Getting Started

Welcome to Asynchronous Programming in Rust! If yo
asynchronous Rust code, you've come to the right plac
server, a database, or an operating system, this book w
asynchronous programming tools to get the most out

What This Book Covers

This book aims to be a comprehensive, up-to-date guic
features and libraries, appropriate for beginners and c

- The early chapters provide an introduction to asy Rust's particular take on it.
- The middle chapters discuss key utilities and con writing async code, and describe best-practices fo applications to maximize performance and reusa
- The last section of the book covers the broader a number of examples of how to accomplish comm

With that out of the way, let's explore the exciting worl Rust!

Why Async?

We all love how Rust empowers us to write fast, safe s
programming fit into this vision?

Asynchronous programming, or async for short, is a *cc*
supported by an increasing number of programming l
number of concurrent tasks on a small number of OS t
look and feel of ordinary synchronous programming, t

Async vs other concurrency mode

Concurrent programming is less mature and "standarc
programming. As a result, we express concurrency diff
concurrent programming model the language is suppc
popular concurrency models can help you understand
within the broader field of concurrent programming:

- **OS threads** don't require any changes to the pro
  easy to express concurrency. However, synchron
  and the performance overhead is large. Thread p
  costs, but not enough to support massive IO-bou
- **Event-driven programming**, in conjunction with
  but tends to result in a verbose, "non-linear" con
  propagation is often hard to follow.
- **Coroutines**, like threads, don't require changes t
  makes them easy to use. Like async, they can als
  However, they abstract away low-level details tha
  programming and custom runtime implementors
- **The actor model** divides all concurrent computa
  communicate through fallible message passing, r
  actor model can be efficiently implemented, but i
  unanswered, such as flow control and retry logic.

In summary, asynchronous programming allows highl
are suitable for low-level languages like Rust, while prc
benefits of threads and coroutines.

Async in Rust vs other languages

Although asynchronous programming is supported in
across implementations. Rust's implementation of asy
few ways:

- **Futures are inert** in Rust and make progress on
  stops it from making further progress.
- **Async is zero-cost** in Rust, which means that you
  Specifically, you can use async without heap alloc
  great for performance! This also lets you use asyn
  as embedded systems.
- **No built-in runtime** is provided by Rust. Instead
  community maintained crates.
- **Both single- and multithreaded** runtimes are a
  strengths and weaknesses.

Async vs threads in Rust

The primary alternative to async in Rust is using OS thr
`std::thread` or indirectly through a thread pool. Migr
versa typically requires major refactoring work, both ir
are building a library) any exposed public interfaces. A
your needs early can save a lot of development time.

OS threads are suitable for a small number of tasks, s
memory overhead. Spawning and switching between t
threads consume system resources. A thread pool libr
costs, but not all. However, threads let you reuse existi
significant code changes—no particular programming
systems, you can also change the priority of a thread, v
latency sensitive applications.

Async provides significantly reduced CPU and memory
with a large amount of IO-bound tasks, such as servers
can have orders of magnitude more tasks than OS thre
a small amount of (expensive) threads to handle a larg
async Rust results in larger binary blobs due to the sta
functions and since each executable bundles an async

On a last note, asynchronous programming is not *bett*
don't need async for performance reasons, threads ca

Example: Concurrent downloading

In this example our goal is to download two web page
application we need to spawn threads to achieve conc

```
fn get_two_sites() {
    // Spawn two threads to do work.
    let thread_one = thread::spawn(|| download(
    let thread_two = thread::spawn(|| download(

    // Wait for both threads to complete.
    thread_one.join().expect("thread one panick
    thread_two.join().expect("thread two panick
}
```

However, downloading a web page is a small task; cre
of work is quite wasteful. For a larger application, it ca
Rust, we can run these tasks concurrently without ext

```
async fn get_two_sites_async() {
    // Create two different "futures" which, wh
    // will asynchronously download the webpage
    let future_one = download_async("https://ww
    let future_two = download_async("https://ww

    // Run both futures to completion at the sa
    join!(future_one, future_two);
}
```

Here, no extra threads are created. Additionally, all fu
and there are no heap allocations! However, we need
in the first place, which this book will help you achieve

Custom concurrency models in Ru

On a last note, Rust doesn't force you to choose betwe
both models within the same application, which can b
threaded and async dependencies. In fact, you can eve
altogether, such as event-driven programming, as lon
it.

The State of Asynchronou

Parts of async Rust are supported with the same stabil
Other parts are still maturing and will change over tim

- Outstanding runtime performance for typical cor
- More frequent interaction with advanced languag
  pinning.
- Some compatibility constraints, both between syı
  different async runtimes.
- Higher maintenance burden, due to the ongoing
  language support.

In short, async Rust is more difficult to use and can res
than synchronous Rust, but gives you best-in-class per
Rust are constantly improving, so the impact of these i

Language and library support

While asynchronous programming is supported by Ru
depend on functionality provided by community crate
mixture of language features and library support:

- The most fundamental traits, types and functions
  provided by the standard library.
- The `async/await` syntax is supported directly by
- Many utility types, macros and functions are prov
  be used in any async Rust application.
- Execution of async code, IO and task spawning aı
  as Tokio and async-std. Most async applications,
  specific runtime. See "The Async Ecosystem" sect

Some language features you may be used to from syn
async Rust. Notably, Rust does not let you declare asyı
need to use workarounds to achieve the same result, v

Compiling and debugging

For the most part, compiler- and runtime errors in asy
have always done in Rust. There are a few noteworthy

Compilation errors

Compilation errors in async Rust conform to the same
but since async Rust often depends on more complex
and pinning, you may encounter these types of errors

Runtime errors

Whenever the compiler encounters an async function,
the hood. Stack traces in async Rust typically contain d
well as function calls from the runtime. As such, interp
involved than it would be in synchronous Rust.

New failure modes

A few novel failure modes are possible in async Rust, f
function from an async context or if you implement th
can silently pass both the compiler and sometimes eve
understanding of the underlying concepts, which this b
avoid these pitfalls.

Compatibility considerations

Asynchronous and synchronous code cannot always b
can't directly call an async function from a sync functio
promote different design patterns, which can make it d
the different environments.

Even async code cannot always be combined freely. So
runtime to function. If so, it is usually specified in the c

These compatibility issues can limit your options, so m
runtime and what crates you may need early. Once yo
won't have to worry much about compatibility.

Performance characteristics

The performance of async Rust depends on the implem
using. Even though the runtimes that power async Rus
perform exceptionally well for most practical workloac

That said, most of the async ecosystem assumes a *mu*
difficult to enjoy the theoretical performance benefits
namely cheaper synchronization. Another overlooked
which are important for drivers, GUI applications and s
and/or OS support in order to be scheduled appropria
support for these use cases in the future.

async/.await Primer

`async`/`.await` is Rust's built-in tool for writing asynch
synchronous code. `async` transforms a block of code
a trait called `Future`. Whereas calling a blocking functi
block the whole thread, blocked `Future`s will yield cor
`Future`s to run.

Let's add some dependencies to the `Cargo.toml` file:

```
[dependencies]
futures = "0.3"
```

To create an asynchronous function, you can use the

```
async fn do_something() { /* ... */ }
```

The value returned by `async fn` is a `Future`. For anyt
be run on an executor.

```
// `block_on` blocks the current thread until t
// completion. Other executors provide more com
// multiple futures onto the same thread.
use futures::executor::block_on;

async fn hello_world() {
    println!("hello, world!");
}

fn main() {
    let future = hello_world(); // Nothing is p
    block_on(future); // `future` is run and "h
}
```

Inside an `async fn`, you can use `.await` to wait for th
implements the `Future` trait, such as the output of an
`.await` doesn't block the current thread, but instead a
complete, allowing other tasks to run if the future is cu

For example, imagine that we have three `async fn`: `l`

```
async fn learn_song() -> Song { /* ... */ }
async fn sing_song(song: Song) { /* ... */ }
async fn dance() { /* ... */ }
```

One way to do learn, sing, and dance would be to bloc

```
fn main() {
    let song = block_on(learn_song());
    block_on(sing_song(song));
    block_on(dance());
}
```

However, we're not giving the best performance possil one thing at once! Clearly we have to learn the song be dance at the same time as learning and singing the sor separate `async fn` which can be run concurrently:

```
async fn learn_and_sing() {
    // Wait until the song has been learned bef
    // We use `.await` here rather than `block_
    // thread, which makes it possible to `danc
    let song = learn_song().await;
    sing_song(song).await;
}

async fn async_main() {
    let f1 = learn_and_sing();
    let f2 = dance();

    // `join!` is like `.await` but can wait fo
    // If we're temporarily blocked in the `lea
    // future will take over the current thread
    // `learn_and_sing` can take back over. If
    // `async_main` is blocked and will yield t
    futures::join!(f1, f2);
}

fn main() {
    block_on(async_main());
}
```

In this example, learning the song must happen before and singing can happen at the same time as dancing. I rather than `learn_song().await` in `learn_and_sing`, anything else while `learn_song` was running. This wou same time. By `.await`-ing the `learn_song` future, we current thread if `learn_song` is blocked. This makes it completion concurrently on the same thread.

