurrent vs. Parallel Programming

- **Concurrent programming**: different parts of a program execute independently, but may not be at the same time
- **Parallel programming**: different parts of a program execute at the same time
- When we mention **concurrency**, we imply **concurrent or parallel** programming

Using Threads to Run Code Simultaneously

- **Multi-threading** allows the execution of multiple **threads** in parallel, at the same time
- But it also leads to subtle problems — a conventional topic in an OS course
  - Race conditions
  - Deadlocks
  - Thread synchronization
- Rust standard library uses a **1:1** (**kernel threads**) model

Creating a New Thread with spawn

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

fn main() {
    // passing a closure to `spawn`
    thread::spawn(|| {
        for i in 1..10 {
            println!("Printing number {i} from the spawned thread!");
            thread::sleep(Duration::from_secs(1));
        }
    });

    for i in 1..5 {
        println!("Printing number {i} from the main thread!");
        thread::sleep(Duration::from_secs(1));
    }
}
```

Waiting for All Threads to Finish

```rust
fn main() {
    // create a join handle to the spawned thread
    let handle = thread::spawn(|| {
        for i in 1..10 {
            println!("hi number {i} from the spawned thread!");
            thread::sleep(Duration::from_secs(1));
        }
    });

    for i in 1..5 {
        println!("hi number {i} from the main thread!");
        thread::sleep(Duration::from_sec(1));
    }

    // wait for the spawned thread to finish
    handle.join().unwrap();
}
```

Live Demo

Capturing the Environment of the Parent Thread

```
1 fn main() {
2     let v = vec![1, 2, 3];
3
4     let handle = thread::spawn(|| {
5         println!("Here's a vector: {v:?}");
6     });
7
8     handle.join().unwrap();
9 }
```

Live Demo

Capturing the Environment of the Parent Thread

```
 1 fn main() {
 2     let v = vec![1, 2, 3];
 3
 4     // `move` takes ownership of the environment
 5     let handle = thread::spawn(move || {
 6         println!("Here's a vector: {v:?}");
 7     });
 8
 9     handle.join().unwrap();
10 }
```

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

— The Go Language Documentation

The Actor Model

- An **actor** is the basic building block of concurrent computation
- In responding to messages that it receives, an **actor** makes local decisions, creates more actors, sends more messages, and modifies private states
- The **Actor Model** removes the need for lock-based synchronization
  - a mandatory design pattern in Erlang and Pony

Channels

- A general programming concept by which data is sent from one thread to another
- Two halves: one or more **transmitter(s)** and one or more **receiver(s)**
  - The transmitter half is the upstream location of a “river”, the receiver half is the downstream
  - **Closed** if either half is dropped

Multiple-Producer Single-Consumer Channels

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

fn main() {
    // returns a tuple that is destructured
    let (tx, rx) = mpsc::channel();
}
```

`send()` / `recv()` vs. `try_send()` / `try_recv()`

- Both variants return `Result<T, E>`
- `send()` / `recv()` **blocks** the thread's execution and wait until a channel has available capacity or becomes non-empty
- `try_send()` / `try_recv()` is **non-blocking** and returns immediately

Implementing the Actor Model: Use channels for message passing

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

fn main() {
    // multiple producer, single consumer (MPSC)
    let (tx, rx) = mpsc::channel();

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

    // use try_recv() to check if a message is available in
    // a non-blocking way
    let received = rx.recv().unwrap();
    println!("Got: {received}");
}
```

Transferring Ownership between Threads With Channels

Will this code compile successfully?

