Lecture 17

Zero-Cost Async IO (Part 1)

Why Async IO (in Rust)?

1. Minimize system resources for handling a large number of concurrent I/O tasks
2. Provide a zero-cost abstraction on top of the async I/O mechanisms provided by operating systems
3. Do it at a library level, instead of introducing a runtime to Rust

A sample problem

- Let's run a separate thread. This thread will have access to two atomic values:
  - ***AtomicBool:*** whether we want the thread to keep running
  - **AtomicUsize:** a counter

- As long as the AtomicBool is true, the thread will:
  1. Sleep for a given number of milliseconds
  2. Print a message to the console
  3. Increment the AtomicUsize counter

A sample problem

```
use std::sync::atomic::{AtomicBool, AtomicUsize, Ordering};
use std::sync::Arc;
use std::thread::{sleep, spawn};
use std::time::Duration;

#[derive(Clone)]
pub struct Interval {
    counter: Arc<AtomicUsize>,
    still_running: Arc<AtomicBool>,
}

impl Drop for Interval {
    fn drop(&mut self) {
        println!("Interval thread shutting down");
        self.still_running.store(false, Ordering::SeqCst);
    }
}
```

A sample problem

```
impl Interval {
    fn from_millis(millis: u64) -> Interval {
     let duration = Duration::from_millis(millis);
     let counter = Arc::new(AtomicUsize::new(0));
     let counter_clone = counter.clone();
     let still_running = Arc::new(AtomicBool::new(true));
     let still_running_clone = still_running.clone();

     spawn(move || {
       println!("Interval thread launched");
       while still_running_clone.load(Ordering::SeqCst)
       {
        sleep(duration);
        let old = counter_clone.fetch_add(1,Ordering::SeqCst);
        println!("Thread alive, value was: {}", old);}});

       Interval {counter, still_running}}

    fn get_counter(&self) -> usize {
        self.counter.load(Ordering::SeqCst)}
}
```

A sample problem – The main function

```
mod interval;
use self::interval::Interval;

fn main() {
   let interval = Interval::from_millis(500);
   let duration = std::time::Duration::from_millis(2000);
   for i in 1..11 {
    println!("Iteration number {}, counter is {}", i,
        interval.get_counter());
    std::thread::sleep(duration);
   }
}
```

A sample problem – Output

```
Iteration number 1, counter is 0
Thread launched
Thread still alive, value was: 0
Thread still alive, value was: 1
...
Thread still alive, value was: 33
Thread still alive, value was: 34
Iteration number 10, counter is 35
Thread still alive, value was: 35
Thread still alive, value was: 36
Thread still alive, value was: 37
Thread still alive, value was: 38
Interval thread shutting down
```

What are the problems with this approach?

- Missing some updates in the main thread.
- The counter jumps!
- we're delaying for 2 seconds instead of half a second. Let's instead delay for a tenth of a second (100ms) in the main thread, and check if the value has changed since last time.

```
fn main() {
    let interval = Interval::from_millis(500);
    let duration = std::time::Duration::from_millis(100);
    let mut last = interval.get_counter();
    for i in 1..51 {
     let curr = interval.get_counter();
     if curr != last {
      last = curr;
      println!("Iteration number {}, counter is {}", i, curr);
     }
    std::thread::sleep(duration);
    }
}
```

A sample problem – Output

```
Interval thread launched
Interval thread alive, value was: 0
Iteration number 6, counter is 1
Interval thread  alive, value was: 1
Iteration number 11, counter is 2
Interval thread  alive, value was: 2
Iteration number 16, counter is 3
Interval thread  alive, value was: 3
Iteration number 21, counter is 4
Interval thread  alive, value was: 4
Iteration number 26, counter is 5
Interval thread  alive, value was: 5
Iteration number 31, counter is 6
Interval thread  alive, value was: 6
Iteration number 36, counter is 7
Interval thread  alive, value was: 7
Iteration number 41, counter is 8
Interval thread  alive, value was: 8
Iteration number 46, counter is 9
Interval thread alive, value was: 9
Interval thread shutting down
```

We didn't lose any counter updates

More problems?

- We are dedicating an entire OS thread to this sleep-and-check iteration.
- With 50 different similar tasks going on, It would require 49 extra threads, most of which would sit around sleeping the majority of the time.
- That's highly wasteful. (and not guaranteed! We got lucky!)