Under the Hood: Executin Tasks

In this section, we'll cover the underlying structure of h are scheduled. If you're only interested in learning hov existing `Future` types and aren't interested in the deta can skip ahead to the `async` / `await` chapter. However this chapter are useful for understanding how `async` / the runtime and performance properties of `async` / `aw` asynchronous primitives. If you decide to skip this sect it to revisit in the future.

Now, with that out of the way, let's talk about the `Futu`

The Future Trait

The `Future` trait is at the center of asynchronous prog
asynchronous computation that can produce a value (
e.g. `()`). A *simplified* version of the future trait might l

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

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

Futures can be advanced by calling the `poll` function,
towards completion as possible. If the future complete
the future is not able to complete yet, it returns `Poll:`
`wake()` function to be called when the `Future` is reac
`wake()` is called, the executor driving the `Future` will
can make more progress.

Without `wake()` , the executor would have no way of k
make progress, and would have to be constantly pollir
executor knows exactly which futures are ready to be

For example, consider the case where we want to reac
have data available already. If there is data, we can rea
`Poll::Ready(data)` , but if no data is ready, our future
progress. When no data is available, we must register
ready on the socket, which will tell the executor that ou
simple `SocketRead` future might look something like t

```
pub struct SocketRead<'a> {
    socket: &'a Socket,
}

impl SimpleFuture for SocketRead<'_> {
    type Output = Vec<u8>;

    fn poll(&mut self, wake: fn()) -> Poll<Self
        if self.socket.has_data_to_read() {
            // The socket has data -- read it i
            Poll::Ready(self.socket.read_buf())
        } else {
            // The socket does not yet have dat
            //
            // Arrange for `wake` to be called
            // When data becomes available, `wa
            // user of this `Future` will know
            // receive data.
            self.socket.set_readable_callback(w
            Poll::Pending
        }
    }
}
```

This model of `Future` s allows for composing together without needing intermediate allocations. Running mu futures together can be implemented via allocation-fre

```
/// A SimpleFuture that runs two other futures
///
/// Concurrency is achieved via the fact that c
/// may be interleaved, allowing each future to
pub struct Join<FutureA, FutureB> {
    // Each field may contain a future that sho
    // If the future has already completed, the
    // This prevents us from polling a future a
    // would violate the contract of the `Futur
    a: Option<FutureA>,
    b: Option<FutureB>,
}

impl<FutureA, FutureB> SimpleFuture for Join<Fu
where
    FutureA: SimpleFuture<Output = ()>,
    FutureB: SimpleFuture<Output = ()>,
{
    type Output = ();
    fn poll(&mut self, wake: fn()) -> Poll<Self
        // Attempt to complete future `a`.
        if let Some(a) = &mut self.a {
            if let Poll::Ready(()) = a.poll(wak
                self.a.take();
            }
        }

        // Attempt to complete future `b`.
        if let Some(b) = &mut self.b {
            if let Poll::Ready(()) = b.poll(wak
                self.b.take();
            }
        }

        if self.a.is_none() && self.b.is_none()
            // Both futures have completed -- w
            Poll::Ready(())
        } else {
            // One or both futures returned `Po
            // work to do. They will call `wake
            Poll::Pending
        }
    }
}
```

This shows how multiple futures can be run simultane
allocations, allowing for more efficient asynchronous p
sequential futures can be run one after another, like th

```
/// A SimpleFuture that runs two futures to com
//
// Note: for the purposes of this simple exampl
// the first and second futures are available a
// `AndThen` combinator allows creating the sec
// of the first future, like `get_breakfast.and
pub struct AndThenFut<FutureA, FutureB> {
    first: Option<FutureA>,
    second: FutureB,
}

impl<FutureA, FutureB> SimpleFuture for AndThen
where
    FutureA: SimpleFuture<Output = ()>,
    FutureB: SimpleFuture<Output = ()>,
{
    type Output = ();
    fn poll(&mut self, wake: fn()) -> Poll<Self
        if let Some(first) = &mut self.first {
            match first.poll(wake) {
                // We've completed the first fu
                // the second!
                Poll::Ready(()) => self.first.t
                // We couldn't yet complete the
                Poll::Pending => return Poll::P
            };
        }
        // Now that the first future is done, a
        self.second.poll(wake)
    }
}
```

These examples show how the `Future` trait can be us
flow without requiring multiple allocated objects and c
control-flow out of the way, let's talk about the real `Fu`

```
trait Future {
    type Output;
    fn poll(
        // Note the change from `&mut self` to
        self: Pin<&mut Self>,
        // and the change from `wake: fn()` to
        cx: &mut Context<'_>,
    ) -> Poll<Self::Output>;
}
```

The first change you'll notice is that our `self` type is n
to `Pin<&mut Self>`. We'll talk more about pinning in a
allows us to create futures that are immovable. Immov
between their fields, e.g. `struct MyFut { a: i32, ptr`
necessary to enable async/await.

Secondly, `wake: fn()` has changed to `&mut Context<` to a function pointer (`fn()`) to tell the future executor polled. However, since `fn()` is just a function pointer, `Future` called `wake` .

In a real-world scenario, a complex application like a w different connections whose wakeups should all be ma solves this by providing access to a value of type `Waker` specific task.

Task Wakeups with Waker

It's common that futures aren't able to complete the fi
happens, the future needs to ensure that it is polled ag
progress. This is done with the `Waker` type.

Each time a future is polled, it is polled as part of a "tas
that have been submitted to an executor.

`Waker` provides a `wake()` method that can be used to
task should be awoken. When `wake()` is called, the ex
with the `Waker` is ready to make progress, and its futu

`Waker` also implements `clone()` so that it can be cop

Let's try implementing a simple timer future using `Wak

Applied: Build a Timer

For the sake of the example, we'll just spin up a new th
for the required time, and then signal the timer future

First, start a new project with `cargo new --lib timer_`
need to get started to `src/lib.rs`:

```
use std::{
    future::Future,
    pin::Pin,
    sync::{Arc, Mutex},
    task::{Context, Poll, Waker},
    thread,
    time::Duration,
};
```

Let's start by defining the future type itself. Our future
communicate that the timer has elapsed and the futur
`Arc<Mutex<..>>` value to communicate between the t

```
pub struct TimerFuture {
    shared_state: Arc<Mutex<SharedState>>,
}

/// Shared state between the future and the wai
struct SharedState {
    /// Whether or not the sleep time has elaps
    completed: bool,

    /// The waker for the task that `TimerFutur
    /// The thread can use this after setting `
    /// `TimerFuture`'s task to wake up, see th
    /// move forward.
    waker: Option<Waker>,
}
```

Now, let's actually write the `Future` implementation!

```
impl Future for TimerFuture {
    type Output = ();
    fn poll(self: Pin<&mut Self>, cx: &mut Cont
        // Look at the shared state to see if t
        let mut shared_state = self.shared_stat
        if shared_state.completed {
            Poll::Ready(())
        } else {
            // Set waker so that the thread can
            // when the timer has completed, en
            // again and sees that `completed =
            //
            // It's tempting to do this once ra
            // the waker each time. However, th
            // tasks on the executor, which cou
            // to the wrong task, preventing `T
            // correctly.
            //
            // N.B. it's possible to check for
`Waker::will_wake`
            // function, but we omit that here
            shared_state.waker = Some(cx.waker(
            Poll::Pending
        }
    }
}
```

Pretty simple, right? If the thread has set `shared_stat` Otherwise, we clone the `Waker` for the current task an that the thread can wake the task back up.

Importantly, we have to update the `Waker` every time future may have moved to a different task with a differ futures are passed around between tasks after being p

Finally, we need the API to actually construct the timer

```
impl TimerFuture {
    /// Create a new `TimerFuture` which will c
    /// timeout.
    pub fn new(duration: Duration) -> Self {
        let shared_state = Arc::new(Mutex::new(
            completed: false,
            waker: None,
        }));

        // Spawn the new thread
        let thread_shared_state = shared_state.
        thread::spawn(move || {
            thread::sleep(duration);
            let mut shared_state = thread_share
            // Signal that the timer has comple
            // task on which the future was pol
            shared_state.completed = true;
            if let Some(waker) = shared_state.w
                waker.wake()
            }
        });

        TimerFuture { shared_state }
    }
}
```

Woot! That's all we need to build a simple timer future
run the future on...

Applied: Build an Executo

Rust's `Future` s are lazy: they won't do anything unless
way to drive a future to completion is to `.await` it insi
pushes the problem one level up: who will run the futu
`async` functions? The answer is that we need a `Futur`

`Future` executors take a set of top-level `Future` s and
`poll` whenever the `Future` can make progress. Typic
once to start off. When `Future` s indicate that they are
`wake()` , they are placed back onto a queue and `poll`
`Future` has completed.

In this section, we'll write our own simple executor cap
top-level futures to completion concurrently.

