Lecture 19

Async I/O with Tokio

What is Tokio?

- Tokio is an event-driven, non-blocking I/O platform for writing asynchronous applications with RUST.
- The async-std library provides similar functionality.
- Network services are the most common domain for a non-blocking I/O system.
- Tokio is the most popular and established of these systems today.

An asynchronous hello world application:

- #[tokio::main] adds boilerplate code, now looks like std calls

```
use tokio::io;

#[tokio::main]
async fn main() -> Result<(), std::io::Error> {
    let mut stdout = io::stdout();
    let mut hello: &[u8] = b"Hello, world!\n";  //b? a slice
    io::copy(&mut hello, &mut stdout).await?;
    Ok(())
}
```

- We are generating a **Future**, so our main function is async.
- Since main is async, we need to use an executor to run it. That's why we use the *#[tokio::main]* attribute.
- Since performing I/O can fail (?), we return a Result.

.await?

- .await?:

  .await for chaining together Futures.

  ? for error handling.

- We use tokio::io::stdout() to get access to some value that lets us interact with standard output.
- looks really similar to std::io::stdout().
- a large part of the tokio API is simply async-ifying things from std.

tokio::io::copy

- An asynchronous version of std::io::copy.
- Instead of working with the *Read* and *Write* traits, this works with their async cousins: *AsyncRead* and *AsyncWrite*.
- A byte slice (&[u8]) is a valid AsyncRead, so we're able to store our input there.
- Stdout is an AsyncWrite.

Example 1

- Modify this application so it copies the entire contents of standard input to standard output.

```
use tokio::io;

#[tokio::main]
async fn main() -> Result<(), std::io::Error> {
    let mut stdin = io::stdin();
    let mut stdout = io::stdout();
    io::copy(&mut stdin, &mut stdout).await?;
    Ok(())
}
```

Spawning Processes

- Tokio provides a tokio::process module which resembles the std::process module.

```
use tokio::process::Command;

#[tokio::main]
async fn main() -> Result<(), std::io::Error> {
    Command::new("echo").arg("Hello, world!").spawn()?.await?;
    Ok(())
}
```

Spawning Processes

```
Command::new("echo").arg("Hello, world!").spawn()?.await?;
```

1. Create a new Command to run echo
2. Give it the argument "Hello, world!"
3. Spawn this, which may fail
4. ?: if it fails, return the error. Otherwise, return a Future
5. Using the .await: wait until that Future completes, and capture its Result
6. ?: if that Result is Err, return that error.

Delay_for

```
use tokio::time;
use tokio::process::Command;
use std::time::Duration;

#[tokio::main]
async fn main() -> Result<(), std::io::Error> {
    Command::new("date").spawn()?.await?;
    time::delay_for(Duration::from_secs(1)).await;
    Command::new("date").spawn()?.await?;
    time::delay_for(Duration::from_secs(1)).await;
    Command::new("date").spawn()?.await?;
    Ok(())
}
```

tokio::time::interval

- Create a stream of “ticks” .
- This program will keep calling date once per second until it is killed:

```
use tokio::time;
use tokio::process::Command;
use std::time::Duration;

#[tokio::main]
async fn main() -> Result<(), std::io::Error> {
    let mut interval = time::interval(Duration::from_secs(1));
    loop {
        interval.tick().await;
        Command::new("date").spawn()?.await?;
    }
}
```

Time to spawn

```
use std::time::Duration;
use tokio::process::Command;
use tokio::task;
use tokio::time;

#[tokio::main]
async fn main() -> Result<(), std::io::Error> {
    task::spawn(dating()).await??;
    Ok(())
}

async fn dating() -> Result<(), std::io::Error> {
    let mut interval = time::interval(Duration::from_secs(1));
    loop {
        interval.tick().await;
        Command::new("date").spawn()?.await?;
    }
}
```

??

- Calling spawn gives us back a JoinHandle<T::Output>.

```
pub fn spawn<T>(task: T) -> JoinHandle<T::Output>;
impl<T> Future for JoinHandle<T> {
    type Output = Result<T, JoinError>;
}
```

- The Future we provide as input is dating(), which has an output of type Result<(), std::io::Error>.
- The type of task::spawn(dating()) is JoinHandle<Result<(), std::io::Error>>.
- JoinHandle implements Future, so we end up with Result<Result<(), std::io::Error>, JoinError>.
- The first ? deals with the outer Result.
- The second ? deals with the std::io::Error.

Synchronous code

- Tokio’s performs blocking calls with the spawn_blocking function.
- The task::spawn_blocking function is similar to the task::spawn function
- But rather than spawning an non-blocking future on the Tokio runtime, it instead spawns a blocking function on a dedicated thread pool for blocking tasks.

Let’s network!

```
use tokio::io;
use tokio::net::{TcpListener, TcpStream};


#[tokio::main]
async fn main() -> io::Result<()> {
    let mut listener =
TcpListener::bind("127.0.0.1:8080").await?;

    loop {     //we loop forever
        let (socket, _) = listener.accept().await?;
        echo(socket).await?;
    }
}

async fn echo(socket: TcpStream) -> io::Result<()> {
    let (mut recv, mut send) = io::split(socket);
    io::copy(&mut recv, &mut send).await?;
    Ok(())
}
```

TcpListener binds a socket. The binding itself is asynchronous, so we use .await to wait for the listening socket to be available.

Accept new connections, and call our echo function and .await it.

We use tokio::io::split to split up our TcpStream into read and write halves, and then pass those into tokio::io::copy.

Let’s network!