We need to be able to abstract over "this thing will produce a value in the future."

Enough why – tell me how!

Example 1: Sleeping

- Write a program which will print the message Sleeping 10 times, with a delay of 0.5 seconds.
- It should also print the message Interrupting 5 times, with a delay of 1 second.

```
use std::thread::{sleep};
use std::time::Duration;
fn sleeping() {
 for i in 1..=10 {
 println!(&quot;Sleeping {}&quot;, i);
 sleep(Duration::from_millis(500));
 }
}
fn interrupting() {
 for i in 1..=5 {
 println!(&quot;Interrupting {}&quot;, i);
 sleep(Duration::from_millis(1000));
 }
}
fn main() {
 sleeping();
 interrupting();
}
```

- This program runs the two operations **synchronously**, first printing Sleeping, then Interrupting.
- Instead, we would want to have these two sets of statements printed in an **interleaved** way.

Introducing async

```
use async_std::task::{sleep, spawn};
use std::time::Duration;

async fn sleeping() {
    for i in 1..=10 {
        println!("Sleeping {}", i);
        sleep(Duration::from_millis(500)).await;
    }
}
async fn interrupting() {
    for i in 1..=5 {
        println!("Interrupting {}", i);
        sleep(Duration::from_millis(1000)).await;
    }
}
#[async_std::main]
async fn main() {
    let sleeping = spawn(sleeping());
    interrupting().await;
    sleeping.await;
}
```

Introducing async

- Both sleeping and interrupting now say **async** in front of fn.
- After the calls to sleep, we have a **.await**.
- Note that this is not a .await() method call, but new syntax.
- We have a new attribute **#[async_std::main]** on the main function.
- The main function also has **async** before fn.
- The call to interrupting() is now followed by **.await**.

Introducing async

- Instead of join()ing, we use the .await syntax.
- That may look like a large list of changes. But in reality, our code is almost identical structural to the previous version, which is a real testament to the async/.await syntax.
- Now everything works under the surface the way we want:

A single operating system thread making ***non-blocking*** calls!

- Let’s analyze what each of these changes actually means.

async functions

- Adding async to the beginning of a function definition does three things:

1. It allows you to use .await.
2. It modifies the return type of the function.

async fn foo() -> Bar returns

impl std::future::Future<Output=Bar>.

3. Automatically wraps up the result value in a new Future.

async functions -- second point

- There's a trait called **Future** defined in the standard library.
- It has an associated type **Output**.
- What this trait means is:

I promise that, when I complete, I will give you a value of type Output.

(We'll play around with Future values more later.)

First steps….

- Rewrite the signature of sleeping to not use the async keyword by modifying its result type.
- Note that the code will not compile when you get the type right.
- Pay attention to the error message you get.
- The result type of async fn sleeping() is the implied unit value ().
- Therefore, the Output of our Future should be unit.
- This means we need to write our signature as:

```
fn sleeping() -> impl std::future::Future<Output=()>
```

Solution

```
fn sleeping() -> impl std::future::Future<Output=()>
```

- However, with only that change in place, we get the following error messages:

```
error[E0728]: `await` is only allowed inside `async` functions and blocks
 --> src/main.rs:7:9
  |
4 | fn sleeping() -> impl std::future::Future<Output=()> {
  |    ------- this is not `async`
...
7 |         sleep(Duration::from_millis(500)).await;
  |         ^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^ only allowed inside `async` 
functions and blocks
error[E0277]: the trait bound `(): std::future::Future` is not satisfied
 --> src/main.rs:4:17
  |
4 | fn sleeping() -> impl std::future::Future<Output=()> {
  |                 ^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^ the trait 
`std::future::Future` is not implemented for `()`
  |
  = note: the return type of a function must have a statically known size
```

Solution

- The first message is pretty direct: you can only use the .await syntax inside an async function or block.

```
async {
     ….
}
```

- The second error message is a result of the first: the async keyword causes the return type to be an impl Future.
- Without that keyword, our for loop evaluates to (), which isn't an impl Future.

Step 2 ….

