CPSC 524 Final Project

Async Runtime in Rust

Lucas Jehu Silva Shepard

12/15/2022

1 Introduction

Rust is a general-purpose programming language that is exceptional for developing safe, parallelized, system-level code. At compile-time, it can identify many memory and concurrency bugs that would otherwise plague programs written in other languages. This is possible thanks to Rust's *borrow-checker*. Every value has an "owning scope," which starts and ends around curly braces. Passing or returning values between scopes means transferring ownership. Values can also be *borrowed* by another scope, and returned to their owner at a later point. These happen via either mutable or immutable borrows. If a value is mutably borrowed, no other scope is allowed to borrow that value, effectively allowing for the detection and prevention of data races at compile time. This is just one example of the constructs provided by Rust that enable the language's self-proclaimed *fearless concurrency,* and motivate Rust as a choice of language for this project.

An asynchronous runtime is a library that executes async applications. One common way of implementing an async runtime is by combining a reactor with one or more executors. The *Reactor* is a design pattern that provides a subscription mechanism for asynchronous events. These events are delivered concurrently, demultiplexed, and dispatched to the

appropriate handler. The *Executor* is a design pattern responsible for handling the scheduling and execution of tasks.

Rust does not provide an async runtime in its standard library. Instead, users can pick and choose from a swathe of open-source libraries (dubbed “crates” in Rust) based on their desired application. The goal of this project is to write an async runtime environment in Rust. This involves developing a task model, a reactor that can accept task subscriptions, an executor for handling tasks dispatched by the reactor, and the language constructs required for this runtime to operate seamlessly in an async Rust application. A sample Rust application will be submitted that showcases this async runtime.

2 Rust Futures

This section will explore the basics of Rust semantics like structs, traits, borrows, and concurrency primitives, in the context of the most important async primitive: the Future.

2.1 Towards Rust Futures

Rather than the traditional inheritance model employed in languages like C++, Rust opts for a Composition over Inheritance model. Rust *traits* define functionality that a type can share with other types. The `Future` trait defines a type that can eventually produce a value. The easiest way of writing a future is with an `async` block. Inside this block, we can `await` other futures, as follows:

```
let future = async {
    some_asynchronous_operation.await();
};
```

To build towards an understanding of how Rust implements this trait, let’s attempt to implement this ourselves. First of all, we need this trait to provide functionality that advances the future as far towards completion as possible. If the future successfully completes, we can return the value it

produced. Otherwise, we should put the future to sleep and wake it up when more progress can be made. The signature could look something like this:

```
trait MyFuture {
    type Output;
    // Mutably borrow the object with `&mut self` so polling can modify it.
    // Provide a `wake` function that can be registered as a callback.
    // Either return a value wrapped inside `Poll::Ready`, or a pending status
    // `poll` should be invoked from a ready queue of tasks.
    fn poll(&mut self, wake: fn()) -> Poll<Self::Output>;
}

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

Remember that Rust seeks to avoid many of the. This means additional constructs need to be added to `MyFuture` in order to make it bulletproof against potential mishandling. These will be explored in the following sections.

2.2 Wakers and Channels

One problem with this signature is that the `wake` function has no idea which future actually called it. Indeed, notice `wake` is a function taking no arguments, with no return value. What we really need is for `wake` to be some kind of `Waker` object that retains information about the specific task we're running. For a task to become a `Waker`, let's define a `MyWake` trait it should implement.

