Asychronous Rust

Performant Software Systems with Rust — Lecture 13

Baochun Li, Professor

Department of Electrical and Computer Engineering

University of Toronto

Entering Asynchronous Rust

Asychronous Programming

- Multi-threaded programming $\rightarrow$ **asynchronous** programming
  - operations may not finish sequentially in the order they were started
- CPU-bound vs. I/O-bound operations: video export vs. downloading from the Internet

Blocking System Calls

- Many operating system APIs — in the form of **system calls** are **blocking**
- We can certainly spawn new threads to use these blocking system calls
  - but watch out on the **overhead** of spawning threads
  - it is not a good idea to have too many threads

Non-Blocking Calls

- Compared to a `spawn` followed by a `join`, it would be great to have something like this:

```rust
let data = fetch_data_from(url).await;

println!("{data}");
```

Threads → Stackless Coroutines

- Migration to **stackless coroutines**
- Concurrency happens entirely (or mostly) within your code
- An **async runtime** — which is just another crate — manages async tasks
- Your code yields control from time to time, say using the `await` keyword
- Rust supports **async** programming with `Future`s and the `async` / `await` syntax

Revisiting the Idea of Parallelism vs. Concurrency

- **Analogy**: a team splitting up work for the course project
  - assign each member multiple tasks, assign each member one task, or a mix of both
- **Concurrency**: assigning each member multiple tasks, can switch but only make progress one at a time — a **single CPU core**
- **Parallelism**: each member takes one task, making progress at the same time — **multiple CPU cores**

Futures

- A **future**: is a value that may not be ready now, but will become ready at some point in the future
  - called a **task** or a **promise** in other programming languages
- Rust provides a `Future` trait as a building block from the Rust standard library (`std::future::Future`)
- Futures are simply types that implement the `Future` trait
- Each future holds its own information about the progress that has been made and what “ready” means

The Async Syntax

- You can apply the `async` keyword to blocks and functions to specify that they can be **interrupted** and **resumed**
- Within an `async` block or `async` function, you can use the `await` keyword to **await** an **future** — waiting for it to become “ready”
- Anywhere you `await` a future, that `async` block or function can **pause** and **resume**
- The process of checking with a future to see if its value is available yet is called **polling**

`Hello, world!` in Async Rust

```rust
// Define an async function.
async fn say_hello() {
    println!("Hello, world!");
}

// Boilerplate which lets us write `async fn main`, needs
// `tokio = { version = "1", features = ["full"] }` in Cargo.toml
#[tokio::main]
async fn main() {
    // Call an async function and await its result.
    say_hello().await;
}
```

The Async Runtime

- The async runtime schedules tasks to be run on CPU cores
- When one task suspends (say by calling `recv()` or `await`), another must be picked to run
- When this task resumes (say the message it is waiting for arrives), it should be ready to run

The Async Runtime

- Rust doesn’t provide such a runtime itself — leaving this to crates
- Much better choice than other languages, where the runtime may also manage memory (such as garbage collection in Go), or even becomes a full virtual machine
- A simple runtime (also called **executor**) can be single-threaded, can be written in less than 500 lines of code

Choice of Async Runtimes

- `tokio` is a full-featured, multi-threaded runtime that we often use
    - But other runtimes are also excellent and can have less overhead
    - such as `async-std` and `smol`
    - Honourable mention: `bytedance/monoio`
        - a more efficient thread-per-core runtime without work-stealing task queues, so that local data does not need to be `Sync` and `Send`

async and await

```rust
// An async function, but it doesn't need to wait for anything.
async fn add(a: u32, b: u32) -> u32 {
  a + b
}

async fn wait_to_add(a: u32, b: u32) -> u32 {
  sleep(1000).await;
  a + b
}
```

Calling an async function directly — without await — returns a **future**, which is a struct or enum that implements the Future trait and represents deferred computation

A Real-World Example — `page_title()`

A Real-World Example — `main()`

```rust
#[tokio::main]
async fn main() {
    let url = "https://www.rust-lang.org";

    match page_title(url).await {
        Some(title) => println!("Title of {} is: {}", url, title),
        None => eprintln!("Could not fetch/parse title for {}", url),
    }
}
```