- Fix the compiler errors by introducing an async block inside the sleeping function. Do not add async to the function signature, keep using impl Future.
- Wrapping the entire function body with an async block solves the problem:

```
fn sleeping() -> impl std::future::Future<Output=()> {
    async {
        for i in 1..=10 {
            println!("Sleeping {}", i);
            sleep(Duration::from_millis(500)).await;
        }
    }
}
```

.await a minute

- Maybe we don't need all this async/.await.
- What if we removed calls to .await in sleeping? It compiles, but note the warning:

```
warning: unused implementer of `std::future::Future` that must
be used
 --> src/main.rs:8:13
  |
8 |             sleep(Duration::from_millis(500));
  |             ^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^
  |
  = note: `#[warn(unused_must_use)]` on by default
  = note: futures do nothing unless you `.await` or poll them
```

- We're generating a Future value but not using it.
- We're going to have to implement our function with much more direct usage of the Future values

Dropping async block

- If we drop the async block, we end up with this code:

```
fn sleeping() -> impl std::future::Future<Output=()> {
    for i in 1..=10 {
        println!("Sleeping {}", i);
        sleep(Duration::from_millis(500));
    }
}
```

- This gives us an error message we saw before:

```
error[E0277]: the trait bound `(): std::future::Future` is not
satisfied
 --> src/main.rs:4:17
  |
4 | fn sleeping() -> impl std::future::Future<Output=()> {
  |                  ^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^ the
trait `std::future::Future` is not implemented for `()`
  |
```

Dropping async block

- This makes sense: the for loop evaluates to (), and unit does not implement Future.
- One way to fix this is to add an expression after the for loop that evaluates to something that implements Future.
- And we already know one such thing: ***sleep***.

```
fn sleeping() -> impl std::future::Future<Output=()> {
    for i in 1..=10 {
        println!("Sleeping {}", i);
        sleep(Duration::from_millis(500));
    }
    sleep(Duration::from_millis(0))
}
```

Dropping async block

- We still get a warning about the unused Future value inside the for loop, but not the one afterwards: return type.
- Sleeping for 0 milliseconds is just do nothing.
- It would be nice if there was a “dummy” Future .
- Replacing the sleep call after the for loop with a call to ready.

```
fn sleeping() -> impl std::future::Future<Output=()> {
    for i in 1..=10 {
        println!("Sleeping {}", i);
        sleep(Duration::from_millis(500));
    }
    async_std::future::ready(())
}
```

Implement our own Future

- To unpeel a bit more, let’s make our life harder, and not use the ready function.
- Instead, we’re going to define our own struct which implements a Future – DoNothing.

```
use std::future::Future;

struct DoNothing;

fn sleeping() -> impl Future<Output=()> {
    for i in 1..=10 {
        println!("Sleeping {}", i);
        sleep(Duration::from_millis(500));
    }
    DoNothing
}
```

Implement our own Future

- Compiler says ….

```
the trait bound `DoNothing: std::future::Future` is not
satisfied
```

- So let’s add in a trait implementation:

```
impl Future for DoNothing {                    }
```

- Which fails with:

```
error[E0046]: not all trait items implemented, missing:
`Output`, `poll`
 --> src/main.rs:7:1
7 | impl Future for DoNothing {
  | ^^^^^^^^^^^^^^^^^^^^^^^^^ missing `Output`, `poll` in
implementation
  = note: `Output` from trait: `type Output;`
  = note: `poll` from trait: `fn(std::pin::Pin<&mut Self>,
&mut std::task::Context<'_>) -> std::task::Poll<<Self as
std::future::Future>::Output>`
```

Implement our own Future

- We don’t really know about the Pin<&mut Self> or Context thing yet, but we do know about Output.
- And since we were previously returning a (), let’s do the same thing here.

```
use std::pin::Pin;
use std::task::{Context, Poll};

impl Future for DoNothing {
    type Output = ();

    fn poll(self: Pin<&mut Self>, ctx: &mut Context) ->
Poll<Self::Output> {
        unimplemented!()
    }
}
```

Implement our own Future

- It compiles! Of course, it fails at runtime due to the ***unimplemented!()*** call:

```
thread 'async-std/executor' panicked at 'not yet implemented',
src/main.rs:13:9
```

- Now let’s try to implement poll.
- We need to return a value of type Poll<Self::Output> or Poll<()>. Let’s look at the definition of Poll:

```
pub enum Poll<T> {
    Ready(T),
    Pending,
}
```