```
trait MyWake {
    fn wake(self: Self);
}
```

Our task struct should hold a future instance that can be sent between threads, as well as a mechanism for re-adding itself to a ready queue once it's been woken up. To accomplish this, let's use Rust's message-passing primitive, a Multiple-Producer Single-Consumer channel. This task struct can be implemented as follows:

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

// Task struct, holding a Future instance and a ready_queue enqueuer.
struct MyTask {
    // Since futures are a trait, we need to store an object implementing this
    // trait via dynamic dispatch (using the `dyn` keyword) on the heap (using
    // We want this future to be sent between threads, so we must require it
    // implements the `Send` trait.
    future: Box<dyn MyFuture<Output = ()> + Send>,
    // Multiple-Producer Single-Consumer channel.
    // Tasks are sent through this ready queue to be scheduled.
    // We use a `SyncSender` as opposed to a regular `Sender` to allow for sen
    // `MyTask` between threads.
    ready_queue: mpsc::SyncSender<MyTask>
}
```

We can implement the `MyWake` trait for our task as follows:

```
// Implement the MyWake trait for MyTask.
impl MyWake for MyTask {
    fn wake(self: Self) {
        // We can't use `self.ready_queue` and pass `self` to the channel
        // at once due to Rust's single-ownership, so let's create a copy
        // of the channel.
        let queue = self.ready_queue.clone();
        queue.send(self).expect("Failed to send task to `ready_queue`");
    }
}
```

2.3 Shared-State Concurrency

This implementation works just fine, but has a significant drawback: only one reference to `MyTask` can exist at a time. To understand this, we need to talk a bit more about Rust's ownership system. Specifically, since `MyTask` holds a `Box` instance, i.e. a heap-allocated pointer, only one copy

of this pointer is allowed at a time. Otherwise, there's the possibility of race conditions due to a shared pointer. This fact is explicit in the language's syntax: `Box` doesn't implement the `Copy` trait, and therefore `MyTask` doesn't implement the `Copy` trait, and therefore a task instance can't be duplicated. Instead, when we're sending this task to `ready_queue`, we actually send an owning instance. This is different from our `mpsc::SyncSender` instance: since we're working with a Multiple-Producer, Single-Consumer channel, it's completely fine to duplicate sender instances, as shown via the `.clone()` method

However, notice we'll likely be working with multiple instances of a single `MyTask`. Remember, Rust does not normally allow shared mutable references to a value. Therefore, if we're passing an instance of this task to a worker thread, it'd be cumbersome if this same worker thread was also responsible for spuriously waking itself up when more progress can be made towards completing the `future`. Therefore, it'd be helpful to work with thread-safe shared references to `MyTask`, as opposed to an owning instance. We can start by enabling mutual exclusion on our boxed future, as follows:

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

struct MyTask {
    // Wrap our boxed future in a Mutex, enabling mutually-exclusive access.
    future: Mutex<Box<dyn MyFuture<Output = ()>>>,
    // ...
}
```

Now, let's allow shared references using Rust's atomic reference counter, `Arc`:

```
use std::sync::{mpsc, Arc};

trait MyWake {
    fn wake(self: Arc<Self>);
}

struct MyTask {
```

```
    // ...
    ready_queue: mpsc::SyncSender<Arc<MyTask>>
}

impl MyWake for MyTask {
    fn wake(self: Arc<Self>) {
        // Create a shared reference to `MyTask`.
        let clone = Arc::clone(&self);
        self.ready_queue.send(clone).expect("Failed to send task to `ready_que
    }
}
```

So our new definition of Future becomes:

```
trait MyFuture {
    type Output;
    // `waker` now takes in a struct implementing the `Wake` trait.
    fn poll(&mut self, waker: impl Wake) -> Poll<Self::Output>;
}
```

2.4 Pinning

Note that the `MyTask` struct holds an owning instance to a mutually-exclusive pointer to a struct implementing the `MyFuture` trait. As long as the value remains boxed at the same memory location, it'll be valid. So what happens if we move this memory location? Here's an example:

```
// Create a mutually-exclusive shared reference to an option holding a future
let future_ref = Arc::new(Mutex::new(Box::new(Some(async {println!("hello worl
// Create a new reference to this future
let cloned_ref = Arc::clone(future);