For this example, we depend on the `futures` crate for
easy way to construct a `Waker` . Edit `Cargo.toml` to ad

```
[package]
name = "timer_future"
version = "0.1.0"
authors = ["XYZ Author"]
edition = "2021"

[dependencies]
futures = "0.3"
```

Next, we need the following imports at the top of `src/`

```
use futures::{
    future::{BoxFuture, FutureExt},
    task::{waker_ref, ArcWake},
};
use std::{
    future::Future,
    sync::mpsc::{sync_channel, Receiver, SyncSe
    sync::{Arc, Mutex},
    task::Context,
    time::Duration,
};
// The timer we wrote in the previous section:
use timer_future::TimerFuture;
```

Our executor will work by sending tasks to run over a c
off of the channel and run them. When a task is ready
schedule itself to be polled again by putting itself back

In this design, the executor itself just needs the receivi
will get a sending end so that they can spawn new futu
that can reschedule themselves, so we'll store them as
the task can use to requeue itself.

```
/// Task executor that receives tasks off of a
struct Executor {
    ready_queue: Receiver<Arc<Task>>,
}

/// `Spawner` spawns new futures onto the task
#[derive(Clone)]
struct Spawner {
    task_sender: SyncSender<Arc<Task>>,
}

/// A future that can reschedule itself to be p
struct Task {
    /// In-progress future that should be pushe
    ///
    /// The `Mutex` is not necessary for correc
    /// one thread executing tasks at once. How
    /// enough to know that `future` is only mu
    /// so we need to use the `Mutex` to prove
    /// executor would not need this, and could
    future: Mutex<Option<BoxFuture<'static, ()>

    /// Handle to place the task itself back on
    task_sender: SyncSender<Arc<Task>>,
}

fn new_executor_and_spawner() -> (Executor, Spa
    // Maximum number of tasks to allow queuein
    // This is just to make `sync_channel` happ
    // a real executor.
    const MAX_QUEUED_TASKS: usize = 10_000;
    let (task_sender, ready_queue) = sync_chann
    (Executor { ready_queue }, Spawner { task_s
}
```

Let's also add a method to spawner to make it easy to
take a future type, box it, and create a new `Arc<Task>`
onto the executor.

```
impl Spawner {
    fn spawn(&self, future: impl Future<Output
        let future = future.boxed();
        let task = Arc::new(Task {
            future: Mutex::new(Some(future)),
            task_sender: self.task_sender.clone
        });
        self.task_sender.send(task).expect("too
    }
}
```

To poll futures, we'll need to create a `Waker`. As discus `Waker` s are responsible for scheduling a task to be po Remember that `Waker` s tell the executor exactly whicl them to poll just the futures that are ready to make pr new `Waker` is by implementing the `ArcWake` trait and `.into_waker()` functions to turn an `Arc<impl ArcWak` `ArcWake` for our tasks to allow them to be turned into

```
impl ArcWake for Task {
    fn wake_by_ref(arc_self: &Arc<Self>) {
        // Implement `wake` by sending this tas
        // so that it will be polled again by t
        let cloned = arc_self.clone();
        arc_self
            .task_sender
            .send(cloned)
            .expect("too many tasks queued");
    }
}
```

When a `Waker` is created from an `Arc<Task>`, calling `Arc` to be sent onto the task channel. Our executor th it. Let's implement that:

```
impl Executor {
    fn run(&self) {
        while let Ok(task) = self.ready_queue.r
            // Take the future, and if it has n
Some),
            // poll it in an attempt to complet
            let mut future_slot = task.future.l
            if let Some(mut future) = future_sl
                // Create a `LocalWaker` from t
                let waker = waker_ref(&task);
                let context = &mut Context::fro
                // `BoxFuture<T>` is a type ali
                // `Pin<Box<dyn Future<Output =
                // We can get a `Pin<&mut dyn F
                // from it by calling the `Pin:
                if future.as_mut().poll(context
                    // We're not done processin
                    // back in its task to be r
                    *future_slot = Some(future)
                }
            }
        }
    }
}
```

Congratulations! We now have a working futures execu `async/.await` code and custom futures, such as the

```
fn main() {
    let (executor, spawner) = new_executor_and_

    // Spawn a task to print before and after w
    spawner.spawn(async {
        println!("howdy!");
        // Wait for our timer future to complet
        TimerFuture::new(Duration::new(2, 0)).a
        println!("done!");
    });

    // Drop the spawner so that our executor kn
    // receive more incoming tasks to run.
    drop(spawner);

    // Run the executor until the task queue is
    // This will print "howdy!", pause, and the
    executor.run();
}
```

Executors and System IO

In the previous section on The `Future` Trait, we discus
performed an asynchronous read on a socket:

```
pub struct SocketRead<'a> {
    socket: &'a Socket,
}

impl SimpleFuture for SocketRead<'_> {
    type Output = Vec<u8>;

    fn poll(&mut self, wake: fn()) -> Poll<Self
        if self.socket.has_data_to_read() {
            // The socket has data -- read it i
            Poll::Ready(self.socket.read_buf())
        } else {
            // The socket does not yet have dat
            //
            // Arrange for `wake` to be called
            // When data becomes available, `wa
            // user of this `Future` will know
            // receive data.
            self.socket.set_readable_callback(w
            Poll::Pending
        }
    }
}
```

This future will read available data on a socket, and if r
executor, requesting that its task be awoken when the
However, it's not clear from this example how the `Soc`
particular it isn't obvious how the `set_readable_callb`
arrange for `wake()` to be called once the socket becon
have a thread that continually checks whether `socket`
appropriate. However, this would be quite inefficient, r
blocked IO future. This would greatly reduce the efficie

In practice, this problem is solved through integration
primitive, such as `epoll` on Linux, `kqueue` on FreeBSI
`port`s on Fuchsia (all of which are exposed through th
These primitives all allow a thread to block on multiple
once one of the events completes. In practice, these AI

```
struct IoBlocker {
    /* ... */
}

struct Event {
    // An ID uniquely identifying the event tha
    id: usize,

    // A set of signals to wait for, or which o
    signals: Signals,
}

impl IoBlocker {
    /// Create a new collection of asynchronous
    fn new() -> Self { /* ... */ }

    /// Express an interest in a particular IO
    fn add_io_event_interest(
        &self,

        /// The object on which the event will
        io_object: &IoObject,

        /// A set of signals that may appear on
        /// which an event should be triggered,
        /// an ID to give to events that result
        event: Event,
    ) { /* ... */ }

    /// Block until one of the events occurs.
    fn block(&self) -> Event { /* ... */ }
}

let mut io_blocker = IoBlocker::new();
io_blocker.add_io_event_interest(
    &socket_1,
    Event { id: 1, signals: READABLE },
);
io_blocker.add_io_event_interest(
    &socket_2,
    Event { id: 2, signals: READABLE | WRITABLE
);
let event = io_blocker.block();

// prints e.g. "Socket 1 is now READABLE" if so
println!("Socket {:?} is now {:?}", event.id, e
```

Futures executors can use these primitives to provide sockets that can configure callbacks to be run when a of our `SocketRead` example above, the `Socket::set_` look like the following pseudocode:

```
impl Socket {
    fn set_readable_callback(&self, waker: Wake
        // `local_executor` is a reference to t
        // this could be provided at creation o
        // many executor implementations pass i
        // storage for convenience.
        let local_executor = self.local_executo

        // Unique ID for this IO object.
        let id = self.id;

        // Store the local waker in the executo
        // once the IO event arrives.
        local_executor.event_map.insert(id, wak
        local_executor.add_io_event_interest(
            &self.socket_file_descriptor,
            Event { id, signals: READABLE },
        );
    }
}
```

We can now have just one executor thread which can r the appropriate `Waker`, which will wake up the corresp drive more tasks to completion before returning to che continues...).

async/.await

In the first chapter, we took a brief look at `async`/`.aw` `async`/`.await` in greater detail, explaining how it wor traditional Rust programs.

`async`/`.await` are special pieces of Rust syntax that r current thread rather than blocking, allowing other coo an operation to complete.

There are two main ways to use `async`: `async fn` anc that implements the `Future` trait:

```
// `foo()` returns a type that implements `Futu
// `foo().await` will result in a value of type
async fn foo() -> u8 { 5 }

fn bar() -> impl Future<Output = u8> {
    // This `async` block results in a type tha
    // `Future<Output = u8>`.
    async {
        let x: u8 = foo().await;
        x + 5
    }
}
```

As we saw in the first chapter, `async` bodies and othe until they are run. The most common way to run a `Fu` called on a `Future`, it will attempt to run it to completi yield control of the current thread. When more progre picked up by the executor and will resume running, all

async Lifetimes

Unlike traditional functions, `async fn`s which take ref arguments return a `Future` which is bounded by the l

```
// This function:
async fn foo(x: &u8) -> u8 { *x }

// Is equivalent to this function:
fn foo_expanded<'a>(x: &'a u8) -> impl Future<O
    async move { *x }
}
```

This means that the future returned from an `async fr` `'static` arguments are still valid. In the common case immediately after calling the function (as in `foo(&x).a` storing the future or sending it over to another task or

One common workaround for turning an `async fn` wi `'static` future is to bundle the arguments with the ca block:

```
fn bad() -> impl Future<Output = u8> {
    let x = 5;
    borrow_x(&x) // ERROR: `x` does not live lo
}

fn good() -> impl Future<Output = u8> {
    async {
        let x = 5;
        borrow_x(&x).await
    }
}
```

By moving the argument into the `async` block, we ext `Future` returned from the call to `good`.