- What should the behavior be if a second connection comes in while the first connection is still active?
- The program has just one task: awaits on each call to echo.
- So the second connection won’t be serviced until the first one closes.
- *Modify the program above so that it handles concurrent connections correctly.*

Let’s network!

- Solution: Wrap the echo call with tokio::spawn:

```
loop {
    let (socket, _) = listener.accept().await?;
    tokio::spawn(echo(socket));
}
```

TCP client and ownership

- It will establish a connection to a hard-coded server.
- Copy all of stdin to the server, and then copy all data from the server to stdout.

```
use tokio::io;
use tokio::net::TcpStream;

#[tokio::main]
async fn main() -> io::Result<()> {
    let stream = TcpStream::connect("127.0.0.1:8080").await?;
    let (mut recv, mut send) = io::split(stream);
    let mut stdin = io::stdin();
    let mut stdout = io::stdout();

    io::copy(&mut stdin, &mut send).await?;
    io::copy(&mut recv, &mut stdout).await?;

    Ok(())
}
```

TCP client and ownership

- But it’s limited.
- It only handles half-duplex protocols like HTTP, and doesn’t actually support keep-alive in any way.
- Can we use spawn to run the two copys in different tasks?

```
let send = spawn(io::copy(&mut stdin, &mut send));
let recv = spawn(io::copy(&mut recv, &mut stdout));

send.await??;
recv.await??;
```

- Note: In a half-duplex system, only one device can transmit at a time.

TCP client and ownership

- However, this doesn't compile:

```
error[E0597]: `stdin` does not live long enough
  --> src/main.rs:12:31
   |
12 |      let send = spawn(io::copy(&mut stdin, &mut send));
   |                           ---------^^^^^^^^^^------------
   |                           |        borrowed value does not
live long enough
   |                           argument requires that `stdin` is
borrowed for `'static`
19 | }
   | - `stdin` dropped here while still borrowed
```

- Copy Future does not own the stdin value.
- To fix this, we need to convince the compiler to make a Future that owns stdin.

TCP client and ownership

```
use tokio::io;
use tokio::spawn;
use tokio::net::TcpStream;

#[tokio::main]
async fn main() ->  Result<(), std::io::Error> {
    let stream = TcpStream::connect("127.0.0.1:8080").await?;
    let (mut recv, mut send) = io::split(stream);
    let mut stdin = io::stdin();
    let mut stdout = io::stdout();

    let send = spawn(async move {
        io::copy(&mut stdin, &mut send).await
    });
    let recv = spawn(async move {
        io::copy(&mut recv, &mut stdout).await
    });

    send.await??;
    recv.await??;

    Ok(())
}
```

Playing with lines

- Let's build an async program that counts the number of lines on standard input.

```
use tokio::prelude::*;
use tokio::io::AsyncBufReadExt;

#[tokio::main]
async fn main() -> Result<(), std::io::Error> {
    let stdin = io::stdin();
    let stdin2 = io::BufReader::new(stdin);
    let mut count = 0u32;
    let mut lines = stdin2.lines();
    while let Some(_) = lines.next_line().await? {
        count += 1;
    }
    println!("Lines on stdin: {}", count);
    Ok(())
}
```

Playing with lines

- Let’s take a list of file names as command line arguments, and count up the total number of lines in all the files.

```
use tokio::prelude::*;
use tokio::io::AsyncBufReadExt;

#[tokio::main]
async fn main() -> Result<(), std::io::Error> {
    let mut args = std::env::args();
    let _me = args.next(); // ignore command name
    let mut count = 0u32;
    for filename in args {
        let file = tokio::fs::File::open(filename).await?;
        let file2 = io::BufReader::new(file);
        let mut lines = file2.lines();
        while let Some(_) = lines.next_line().await? {
            count += 1;        }
    }
    println!("Total lines: {}", count);
    Ok(())
}
```

Make it asynchronous

```
use tokio::prelude::*;
use tokio::io::AsyncBufReadExt;

#[tokio::main]
async fn main() -> Result<(), std::io::Error> {
    let mut args = std::env::args();
    let _me = args.next(); // ignore command name
    let mut tasks = vec![];
    for filename in args {
        tasks.push(tokio::spawn(async {
            let file = tokio::fs::File::open(filename).await?;
            let file2 = io::BufReader::new(file);
            let mut lines = file2.lines();
            let mut count = 0u32;
            while let Some(_) = lines.next_line().await? {
                count += 1;}
            Ok(count) as Result<u32, std::io::Error>
        }));     }
    let mut count = 0;
    for task in tasks {
        count += task.await??;     }
    println!("Total lines: {}", count);
    Ok(())
}
```

Playing with lines

- Let’s change how we handle the count.
- Instead of .awaiting the count in the second for loop, let’s have each individual task update a shared mutable variable.
- You should use an Arc<Mutex<u32>>.

```
#[tokio::main]
async fn main() -> Result<(), std::io::Error> {
    let mut args = std::env::args();
    let _me = args.next(); // ignore command name
    let mut tasks = vec![];
    let count = Arc::new(Mutex::new(0u32));
    for filename in args {
        let count = count.clone();
        tasks.push(tokio::spawn(async move {
            let file = tokio::fs::File::open(filename).await?;
            let file = io::BufReader::new(file);
            let mut lines = file.lines();
            let mut local_count = 0u32;
            while let Some(_) = lines.next_line().await? {
                local_count += 1;            }
            let mut count = count.lock().await;
            *count += local_count;
            Ok(()) as Result<(), std::io::Error>   }));    }
    for task in tasks {
        task.await??;
    }
    let count = count.lock().await;
    println!("Total lines: {}", *count);
    Ok(())
}
```