Desugared async function

```rust
fn page_title(url: &str) -> impl Future<Output = Option<String>> + '_ {
    async move {
        let response = reqwest::get(url).await.ok()?;
        let response_text = response.text().await.ok()?;
        let document = Html::parse_document(&response_text);
        let selector = Selector::parse("title").ok()?;

        document
            .select(&selector)
            .next()
            .map(|title| title.inner_html())
    }
}
```

Desugaring `async` function

- Uses the `impl Trait` syntax — Traits as parameters
- Returns a `Future` with an associated type of `Output`
  - type is `Option<String>`
  - the same as the original return type from the `async fn` version of `page_title`

Desugaring async function

What Is `+ '_` Doing Here?

```rust
fn page_title(url: &str) -> impl Future<Output = Option<String>> + '_ {
    async move {
        // captures `url` inside this closure
    }
}
```

What Is `` + `_ `` Doing Here?

- The returned future captures `url`, which is a `&str`
  - The future borrows from `url`
  - So the future cannot `outlive` the reference it holds
- Therefore its lifetime is tied to the lifetime of `url`
- `` + `_ `` implies “This `impl Future` may contain borrows, and its lifetime is some anonymous lifetime that’s at most as long as the references it captures (like `url`).”

Good news: Only needed in older versions of the Rust compiler (before v1.7), no longer needed now.

Desugaring #[tokio::main]

```rust
#[tokio::main]
async fn main() {
    let url = "https://www.rust-lang.org";

    match page_title(url).await {
        Some(title) => println!("Title of {} is: {}", url, title),
        None => eprintln!("Could not fetch/parse title for {}", url),
    }
}
```

Desugaring #[tokio::main]

```rust
fn main() {
    // Create a multi-threaded Tokio runtime
    let rt = Runtime::new().expect("failed to create Tokio runtime");
    let url = "https://www.rust-lang.org";

    rt.block_on(async {
        match page_title(url).await {
            Some(title) => println!("Title of {} is: {}", url, title),
            None => eprintln!("Could not fetch or parse title for {}", url)
        }
    });
}
```

Deep dive: What's a `Future` in Rust anyway?

A value that represents a computation that may not be finished yet.

Future: Formal Definition

```rust
pub trait Future {
    type Output;

    fn poll(
        self: Pin<&mut Self>,
        cx: &mut Context<'_>,
    ) -> Poll<Self::Output>;
}
```

Key Points

- type `Output`: the type the future will eventually produce (like `u32`, `Result<T, E>`, etc.)
- `poll(...)`: asks the future "are you ready yet?"
  - Returns `Poll::Pending` if it's not done
  - Returns `Poll::Ready(value)` when it has the final result
- You almost never write `poll` by hand in normal code — the `async/await` syntax and executors do that for you

Futures are Lazy

- Just like **iterators** and unlike **threads**
  - A `std::thread::spawn` starts running immediately on another OS thread
  - A `Future` does nothing until an executor polls it

```rust
let fut = async { println!("hi"); }; // Nothing printed yet
// Only when we .await it (or poll it) will it actually run.
```

Executors and Tasks: The Runtime

- A runtime (like `Tokio`, `async-std`, etc.) provides an executor that
  - Keeps a queue of futures (often called `tasks`)
  - Repeatedly calls `poll` on them
- Uses a `Waker` to be notified when I/O or timers are ready, so it can poll again

What Does `self: Pin<&mut Self>` Mean?

```rust
pub trait Future {
    type Output;

    fn poll(
        self: Pin<&mut Self>,
        cx: &mut Context<'_>,
    ) -> Poll<Self::Output>;
}
```

What Does `self: Pin<&mut Self>` Mean?

```rust
future.poll(cx);
```

- `poll` takes `self` by `Pin<&mut Self>` instead of by `&mut Self`
- `Pin<T>` is a wrapper that says "Through this pointer, you are not allowed to move the value `T` in memory."

`Pin<&mut T>`

- Logically a mutable reference, but with an extra guarantee:
  - you can mutate `T`
  - but you must not move `T` to a different memory location via this reference

Why Do We Care About Moving?