`async move`

`async` blocks and closures allow the `move` keyword, m `move` block will take ownership of the variables it refer current scope, but giving up the ability to share those v

```
/// `async` block:
///
/// Multiple different `async` blocks can acces
/// so long as they're executed within the vari
async fn blocks() {
    let my_string = "foo".to_string();

    let future_one = async {
        // ...
        println!("{my_string}");
    };

    let future_two = async {
        // ...
        println!("{my_string}");
    };

    // Run both futures to completion, printing
    let ((), ()) = futures::join!(future_one, f
}

/// `async move` block:
///
/// Only one `async move` block can access the
/// captures are moved into the `Future` genera
/// However, this allows the `Future` to outliv
/// variable:
fn move_block() -> impl Future<Output = ()> {
    let my_string = "foo".to_string();
    async move {
        // ...
        println!("{my_string}");
    }
}
```

`.await`ing on a Multithreaded Exe

Note that, when using a multithreaded `Future` execut threads, so any variables used in `async` bodies must b any `.await` can potentially result in a switch to a new

This means that it is not safe to use `Rc`, `&RefCell` or a the `Send` trait, including references to types that don't

(Caveat: it is possible to use these types as long as they `.await`.)

Similarly, it isn't a good idea to hold a traditional non-f as it can cause the threadpool to lock up: one task cou

the executor, allowing another task to attempt to take
avoid this, use the `Mutex` in `futures::lock` rather tha

Pinning

To poll futures, they must be pinned using a special ty|
explanation of the `Future` trait in the previous sectior
you'll recognize `Pin` from the `self: Pin<&mut Self>`
definition. But what does it mean, and why do we need

Why Pinning

`Pin` works in tandem with the `Unpin` marker. Pinning
an object implementing `!Unpin` won't ever be moved.
we need to remember how `async` / `.await` works. Cor

```
let fut_one = /* ... */;
let fut_two = /* ... */;
async move {
    fut_one.await;
    fut_two.await;
}
```

Under the hood, this creates an anonymous type that
`poll` method that looks something like this:

```
// The `Future` type generated by our `async {
struct AsyncFuture {
    fut_one: FutOne,
    fut_two: FutTwo,
    state: State,
}

// List of states our `async` block can be in
enum State {
    AwaitingFutOne,
    AwaitingFutTwo,
    Done,
}

impl Future for AsyncFuture {
    type Output = ();

    fn poll(mut self: Pin<&mut Self>, cx: &mut
        loop {
            match self.state {
                State::AwaitingFutOne => match
                    Poll::Ready(()) => self.sta
                    Poll::Pending => return Pol
                }
                State::AwaitingFutTwo => match
                    Poll::Ready(()) => self.sta
                    Poll::Pending => return Pol
                }
                State::Done => return Poll::Rea
            }
        }
    }
}
```

When `poll` is first called, it will poll `fut_one`. If `fut_o` `AsyncFuture::poll` will return. Future calls to `poll` w off. This process continues until the future is able to su

However, what happens if we have an `async` block th

```
async {
    let mut x = [0; 128];
    let read_into_buf_fut = read_into_buf(&mut
    read_into_buf_fut.await;
    println!("{:?}", x);
}
```

What struct does this compile down to?

```
struct ReadIntoBuf<'a> {
    buf: &'a mut [u8], // points to `x` below
}

struct AsyncFuture {
    x: [u8; 128],
    read_into_buf_fut: ReadIntoBuf<'what_lifeti
}
```

Here, the `ReadIntoBuf` future holds a reference into tl However, if `AsyncFuture` is moved, the location of `x` \ pointer stored in `read_into_buf_fut.buf`.

Pinning futures to a particular spot in memory prevent create references to values inside an `async` block.

Pinning in Detail

Let's try to understand pinning by using an slightly sim encounter above is a problem that ultimately boils dov self-referential types in Rust.

For now our example will look like this:

```
#[derive(Debug)]
struct Test {
    a: String,
    b: *const String,
}

impl Test {
    fn new(txt: &str) -> Self {
        Test {
            a: String::from(txt),
            b: std::ptr::null(),
        }
    }

    fn init(&mut self) {
        let self_ref: *const String = &self.a;
        self.b = self_ref;
    }

    fn a(&self) -> &str {
        &self.a
    }

    fn b(&self) -> &String {
        assert!(!self.b.is_null(), "Test::b cal
called first");
        unsafe { &*(self.b) }
    }
}
```

`Test` provides methods to get a reference to the valu
reference to `a` we store it as a pointer since the borro
define this lifetime. We now have what we call a self-re

Our example works fine if we don't move any of our da
running this example:

```
fn main() {
    let mut test1 = Test::new("test1");
    test1.init();
    let mut test2 = Test::new("test2");
    test2.init();

    println!("a: {}, b: {}", test1.a(), test1.b
    println!("a: {}, b: {}", test2.a(), test2.b

}
```

We get what we'd expect:

```
a: test1, b: test1
a: test2, b: test2
```

Let's see what happens if we swap `test1` with `test2`

```
fn main() {
    let mut test1 = Test::new("test1");
    test1.init();
    let mut test2 = Test::new("test2");
    test2.init();

    println!("a: {}, b: {}", test1.a(), test1.b
    std::mem::swap(&mut test1, &mut test2);
    println!("a: {}, b: {}", test2.a(), test2.b

}
```

Naively, we could think that what we should get a debu

```
a: test1, b: test1
a: test1, b: test1
```

But instead we get:

```
a: test1, b: test1
a: test1, b: test2
```

The pointer to `test2.b` still points to the old location w
is not self-referential anymore, it holds a pointer to a fi
we can't rely on the lifetime of `test2.b` to be tied to tl

If you're still not convinced, this should at least convinc

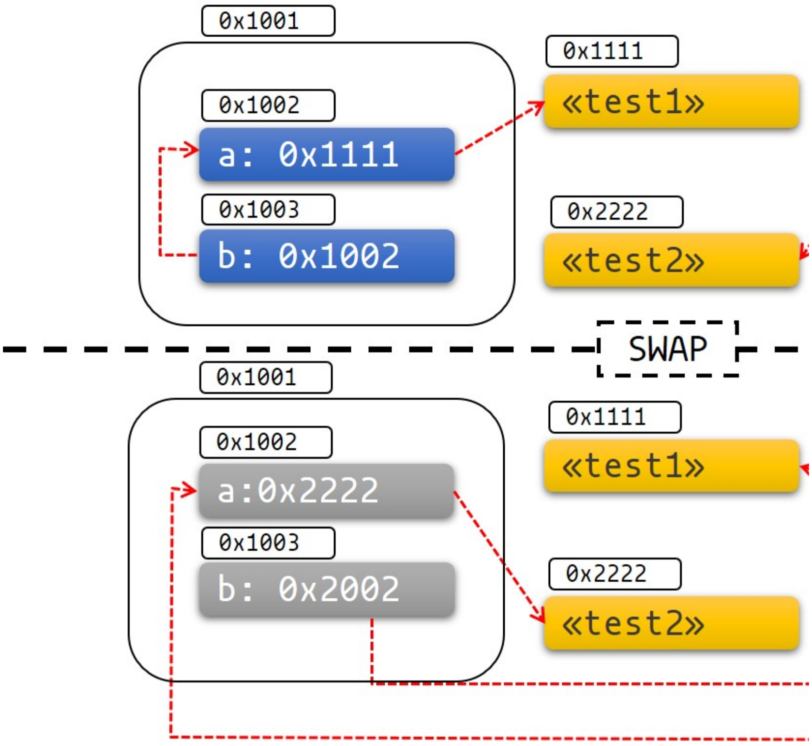
```
fn main() {
    let mut test1 = Test::new("test1");
    test1.init();
    let mut test2 = Test::new("test2");
    test2.init();

    println!("a: {}, b: {}", test1.a(), test1.b
    std::mem::swap(&mut test1, &mut test2);
    test1.a = "I've totally changed now!".to_st
    println!("a: {}, b: {}", test2.a(), test2.b

}
```

The diagram below can help visualize what's going on:

Fig 1: Before and after swap



It's easy to get this to show undefined behavior and fa

Pinning in Practice

Let's see how pinning and the `Pin` type can help us sc

The `Pin` type wraps pointer types, guaranteeing that t
moved if it is not implementing `Unpin`. For example, `P`
all guarantee that `T` won't be moved if `T: !Unpin`.

Most types don't have a problem being moved. These
Pointers to `Unpin` types can be freely placed into or ta
`Unpin`, so `Pin<&mut u8>` behaves just like a normal `&`

However, types that can't be moved after they're pinne
Futures created by async/await is an example of this.