// Spawn a new thread, and give it ownership of cloned_ref.
let handle = thread::spawn(move || {
    let future = cloned_ref
        .lock().unwrap()    // Retrieve a mutex guard to the future option.
        .take().unwrap();   // Take ownership of future, leaving behind a None
    // `future` goes out of scope and gets dropped.
});

handle.join();    // Wait for the new thread to finish running.
let future = future_ref
```

```
    .lock().unwrap()     // Retrieve a mutex guard.
    .take().unwrap();    // The future was taken from the option, so this pani
```

It'd be problematic if threads sharing a reference to a future could arbitrarily decide to take ownership of that future. Rust provides a special type called `Pin<T>` that pins a value to memory, preventing it from being moved around. In order to fix the above code, we could replace the call to `Box::new` with `Box::pin`. This would return a `Pin<Box<T>>` , causing the above code to fail compiling. We've now successfully caught the move at compile-time, preventing potential runtime panics.

Similarly, we'd like the `Box` holding `MyFuture` to be pinned in memory, as follows:

```
struct MyTask {
    future: Mutex<Pin<Box<dyn MyFuture<Output = ()> + Send>>>,
    // ...
}
```

And we'd like `MyFuture` to ensure we can only call `poll` when it is indeed pinned:

```
trait MyFuture {
    type Output;
    fn poll(self: Pin<&mut self>, wake: impl Wake) -> Poll<Self::Output>;
}
```

2.5 Finishing Touches

This is almost a complete view of Rust Futures, and the properties that enable their thread-safety at compile time. The actual signature is as follows:

```
trait Future {
    type Output;
    fn poll(self: Pin<&mut self>, wake: &mut Context<'_>) -> Poll<Self::Output
}
```

Here, `Context` is simply a wrapper around a `Waker` object. A `Waker` is created from a type implementing the `Wake` trait, such as the `MyTask` object from earlier. The idea here is to only provide the information necessary to wake the object, rather than the entire `MyTask` struct. We now have a complete understanding of how Rust implements its futures!

3 I/O Events

As outlined in Section 2.1, in order to implement a future, we need some way of pending a running future if `poll` returns `Poll::Pending` , and picking it back up once progress can be made. One way of doing this would be to continually poll until we receive a `Poll::Ready<T>`, but this obviously isn't ideal for performance reasons. Instead, we can use OS-level I/O primitives that allow a thread to block on I/O events and return once they've completed.

3.1 Epoll

Linux uses the `epoll` family of system calls to monitor I/O events. These are provided by the kernel in C, with the following signature:

```
int epoll_create(int size);
int epoll_ctl(int epfd, int op, int fd, struct epoll_event *event);
int epoll_wait(int epfd, struct epoll_event *events, int maxevents, int timeou
int close(int fd);
```

- `epoll_create` initializes an epoll queue, and returns it as a file descriptor. This should be called once by our program.
- `epoll_ctl` is used to add, modify, and remove entries from our epoll's interest list. This means performing an add/modify/remove operation `op` on a file descriptor `fd` of interest, in our epoll queue `epfd`.
- `epoll_wait` suspends the current thread until a file descriptor in our epoll queue `epfd` triggers an event.

- `close` is not an epoll system call, but rather the general-purpose system call for closing a file descriptor. This should be called at the end of the program on the descriptor returned by `epoll_create`.

The `struct epoll_event` is defined as follows:

```
#define EPOLL_PACKED __attribute__((packed))

typedef union epoll_data {
   void    *ptr;
   int      fd;
   uint32_t u32;
   uint64_t u64;
} epoll_data_t;