Implement our own Future

- Ready means “our Future is complete, and the output is T”
- while Pending means “it’s not done yet.”
- Given that our DoNothing wants to return the output of () immediately, we can just use Ready .
- Implement a working version of poll.

```
fn poll(self: Pin<&mut Self>, _ctx: &mut Context) ->
Poll<Self::Output> {
    Poll::Ready(())
}
```

The third async difference

- Automatically wraps up the result value in a new Future
- Let’s simplify the definition of sleeping to:

```
fn sleeping() -> impl Future<Output=()> {
    DoNothing
}
```

- The compiles and runs just fine. Let’s try switching back to the async way of writing the signature:

```
async fn sleeping() {
    DoNothing
}
```

The third async difference

- This now gives us an error:

```
error[E0271]: type mismatch resolving `<impl
std::future::Future as std::future::Future>::Output == ()`
  --> src/main.rs:17:20
   |
17 | async fn sleeping() {
   |                    ^ expected struct `DoNothing`, found
()
   |
   = note: expected type `DoNothing`
              found type `()`
```

- You see, when you have an async function or block, the result is automatically wrapped up in a Future.
- So instead of returning a DoNothing, we're returning a impl Future<Output=DoNothing>. And our type wants Output=().

The third async difference

- Working around this is pretty easy: you simply append .await to DoNothing:

```
async fn sleeping() {
    DoNothing.await
}
```

- This gives us a little more intuition for what .await is doing:
- it's extracting the () Output from the DoNothing Future… somehow.
- However, we still don't really know how it's achieving that. Let's build up a more complicated Future .

SleepPrint

- We’re going to build a new Future implementation which:
  - Sleeps for a certain amount of time
  - Then prints a message
- This is going to involve using pinned pointers – Rust normally solves a problem with a box solution.
- SleepPrint will wrap an existing sleep Future with our own implementation of Future.
- Since we don’t know the exact type of the result of a sleep call (it’s just an impl Future), we’ll use a parameter:

```
struct SleepPrint<Fut> {
    sleep: Fut,
}
```

SleepPrint

- And we can call this in our sleeping function with:

```
fn sleeping() -> impl Future<Output=()> {
    SleepPrint {
        sleep: sleep(Duration::from_millis(3000)),
    }
}
```

- Of course, we now get a compiler error about a missing Future implementation. So let's work on that. Our impl starts with:

```
impl<Fut: Future<Output=()>> Future for SleepPrint<Fut> {
    ...
}
```

SleepPrint

- This says that SleepPrint is a Future if the sleep value it contains is a Future with an Output of type (). Which is true in the case of the sleep function, so we're good.

- We need to define Output:

```
type Output = ();
```

- And then we need a poll function:

```
fn poll(self: Pin<&mut Self>, ctx: &mut Context) ->
Poll<Self::Output> {
    ...
}
```

SleepPrint

- Pinned pointers. Lets cheat to get started.
- We need to project the Pin<&mut Self> into a Pin<&mut Fut> so that we can work on the underlying sleep Future.
- Yikes, how would I do that? Remember we are cheating, so unsafe.

```
let sleep: Pin<&mut Fut> = unsafe { self.map_unchecked_mut(|s|
&mut s.sleep) };
```

SleepPrint

- We've got our underlying Future, and we need to do something with it.
- The only thing we can do with it is – call poll.
- poll requires a &mut Context, which fortunate is provided.
- That Context contains information about the currently running task, so it can be woken up (via a Waker) when the task is ready.
- For now, let's do the only thing we can reasonably do:

```
match sleep.poll(ctx) {
    ...
}
```

SleepPrint

- We’ve got two possibilities:
- If poll returns a Pending, it means that the sleep hasn’t completed yet. In that case, we want our Future to also indicate that it’s not done.
- To make that work, we just propagate the Pending value:

```
Poll::Pending => Poll::Pending,
```

SleepPrint

- However, if the sleep is already complete, we'll receive a Ready(()) variant.
- In that case, it's finally time to print our message and then propagate the Ready:

```
Poll::Ready(()) => {
    println!("Inside SleepPrint");
    Poll::Ready(())
},
```

- We've built a more complex Future from a simpler one. Hooray!