Pinning to the Stack

Back to our example. We can solve our problem by usi
example would look like if we required a pinned pointe

```
use std::pin::Pin;
use std::marker::PhantomPinned;

#[derive(Debug)]
struct Test {
    a: String,
    b: *const String,
    _marker: PhantomPinned,
}


impl Test {
    fn new(txt: &str) -> Self {
        Test {
            a: String::from(txt),
            b: std::ptr::null(),
            _marker: PhantomPinned, // This mak
        }
    }

    fn init(self: Pin<&mut Self>) {
        let self_ptr: *const String = &self.a;
        let this = unsafe { self.get_unchecked_
        this.b = self_ptr;
    }

    fn a(self: Pin<&Self>) -> &str {
        &self.get_ref().a
    }

    fn b(self: Pin<&Self>) -> &String {
        assert!(!self.b.is_null(), "Test::b cal
called first");
        unsafe { &*(self.b) }
    }
}
```

Pinning an object to the stack will always be `unsafe` if
can use a crate like `pin_utils` to avoid writing our ow
stack.

Below, we pin the objects `test1` and `test2` to the sta

```
pub fn main() {
    // test1 is safe to move before we initiali
    let mut test1 = Test::new("test1");
    // Notice how we shadow `test1` to prevent
    let mut test1 = unsafe { Pin::new_unchecked
    Test::init(test1.as_mut());

    let mut test2 = Test::new("test2");
    let mut test2 = unsafe { Pin::new_unchecked
    Test::init(test2.as_mut());

    println!("a: {}, b: {}", Test::a(test1.as_r
    println!("a: {}, b: {}", Test::a(test2.as_r
}
```

Now, if we try to move our data now we get a compilat

```
pub fn main() {
    let mut test1 = Test::new("test1");
    let mut test1 = unsafe { Pin::new_unchecked
    Test::init(test1.as_mut());

    let mut test2 = Test::new("test2");
    let mut test2 = unsafe { Pin::new_unchecked
    Test::init(test2.as_mut());

    println!("a: {}, b: {}", Test::a(test1.as_r
    std::mem::swap(test1.get_mut(), test2.get_m
    println!("a: {}, b: {}", Test::a(test2.as_r
}
```

The type system prevents us from moving the data, as

```
error[E0277]: `PhantomPinned` cannot be unpinne
   --> src\test.rs:56:30
    |
56  |         std::mem::swap(test1.get_mut(), t
    |                              ^^^^^^^ with
`Unpin` is not implemented for `PhantomPinned`
    |
    = note: consider using `Box::pin`
note: required because it appears within the ty
   --> src\test.rs:7:8
    |
7   | struct Test {
    |        ^^^^
note: required by a bound in `std::pin::Pin::<&
   --> <...>rustlib/src/rust\library\core\src\p
    |
748 |         T: Unpin,
    |            ^^^^^ required by this bound i
T>::get_mut`
```

It's important to note that stack pinning will always
writing `unsafe`. While we know that the *pointee* of
lifetime of `'a` we can't know if the data `&'a mut T`
ends. If it does it will violate the Pin contract.

A mistake that is easy to make is forgetting to shado
could drop the `Pin` and move the data after `&'a mu`
violates the Pin contract):

```
fn main() {
    let mut test1 = Test::new("test1");
    let mut test1_pin = unsafe { Pin::new_unch
    Test::init(test1_pin.as_mut());

    drop(test1_pin);
    println!(r#"test1.b points to "test1": {:

    let mut test2 = Test::new("test2");
    mem::swap(&mut test1, &mut test2);
    println!("... and now it points nowhere:
}
```

Pinning to the Heap

Pinning an `!Unpin` type to the heap gives our data a s
data we point to can't move after it's pinned. In contra
data will be pinned for the lifetime of the object.

```
use std::pin::Pin;
use std::marker::PhantomPinned;

#[derive(Debug)]
struct Test {
    a: String,
    b: *const String,
    _marker: PhantomPinned,
}

impl Test {
    fn new(txt: &str) -> Pin<Box<Self>> {
        let t = Test {
            a: String::from(txt),
            b: std::ptr::null(),
            _marker: PhantomPinned,
        };
        let mut boxed = Box::pin(t);
        let self_ptr: *const String = &boxed.a;
        unsafe { boxed.as_mut().get_unchecked_m

        boxed
    }

    fn a(self: Pin<&Self>) -> &str {
        &self.get_ref().a
    }

    fn b(self: Pin<&Self>) -> &String {
        unsafe { &*(self.b) }
    }
}

pub fn main() {
    let test1 = Test::new("test1");
    let test2 = Test::new("test2");

    println!("a: {}, b: {}",test1.as_ref().a(),
    println!("a: {}, b: {}",test2.as_ref().a(),
}
```

Some functions require the futures they work with to b `Stream` that isn't `Unpin` with a function that requires the value using either `Box::pin` (to create a `Pin<Box<` macro (to create a `Pin<&mut T>`). `Pin<Box<Fut>>` and futures, and both implement `Unpin`.

For example:

```
use pin_utils::pin_mut; // `pin_utils` is a han

// A function which takes a `Future` that imple
fn execute_unpin_future(x: impl Future<Output =

let fut = async { /* ... */ };
execute_unpin_future(fut); // Error: `fut` does

// Pinning with `Box`:
let fut = async { /* ... */ };
let fut = Box::pin(fut);
execute_unpin_future(fut); // OK

// Pinning with `pin_mut!`:
let fut = async { /* ... */ };
pin_mut!(fut);
execute_unpin_future(fut); // OK
```

Summary

1. If `T: Unpin` (which is the default), then `Pin<'a,` `T`. In other words: `Unpin` means it's OK for this t so `Pin` will have no effect on such a type.
2. Getting a `&mut T` to a pinned T requires unsafe i
3. Most standard library types implement `Unpin`. T you encounter in Rust. A `Future` generated by as
4. You can add a `!Unpin` bound on a type on nightl `std::marker::PhantomPinned` to your type on st
5. You can either pin data to the stack or to the hea
6. Pinning a `!Unpin` object to the stack requires `un`
7. Pinning a `!Unpin` object to the heap does not re doing this using `Box::pin`.
8. For pinned data where `T: !Unpin` you have to m will not get invalidated or repurposed *from the m* called. This is an important part of the *pin contrac*

The Stream Trait

The `Stream` trait is similar to `Future` but can yield mu
similar to the `Iterator` trait from the standard library

```
trait Stream {
    /// The type of the value yielded by the st
    type Item;

    /// Attempt to resolve the next item in the
    /// Returns `Poll::Pending` if not ready, `
    /// is ready, and `Poll::Ready(None)` if th
    fn poll_next(self: Pin<&mut Self>, cx: &mut
        -> Poll<Option<Self::Item>>;
}
```

One common example of a `Stream` is the `Receiver` fo
crate. It will yield `Some(val)` every time a value is sent
`None` once the `Sender` has been dropped and all pen

```
async fn send_recv() {
    const BUFFER_SIZE: usize = 10;
    let (mut tx, mut rx) = mpsc::channel::<i32>

    tx.send(1).await.unwrap();
    tx.send(2).await.unwrap();
    drop(tx);

    // `StreamExt::next` is similar to `Iterato
    // type that implements `Future<Output = Op
    assert_eq!(Some(1), rx.next().await);
    assert_eq!(Some(2), rx.next().await);
    assert_eq!(None, rx.next().await);
}
```

Iteration and Concurrenc

Similar to synchronous `Iterator` s, there are many dif process the values in a `Stream` . There are combinator and `fold` , and their early-exit-on-error cousins `try_m`

Unfortunately, `for` loops are not usable with `Stream` s let and the `next` / `try_next` functions can be used:

```
async fn sum_with_next(mut stream: Pin<&mut dyn
    use futures::stream::StreamExt; // for `nex
    let mut sum = 0;
    while let Some(item) = stream.next().await
        sum += item;
    }
    sum
}

async fn sum_with_try_next(
    mut stream: Pin<&mut dyn Stream<Item = Resu
) -> Result<i32, io::Error> {
    use futures::stream::TryStreamExt; // for `
    let mut sum = 0;
    while let Some(item) = stream.try_next().aw
        sum += item;
    }
    Ok(sum)
}
```

However, if we're just processing one element at a tim opportunity for concurrency, which is, after all, why we place. To process multiple items from a stream concur and `try_for_each_concurrent` methods:

```
async fn jump_around(
    mut stream: Pin<&mut dyn Stream<Item = Resu
) -> Result<(), io::Error> {
    use futures::stream::TryStreamExt; // for `
    const MAX_CONCURRENT_JUMPERS: usize = 100;

    stream.try_for_each_concurrent(MAX_CONCURRE
        jump_n_times(num).await?;
        report_n_jumps(num).await?;
        Ok(())
    }).await?;

    Ok(())
}
```

Executing Multiple Future

Up until now, we've mostly executed futures by using
until a particular `Future` completes. However, real asy
execute several different operations concurrently.

In this chapter, we'll cover some ways to execute multi
same time:

- `join!` : waits for futures to all complete
- `select!` : waits for one of several futures to com
- Spawning: creates a top-level task which ambient
- `FuturesUnordered` : a group of futures which yiel

join!

The `futures::join` macro makes it possible to wait fc
complete while executing them all concurrently.

join!