```
 1 fn main() {
 2     let (tx, rx) = mpsc::channel();
 3
 4     thread::spawn(move || {
 5         let val = String::from("hi");
 6         tx.send(val).unwrap();
 7         println!("val is {val}");
 8     });
 9
10     let received = rx.recv().unwrap();
11     println!("Got: {received}");
12 }
```

Cloning Multiple Producers with an MPSC Channel

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

let tx1 = tx.clone();

thread::spawn(move || {
    let vals = vec![
        String::from("hi"),
        String::from("from"),
        String::from("the"),
        String::from("thread"),
    ];

    for val in vals {
        tx1.send(val).unwrap();
        thread::sleep(Duration::from_secs(1));
    }
});

```

Again, do no communicate by sharing memory, and keep the states in each thread private and local!

But what if I really want to share memory?

Shared-State Concurrency with Mutex<T>

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

fn main() {
    let m = Mutex::new(5);

    {
        // block the current thread until having the lock
        // call to `lock` would fail if the holder thread panicked
        // as `m` is Mutex<T>, cannot access its value directly
        let mut num = m.lock().unwrap();
        // returned value is a smart pointer, `MutexGuard`,
        // which implements `Deref` and `Drop` traits
        *num = 6;
    }

    println!("m = {m:?}");
}
```

Sharing a Mutex<T> Between Threads

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

fn main() {
    let counter = Mutex::new(0);
    let mut handles = vec![];

    for _ in 0..10 {
        let handle = thread::spawn(move || {
            let mut num = counter.lock().unwrap();

            *num += 1;
        });
        handles.push(handle);
    }

    for handle in handles {
        handle.join().unwrap();
```

Will this compile successfully?

Multiple Ownership with Multiple Threads

```rust
use std::rc::Rc; // use reference counting to share ownership

fn main() {
    let counter = Rc::new(Mutex::new(0));
    let mut handles = vec![];

    for _ in 0..10 {
        let counter = Rc::clone(&counter);
        let handle = thread::spawn(move || {
            let mut num = counter.lock().unwrap();
    // snip
}
```

Will this compile successfully?

Atomic Reference Counting with Arc<T>

```rust
use std::sync::{Arc, Mutex}; // use atomic reference counting instead
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();
```

Other Atomic Types in the Rust Standard Library

- Several atomic types that provide atomic access to primitive types
  - **safe** to share between threads
  - such as `AtomicUsize`, `AtomicBool`, and so on
- `load()` and `store()`
- Guaranteed to be **lock-free**
- Can be used as building blocks of other concurrent types

```rust
use std::sync::atomic::{AtomicUsize, Ordering};

static GLOBAL_THREAD_COUNT: AtomicUsize = AtomicUsize::new(0);

// relaxed ordering doesn't synchronize anything except the global
// thread counter itself
let old_thread_count = GLOBAL_THREAD_COUNT.fetch_add(1,
    Ordering::Relaxed);

// this number may not be true at the moment of printing because some
// other thread may have changed static value already
println!("live threads: {}", old_thread_count + 1);
```

Extensible Concurrency with the `Send` Marker Trait

- `Send`, a `std::marker` trait, indicates that ownership of values of the type implementing `Send` can be transferred between threads
- It is safe to **send** it to another thread
- Automatically implemented when the compiler thinks it's appropriate

Types That Are Send

- Almost every Rust type is `Send`, with a few exceptions
  - `Rc<T>` cannot be `Send` as both threads may update the reference count at the same time
- Any type composed entirely of `Send` types is automatically marked as `Send`

Extensible Concurrency with the `Sync` Marker Trait

- `Sync` indicates that it is safe for the type implementing `Sync` to be **referenced** from multiple threads
- A type `T` is `Sync` if and only if `&T` is `Send`
- Automatically implemented when the compiler thinks it's appropriate

Types That Are `Sync`

- Almost every Rust type is `Sync`, with a few exceptions
  - `Rc<T>` is not `Sync`
  - `RefCell<T>` is not `Sync` since its implementation of borrow checking at runtime is not thread-safe
  - `Mutex<T>` is `Sync`
- Any type composed entirely of `Sync` types is automatically marked as `Sync`

So long, multi-threading!

Required Additional Reading

The Rust Programming Language, Chapter 16 and 21