 struct epoll_event {
     uint32_t     events;    /* Epoll events */
     epoll_data_t data;      /* User data variable */
 } EPOLL_PACKED;
```

- `events` is a bit mask indicating which event occurred for a monitored file descriptor.
- `data` specifies what the kernel should return once an event is triggered in `epoll_wait`. This is helpful to store, for example, the address of a function callback.

In-kernel, epoll boils down to a set of file descriptors that are monitored for events. When a program creates an epoll instance using the **`epoll_create`** system call, the kernel allocates an epoll instance data structure and returns a file descriptor that represents the epoll instance. The program can then use the file descriptor to manipulate the epoll instance using the **`epoll_ctl`** and **`epoll_wait`** system calls.

The epoll instance data structure contains a list of file descriptors that are being monitored, as well as a list of events that the program is interested in for each file descriptor. When an event occurs on one of the file descriptors, the kernel adds an entry to the list of events for that file

descriptor. The kernel can detect when an event occurs since these events involve simple I/O operations like reading or writing. For example, an open TCP socket will eventually receive network packets, which will trigger an interrupt in-kernel, which can then mark this I/O event as having occurred.

When the program calls `epoll_wait`, the kernel checks the list of events for each file descriptor in the epoll instance. If there are any events that match the events the program is interested in, the kernel adds them to a list of events that is returned to the program. The kernel uses a a red-black tree to store the file descriptors in the epoll instance. This allows the kernel to efficiently add and remove file descriptors from the epoll instance, as well as quickly search for events on specific file descriptors.

3.2 Using Epoll in Rust

In order to invoke these system calls in Rust, we have to provide an FFI layer. Notice the `epoll_data` union has a maximum size of 64 bits, thanks to the `uint64_t` and the `void *`. Because of this, we can get away with a `usize` in our FFI. Creating the FFI layer is actually quite easy in Rust:

```
mod ffi {
  #[link(name = "c")]
  extern "C" {
    // https://man7.org/linux/man-pages/man2/epoll_create.2.html
    pub fn epoll_create(size: i32) -> i32;
    // https://man7.org/linux/man-pages/man2/close.2.html
    pub fn close(fd: i32) -> i32;
    // https://man7.org/linux/man-pages/man2/epoll_ctl.2.html
    pub fn epoll_ctl(epfd: i32, op: i32, fd: i32, event: *mut super::Event) ->
    // https://man7.org/linux/man-pages/man2/epoll_wait.2.html
    pub fn epoll_wait(
        epfd: i32,
        events: *mut super::Event,
        maxevents: i32,
        timeout: i32,
    ) -> i32;
  }
}
```

4 Development

In Section 2.1, we saw that futures can only be awaited inside `async` blocks. This means that, in order to invoke our future from synchronous code, we need to implement a special intermediary layer. This is the so-called asynchronous runtime discussed in Section 1.

This was significantly more challenging than I'd initially envisioned. Understanding why certain things are implemented the way they are (particularly `Future` and `epoll`) took a lot of time, so hopefully the above explanations provide a good description of what I've learned. The provided code contains four modules. They modules, as well as their provided structs, are described below:

- `epoll`: provides the FFI and safe bindings for invoking the Linux epoll system calls. This also provides the `Events` struct, as described earlier.
- `blockers`: provides an `IoBlocker` class which interacts directly with the `epoll` module. It also provides a `Registrator`, which is used to register new events to `IoBlocker.`
- `services`: this implements a simple `TcpStream` reader, which enables asynchronous polling of TCP connections.
- `runtime`: this implements our `Task`, `Executor` and `Reactor`.
    - The Task is much like described in previous sections.
    - The Executor receives tasks from the Reactor and polls them to completion.
    - The Reactor receives I/O events, turns them into tasks and sends them to the Executor.

The code also provides a test showcasing functionality. It creates six TCP Streams, and connects them to a remote server that adds a delay to when they are received. They are then polled to completion. To run this, type `cargo test -- --nocapture` .

5 References

https://rust-lang.github.io/async-book/

https://cfsamson.github.io/book-exploring-async-basics/

https://cfsamsonbooks.gitbook.io/epoll-kqueue-iocp-explained/

https://os.phil-opp.com/async-await/

https://docs.rs/async-std/latest/async_std/index.html