When performing multiple asynchronous operations, i
in a series:

```
async fn get_book_and_music() -> (Book, Music)
    let book = get_book().await;
    let music = get_music().await;
    (book, music)
}
```

However, this will be slower than necessary, since it wo
after `get_book` has completed. In some other languag
completion, so two operations can be run concurrently
start the futures, and then awaiting them both:

```
// WRONG -- don't do this
async fn get_book_and_music() -> (Book, Music)
    let book_future = get_book();
    let music_future = get_music();
    (book_future.await, music_future.await)
}
```

However, Rust futures won't do any work until they're
the two code snippets above will both run `book_futur`
than running them concurrently. To correctly run the t
`futures::join!`:

```
use futures::join;

async fn get_book_and_music() -> (Book, Music)
    let book_fut = get_book();
    let music_fut = get_music();
    join!(book_fut, music_fut)
}
```

The value returned by `join!` is a tuple containing the

`try_join!`

For futures which return `Result`, consider using `try_` `join!` only completes once all subfutures have compl futures even after one of its subfutures has returned a

Unlike `join!`, `try_join!` will complete immediately i error.

```
use futures::try_join;

async fn get_book() -> Result<Book, String> { /
async fn get_music() -> Result<Music, String> {

async fn get_book_and_music() -> Result<(Book,
    let book_fut = get_book();
    let music_fut = get_music();
    try_join!(book_fut, music_fut)
}
```

Note that the futures passed to `try_join!` must all ha using the `.map_err(|e| ...)` and `.err_into()` funct `futures::future::TryFutureExt` to consolidate the e

```
use futures::{
    future::TryFutureExt,
    try_join,
};

async fn get_book() -> Result<Book, ()> { /* ..
async fn get_music() -> Result<Music, String> {

async fn get_book_and_music() -> Result<(Book,
    let book_fut = get_book().map_err(|()| "Una
    let music_fut = get_music();
    try_join!(book_fut, music_fut)
}
```

select!

The `futures::select` macro runs multiple futures sin
respond as soon as any future completes.

```
use futures::{
    future::FutureExt, // for `.fuse()`
    pin_mut,
    select,
};

async fn task_one() { /* ... */ }
async fn task_two() { /* ... */ }

async fn race_tasks() {
    let t1 = task_one().fuse();
    let t2 = task_two().fuse();

    pin_mut!(t1, t2);

    select! {
        () = t1 => println!("task one completed
        () = t2 => println!("task two completed
    }
}
```

The function above will run both `t1` and `t2` concurre
the corresponding handler will call `println!`, and the
the remaining task.

The basic syntax for `select` is `<pattern> = <express`
many futures as you would like to `select` over.

`default => ...` and `complete =`

`select` also supports `default` and `complete` branch

A `default` branch will run if none of the futures being
`select` with a `default` branch will therefore always r
be run if none of the other futures are ready.

`complete` branches can be used to handle the case wl
have completed and will no longer make progress. Thi
`select!`.

```
use futures::{future, select};

async fn count() {
    let mut a_fut = future::ready(4);
    let mut b_fut = future::ready(6);
    let mut total = 0;

    loop {
        select! {
            a = a_fut => total += a,
            b = b_fut => total += b,
            complete => break,
            default => unreachable!(), // never
complete)
        };
    }
    assert_eq!(total, 10);
}
```

Interaction with `Unpin` and `Fused`

One thing you may have noticed in the first example a
on the futures returned by the two `async fn` s, as well
of these calls are necessary because the futures used i
`Unpin` trait and the `FusedFuture` trait.

`Unpin` is necessary because the futures used by `sele`
mutable reference. By not taking ownership of the futu
again after the call to `select`.

Similarly, the `FusedFuture` trait is required because `s`
has completed. `FusedFuture` is implemented by futur
have completed. This makes it possible to use `select`
which still have yet to complete. This can be seen in th
`b_fut` will have completed the second time through tl
by `future::ready` implements `FusedFuture`, it's able

Note that streams have a corresponding `FusedStream`
trait or have been wrapped using `.fuse()` will yield `F`
`.next()` / `.try_next()` combinators.

```
use futures::{
    stream::{Stream, StreamExt, FusedStream},
    select,
};

async fn add_two_streams(
    mut s1: impl Stream<Item = u8> + FusedStrea
    mut s2: impl Stream<Item = u8> + FusedStrea
) -> u8 {
    let mut total = 0;

    loop {
        let item = select! {
            x = s1.next() => x,
            x = s2.next() => x,
            complete => break,
        };
        if let Some(next_num) = item {
            total += next_num;
        }
    }

    total
}
```

Concurrent tasks in a select loop FuturesUnordered

One somewhat hard-to-discover but handy function is constructing an empty future which is already termina future that needs to be run.

This can be handy when there's a task that needs to be is created inside the `select` loop itself.

Note the use of the `.select_next_some()` function. Th run the branch for `Some(_)` values returned from the

```
use futures::{
    future::{Fuse, FusedFuture, FutureExt},
    stream::{FusedStream, Stream, StreamExt},
    pin_mut,
    select,
};

async fn get_new_num() -> u8 { /* ... */ 5 }

async fn run_on_new_num(_: u8) { /* ... */ }

async fn run_loop(
    mut interval_timer: impl Stream<Item = ()>
    starting_num: u8,
) {
    let run_on_new_num_fut = run_on_new_num(sta
    let get_new_num_fut = Fuse::terminated();
    pin_mut!(run_on_new_num_fut, get_new_num_fu
    loop {
        select! {
            () = interval_timer.select_next_som
                // The timer has elapsed. Start
                // if one was not already runni
                if get_new_num_fut.is_terminate
                    get_new_num_fut.set(get_new
                }
            },
            new_num = get_new_num_fut => {
                // A new number has arrived --
`run_on_new_num_fut`,
                // dropping the old one.
                run_on_new_num_fut.set(run_on_n
            },
            // Run the `run_on_new_num_fut`
            () = run_on_new_num_fut => {},
            // panic if everything completed, s
            // keep yielding values indefinitel
            complete => panic!("`interval_timer
        }
    }
}
```

When many copies of the same future need to be run
`FuturesUnordered` type. The following example is sim
copy of `run_on_new_num_fut` to completion, rather tha
created. It will also print out a value returned by `run_c`

```
use futures::{
    future::{Fuse, FusedFuture, FutureExt},
    stream::{FusedStream, FuturesUnordered, Str
    pin_mut,
    select,
};

async fn get_new_num() -> u8 { /* ... */ 5 }

async fn run_on_new_num(_: u8) -> u8 { /* ... *

async fn run_loop(
    mut interval_timer: impl Stream<Item = ()>
    starting_num: u8,
) {
    let mut run_on_new_num_futs = FuturesUnorde
    run_on_new_num_futs.push(run_on_new_num(sta
    let get_new_num_fut = Fuse::terminated();
    pin_mut!(get_new_num_fut);
    loop {
        select! {
            () = interval_timer.select_next_som
                // The timer has elapsed. Start
                // if one was not already runni
                if get_new_num_fut.is_terminate
                    get_new_num_fut.set(get_new
                }
            },
            new_num = get_new_num_fut => {
                // A new number has arrived --
`run_on_new_num_fut`.
                run_on_new_num_futs.push(run_on
            },
            // Run the `run_on_new_num_futs` an
            res = run_on_new_num_futs.select_ne
                println!("run_on_new_num_fut re
            },
            // panic if everything completed, s
            // keep yielding values indefinitel
            complete => panic!("`interval_timer
        }
    }
}
```

Spawning

Spawning allows you to run a new asynchronous task i
continue executing other code while it runs.

Say we have a web server that wants to accept connec
thread. To achieve this, we can use the `async_std::ta`
a new task that handles the connections. This function
`JoinHandle`, which can be used to wait for the result c

```
use async_std::{task, net::TcpListener, net::Tc
use futures::AsyncWriteExt;

async fn process_request(stream: &mut TcpStream
    stream.write_all(b"HTTP/1.1 200 OK\r\n\r\n"
    stream.write_all(b"Hello World").await?;
    Ok(())
}

async fn main() {
    let listener = TcpListener::bind("127.0.0.1
    loop {
        // Accept a new connection
        let (mut stream, _) = listener.accept()
        // Now process this request without blo
        task::spawn(async move {process_request
    }
}
```

The `JoinHandle` returned by `spawn` implements the
get the result of the task. This will block the current tas
the task is not awaited, your program will continue exe
cancelling it if the function is completed before the tas

```
use futures::future::join_all;
async fn task_spawner(){
    let tasks = vec![
        task::spawn(my_task(Duration::from_secs
        task::spawn(my_task(Duration::from_secs
        task::spawn(my_task(Duration::from_secs
    ];
    // If we do not await these tasks and the f
dropped
    join_all(tasks).await;
}
```

To communicate between the main task and the spaw
provided by the async runtime used.

Workarounds to Know an

Rust's `async` support is still fairly new, and there are a
still under active development, as well as some subpar
some common pain points and explain how to work ar

? in async Blocks

Just as in `async fn`, it's common to use `?` inside `asyn`
`async` blocks isn't explicitly stated. This can cause the
of the `async` block.