What Does `self: Pin<&mut Self>` Enforce?

- Before calling `poll`, the executor must have pinned the future
- Inside `poll`, the future is treated as **immovable** (through this handle)

What Does an Executor Do?

```rust
use std::pin::Pin;
use std::future::Future;
use std::task::Context;

fn drive<F: Future + Unpin>(fut: &mut F, cx: &mut Context<'_>) {
    // Pin the &mut F temporarily
    let pinned = Pin::new(fut);
    match Future::poll(pinned, cx) {
        Poll::Pending => { /* ... */ }
        Poll::Ready(output) => { /* ... */ }
    }
}
```

The Unpin Marker Trait

If the future is not `Unpin` (e.g., most async fn futures), the executor must pin it in a stable place first:

```rust
let fut = my_async_fn();
let mut fut = Box::pin(fut); // heap-allocate and pin

// later:
let waker = /* ... */;
let mut cx = Context::from_waker(&waker);
let _ = fut.as_mut().poll(&mut cx);
```

The `Unpin` Marker Trait

```rust
pub trait Unpin {}
```

“Is it still safe to move this thing around in memory, even if it has been pinned?”

- **Yes**: the type is `Unpin`
- **No**: the type is not `Unpin` (written `!Unpin`)

The `Unpin` Marker Trait

- If `Self: Unpin`, then `Pin<&mut Self>` is the same as `&mut Self` in practice
- Most “normal” types are `Unpin`
- Compiler-generated `async fn` futures are usually not `Unpin`, so they need pinning
- So `self: Pin<&mut Self>` is a general signature:
  - Works for both `Unpin` and `!Unpin` futures
  - Enforces “must be pinned before polling” at the type level

What’s a `Waker`?

- A `Waker` is basically the doorbell for a future
- A `Waker` is a handle created by the executor that a future can store and later call `wake()` on, to say:
- **“Hey runtime, I might be ready now — please poll me again soon.”**

Where Does Waker Live?

```rust
pub trait Future {
    type Output;

    fn poll(
        self: Pin<&mut Self>,
        cx: &mut Context<'_>,
    ) -> Poll<Self::Output>;
}
```

The Context carries a Waker:

```rust
impl<'a> Context<'a> {
    pub fn waker(&self) -> &Waker { ... }
}
```

poll()

```rust
fn poll(self: Pin<&mut Self>, cx: &mut Context<'_>) -> Poll<Self::Output> {
    let waker = cx.waker();
    // ... maybe store it somewhere ...
}
```

Why Do We Need a `Waker`?

- Without `Waker`, if a future isn't ready yet, the executor would have to:
  - keep polling it in a loop (busy-waiting), or
  - poll every future all the time "just in case"
- That's terrible for CPU!

Why Do We Need a `Waker`?

- The contract in async Rust is:
  - When `poll` returns `Poll::Pending`, the future is not ready yet
  - If it ever becomes ready to make progress, it must ensure that someone calls `waker.wake()`
  - The executor, when `wake()` is called, will schedule that future to be polled again

Why Do We Need a `Waker`?

- So `Waker` is how a future tells the executor:
  - **“Some external event happened (I/O ready, timer fired, etc.). Please come back and poll me.”**

Spawning Tasks

```rust
use tokio::{spawn, time::{sleep, Duration}};

async fn say_hello() {
    // Wait for a while before printing
    sleep(Duration::from_millis(100)).await;
    println!("hello");
}

async fn say_world() {
    sleep(Duration::from_millis(100)).await;
    println!("world!");
}

#[tokio::main]
async fn main() {
    spawn(say_hello());
    spawn(say_world());

```

Joining Tasks

```rust
#[tokio::main]
async fn main() {
    let handle1 = spawn(say_hello());
    let handle2 = spawn(say_world());

    // Wait for both tasks to finish
    // spawn returns `JoinHandle`, which implements `Future`
    let _ = handle1.await;
    let _ = handle2.await;
}
```

Required Additional Reading

The Rust Programming Language, Chapter 17

The Rustonomicon, Chapter 8

Asynchronous Programming in Rust, Chapter 1-4

The State of Async Rust: Runtimes

Tokio: An Asynchronous Runtime