For example, this code:

```
let fut = async {
    foo().await?;
    bar().await?;
    Ok(())
};
```

will trigger this error:

```
error[E0282]: type annotations needed
 --> src/main.rs:5:9
  |
4 |     let fut = async {
  |         --- consider giving `fut` a type
5 |         foo().await?;
  |         ^^^^^^^^^^^^ cannot infer type
```

Unfortunately, there's currently no way to "give `fut` a
the return type of an `async` block. To work around thi
supply the success and error types for the `async` bloc

```
let fut = async {
    foo().await?;
    bar().await?;
    Ok::<(), MyError>(()) // <- note the explic
};
```

Send Approximation

Some `async fn` state machines are safe to be sent ac
Whether or not an `async fn` `Future` is `Send` is deter
held across an `.await` point. The compiler does its be
be held across an `.await` point, but this analysis is to
today.

For example, consider a simple non- `Send` type, perha

```
use std::rc::Rc;

#[derive(Default)]
struct NotSend(Rc<()>);
```

Variables of type `NotSend` can briefly appear as tempo
resulting `Future` type returned by the `async fn` mus

```
async fn bar() {}
async fn foo() {
    NotSend::default();
    bar().await;
}

fn require_send(_: impl Send) {}

fn main() {
    require_send(foo());
}
```

However, if we change `foo` to store `NotSend` in a vari

```
async fn foo() {
    let x = NotSend::default();
    bar().await;
}
```

```
error[E0277]: `std::rc::Rc<()>` cannot be sent
  --> src/main.rs:15:5
   |
15 |     require_send(foo());
   |     ^^^^^^^^^^^^ `std::rc::Rc<()>` cannot
   |
   = help: within `impl std::future::Future`, t
not implemented for `std::rc::Rc<()>`
   = note: required because it appears within t
   = note: required because it appears within t
std::future::Future, ()}`
   = note: required because it appears within t
generator@src/main.rs:7:16: 10:2 {NotSend, impl
   = note: required because it appears within t
`std::future::GenFuture<[static generator@src/m
std::future::Future, ()}]>`
   = note: required because it appears within t
std::future::Future`
   = note: required because it appears within t
std::future::Future`
note: required by `require_send`
  --> src/main.rs:12:1
   |
12 | fn require_send(_: impl Send) {}
   | ^^^^^^^^^^^^^^^^^^^^^^^^^^^^^

error: aborting due to previous error

For more information about this error, try `rus
```

This error is correct. If we store `x` into a variable, it wo `.await`, at which point the `async fn` may be running `Send`, allowing it to travel across threads would be un: would be to `drop` the `Rc` before the `.await`, but unfo

In order to successfully work around this issue, you ma encapsulating any non-`Send` variables. This makes it e these variables do not live across an `.await` point.

```rust
async fn foo() {
    {
        let x = NotSend::default();
    }
    bar().await;
}
```

Recursion

Internally, `async fn` creates a state machine type con
`.await`ed. This makes recursive `async fn`s a little tric
type has to contain itself:

```
// This function:
async fn foo() {
    step_one().await;
    step_two().await;
}
// generates a type like this:
enum Foo {
    First(StepOne),
    Second(StepTwo),
}

// So this function:
async fn recursive() {
    recursive().await;
    recursive().await;
}

// generates a type like this:
enum Recursive {
    First(Recursive),
    Second(Recursive),
}
```

This won't work—we've created an infinitely-sized type

```
error[E0733]: recursion in an `async fn` requir
 --> src/lib.rs:1:22
  |
1 | async fn recursive() {
  |                      ^ an `async fn` cannot
  |
  = note: a recursive `async fn` must be rewrit
```

In order to allow this, we have to introduce an indirect
compiler limitations mean that just wrapping the calls
enough. To make this work, we have to make `recursi`
returns a `.boxed()` `async` block:

```rust
use futures::future::{BoxFuture, FutureExt};

fn recursive() -> BoxFuture<'static, ()> {
    async move {
        recursive().await;
        recursive().await;
    }.boxed()
}
```

async in Traits

Currently, `async fn` cannot be used in traits on the st
November 2022, an MVP of async-fn-in-trait is availabl
compiler tool chain, see here for details.

In the meantime, there is a work around for the stable
from crates.io.

Note that using these trait methods will result in a hea
not a significant cost for the vast majority of applicatio
deciding whether to use this functionality in the public
expected to be called millions of times a second.

The Async Ecosystem

Rust currently provides only the bare essentials for wr
executors, tasks, reactors, combinators, and low-level
provided in the standard library. In the meantime, com
fill in these gaps.

The Async Foundations Team is interested in extendin
multiple runtimes. If you're interested in contributing t
on Zulip.

Async Runtimes

Async runtimes are libraries used for executing async a
together a *reactor* with one or more *executors*. Reactor
for external events, like async I/O, interprocess commu
runtime, subscribers are typically futures representing
handle the scheduling and execution of tasks. They ke
tasks, poll futures to completion, and wake tasks wher
"executor" is frequently used interchangeably with "ru
"ecosystem" to describe a runtime bundled with comp

Community-Provided Async Crate

The Futures Crate

The `futures` crate contains traits and functions usefu
the `Stream`, `Sink`, `AsyncRead`, and `AsyncWrite` traits
These utilities and traits may eventually become part c

`futures` has its own executor, but not its own reactor
async I/O or timer futures. For this reason, it's not cons
choice is to use utilities from `futures` with an executo

Popular Async Runtimes

There is no asynchronous runtime in the standard libr
recommended. The following crates provide popular r

- Tokio: A popular async ecosystem with HTTP, gRP
- async-std: A crate that provides asynchronous co components.
- smol: A small, simplified async runtime. Provides wrap structs like `UnixStream` or `TcpListener`.
- fuchsia-async: An executor for use in the Fuchsia

Determining Ecosystem Compatib

Not all async applications, frameworks, and libraries ar every OS or platform. Most async code can be used wi frameworks and libraries require the use of a specific e not always documented, but there are several rules of library, trait, or function depends on a specific ecosyst

Any async code that interacts with async I/O, timers, in generally depends on a specific async executor or reac async expressions, combinators, synchronization types independent, provided that any nested futures are also beginning a project, it's recommended to research rele to ensure compatibility with your chosen runtime and

Notably, `Tokio` uses the `mio` reactor and defines its c including `AsyncRead` and `AsyncWrite`. On its own, it's `smol`, which rely on the `async-executor` crate, and th defined in `futures`.

Conflicting runtime requirements can sometimes be re allow you to call code written for one runtime within a `async_compat` crate provides a compatibility layer bet

Libraries exposing async APIs should not depend on a they need to spawn tasks or define their own async I/O binaries should be responsible for scheduling and run

Single Threaded vs Multi-Threade

Async executors can be single-threaded or multi-threa `executor` crate has both a single-threaded `LocalExecu`

A multi-threaded executor makes progress on several the execution greatly for workloads with many tasks, b

is usually more expensive. It is recommended to meas
when you are choosing between a single- and a multi-t

Tasks can either be run on the thread that created the
runtimes often provide functionality for spawning task
are executed on separate threads, they should still be
tasks on a multi-threaded executor, they must also be
functions for spawning non- `Send` tasks, which ensures
that spawned it. They may also provide functions for s
dedicated threads, which is useful for running blocking
libraries.

Final Project: Building a C Server with Async Rust

In this chapter, we'll use asynchronous Rust to modify server to serve requests concurrently.

Recap

Here's what the code looked like at the end of the less

`src/main.rs`:

```rust
use std::fs;
use std::io::prelude::*;
use std::net::TcpListener;
use std::net::TcpStream;

fn main() {
    // Listen for incoming TCP connections on l
    let listener = TcpListener::bind("127.0.0.1

    // Block forever, handling each request tha
    for stream in listener.incoming() {
        let stream = stream.unwrap();

        handle_connection(stream);
    }
}

fn handle_connection(mut stream: TcpStream) {
    // Read the first 1024 bytes of data from t
    let mut buffer = [0; 1024];
    stream.read(&mut buffer).unwrap();

    let get = b"GET / HTTP/1.1\r\n";

    // Respond with greetings or a 404,
    // depending on the data in the request
    let (status_line, filename) = if buffer.sta
        ("HTTP/1.1 200 OK\r\n\r\n", "hello.html
    } else {
        ("HTTP/1.1 404 NOT FOUND\r\n\r\n", "404
    };
    let contents = fs::read_to_string(filename)

    // Write response back to the stream,
    // and flush the stream to ensure the respo
    let response = format!("{status_line}{conte
    stream.write_all(response.as_bytes()).unwra
    stream.flush().unwrap();
}
```

`hello.html`:

```html
<!DOCTYPE html>
<html lang="en">
  <head>
    <meta charset="utf-8">
    <title>Hello!</title>
  </head>
  <body>
    <h1>Hello!</h1>
    <p>Hi from Rust</p>
  </body>
</html>
```

`404.html`:

```html
<!DOCTYPE html>
<html lang="en">
  <head>
    <meta charset="utf-8">
    <title>Hello!</title>
  </head>
  <body>
    <h1>Oops!</h1>
    <p>Sorry, I don't know what you're asking f
  </body>
</html>
```

If you run the server with `cargo run` and visit `127.0.(`
greeted with a friendly message from Ferris!

Running Asynchronous C

An HTTP server should be able to serve multiple client
wait for previous requests to complete before handlin
this problem by creating a thread pool where each cor
Here, instead of improving throughput by adding threa
using asynchronous code.

Let's modify `handle_connection` to return a future by

```
async fn handle_connection(mut stream: TcpStrea
    //<-- snip -->
}
```

Adding `async` to the function declaration changes its r
type that implements `Future<Output=()>` .

If we try to compile this, the compiler warns us that it w

```
$ cargo check
    Checking async-rust v0.1.0 (file:///project
warning: unused implementer of `std::future::Fu
  --> src/main.rs:12:9
   |
12 |         handle_connection(stream);
   |         ^^^^^^^^^^^^^^^^^^^^^^^^^^
   |
   = note: `#[warn(unused_must_use)]` on by def
   = note: futures do nothing unless you `.awai
```

Because we haven't `await` ed or `poll` ed the result of
you run the server and visit `127.0.0.1:7878` in a brow
refused; our server is not handling requests.

We can't `await` or `poll` futures within synchronous c
asynchronous runtime to handle scheduling and runni
consult the section on choosing a runtime for more inf
executors, and reactors. Any of the runtimes listed will
examples, we've chosen to use the `async-std` crate.

Adding an Async Runtime

The following example will demonstrate refactoring sy
runtime; here, `async-std` . The `#[async_std::main]` a

write an asynchronous main function. To use it, enable
std in Cargo.toml:

```
[dependencies.async-std]
version = "1.6"
features = ["attributes"]
```

As a first step, we'll switch to an asynchronous main fu
returned by the async version of handle_connection.
responds. Here's what that would look like:

```
#[async_std::main]
async fn main() {
    let listener = TcpListener::bind("127.0.0.1
    for stream in listener.incoming() {
        let stream = stream.unwrap();
        // Warning: This is not concurrent!
        handle_connection(stream).await;
    }
}
```

Now, let's test to see if our server can handle connecti
handle_connection asynchronous doesn't mean that
connections at the same time, and we'll soon see why.

To illustrate this, let's simulate a slow request. When a
127.0.0.1:7878/sleep, our server will sleep for 5 sec

```
use std::time::Duration;
use async_std::task;

async fn handle_connection(mut stream: TcpStrea
    let mut buffer = [0; 1024];
    stream.read(&mut buffer).unwrap();

    let get = b"GET / HTTP/1.1\r\n";
    let sleep = b"GET /sleep HTTP/1.1\r\n";

    let (status_line, filename) = if buffer.sta
        ("HTTP/1.1 200 OK\r\n\r\n", "hello.html
    } else if buffer.starts_with(sleep) {
        task::sleep(Duration::from_secs(5)).awa
        ("HTTP/1.1 200 OK\r\n\r\n", "hello.html
    } else {
        ("HTTP/1.1 404 NOT FOUND\r\n\r\n", "404
    };
    let contents = fs::read_to_string(filename)

    let response = format!("{status_line}{conte
    stream.write(response.as_bytes()).unwrap();
    stream.flush().unwrap();
}
```

This is very similar to the simulation of a slow request
important difference: we're using the non-blocking fur
instead of the blocking function `std::thread::sleep`.
if a piece of code is run within an `async fn` and `awai`
our server handles connections concurrently, we'll nee
is non-blocking.

If you run the server, you'll see that a request to `127.0`
incoming requests for 5 seconds! This is because there
can make progress while we are `await`ing the result c
section, we'll see how to use async code to handle con

Handling Connections Co

The problem with our code so far is that `listener.inc` executor can't run other futures while `listener` waits can't handle a new connection until we're done with th

In order to fix this, we'll transform `listener.incoming` blocking Stream. Streams are similar to Iterators, but c more information, see the chapter on Streams.

Let's replace our blocking `std::net::TcpListener` wit `async_std::net::TcpListener`, and update our conn `async_std::net::TcpStream`:

```
use async_std::prelude::*;

async fn handle_connection(mut stream: TcpStrea
    let mut buffer = [0; 1024];
    stream.read(&mut buffer).await.unwrap();

    //<-- snip -->
    stream.write(response.as_bytes()).await.unw
    stream.flush().await.unwrap();
}
```

The asynchronous version of `TcpListener` implement `listener.incoming()`, a change which provides two b `listener.incoming()` no longer blocks the executor. pending futures while there are no incoming TCP conn

The second benefit is that elements from the Stream c concurrently, using a Stream's `for_each_concurrent` this method to handle each incoming request concurre `Stream` trait from the `futures` crate, so our Cargo.to

```
+[dependencies]
+futures = "0.3"

 [dependencies.async-std]
 version = "1.6"
 features = ["attributes"]
```

Now, we can handle each connection concurrently by through a closure function. The closure function takes run as soon as a new `TcpStream` becomes available. A not block, a slow request will no longer prevent other

```
use async_std::net::TcpListener;
use async_std::net::TcpStream;
use futures::stream::StreamExt;

#[async_std::main]
async fn main() {
    let listener = TcpListener::bind("127.0.0.1
    listener
        .incoming()
        .for_each_concurrent(/* limit */ None,
            let tcpstream = tcpstream.unwrap();
            handle_connection(tcpstream).await;
        })
        .await;
}
```

Serving Requests in Paral

Our example so far has largely presented concurrency to parallelism (using threads). However, async code an In our example, `for_each_concurrent` processes each same thread. The `async-std` crate allows us to spawn Because `handle_connection` is both `Send` and non-bl `async_std::task::spawn`. Here's what that would loo

```
use async_std::task::spawn;

#[async_std::main]
async fn main() {
    let listener = TcpListener::bind("127.0.0.1
    listener
        .incoming()
        .for_each_concurrent(/* limit */ None,
            let stream = stream.unwrap();
            spawn(handle_connection(stream));
        })
        .await;
}
```

Now we are using both concurrency and parallelism tc time! See the section on multithreaded executors for r

Testing the TCP Server

Let's move on to testing our `handle_connection` funct

First, we need a `TcpStream` to work with. In an end-to-want to make a real TCP connection to test our code. C listener on `localhost` port 0. Port 0 isn't a valid UNIX operating system will pick an open TCP port for us.

Instead, in this example we'll write a unit test for the co correct responses are returned for the respective inpu deterministic, we'll replace the `TcpStream` with a mock

First, we'll change the signature of `handle_connection` `handle_connection` doesn't actually require an `async` any struct that implements `async_std::io::Read`, `as` `marker::Unpin`. Changing the type signature to reflect testing.

```
use async_std::io::{Read, Write};

async fn handle_connection(mut stream: impl Rea
```

Next, let's build a mock `TcpStream` that implements th `Read` trait, with one method, `poll_read`. Our mock `T` is copied into the read buffer, and we'll return `Poll::I` complete.

```
use super::*;
use futures::io::Error;
use futures::task::{Context, Poll};

use std::cmp::min;
use std::pin::Pin;

struct MockTcpStream {
    read_data: Vec<u8>,
    write_data: Vec<u8>,
}

impl Read for MockTcpStream {
    fn poll_read(
        self: Pin<&mut Self>,
        _: &mut Context,
        buf: &mut [u8],
    ) -> Poll<Result<usize, Error>> {
        let size: usize = min(self.read_dat
        buf[..size].copy_from_slice(&self.r
        Poll::Ready(Ok(size))
    }
}
```

Our implementation of `Write` is very similar, althougł `poll_write`, `poll_flush`, and `poll_close`. `poll_wri` mock `TcpStream`, and return `Poll::Ready` when com flush or close the mock `TcpStream`, so `poll_flush` an `Poll::Ready`.

```
impl Write for MockTcpStream {
    fn poll_write(
        mut self: Pin<&mut Self>,
        _: &mut Context,
        buf: &[u8],
    ) -> Poll<Result<usize, Error>> {
        self.write_data = Vec::from(buf);

        Poll::Ready(Ok(buf.len()))
    }

    fn poll_flush(self: Pin<&mut Self>, _:
 Error>> {
        Poll::Ready(Ok(()))
    }

    fn poll_close(self: Pin<&mut Self>, _:
 Error>> {
        Poll::Ready(Ok(()))
    }
}
```

Lastly, our mock will need to implement `Unpin`, signify
safely be moved. For more information on pinning anc
pinning.

```
impl Unpin for MockTcpStream {}
```

Now we're ready to test the `handle_connection` funct
`MockTcpStream` containing some initial data, we can ru
attribute `#[async_std::test]`, similarly to how we us
that `handle_connection` works as intended, we'll chec
the `MockTcpStream` based on its initial contents.

```
use std::fs;

#[async_std::test]
async fn test_handle_connection() {
    let input_bytes = b"GET / HTTP/1.1\r\n"
    let mut contents = vec![0u8; 1024];
    contents[..input_bytes.len()].clone_fro
    let mut stream = MockTcpStream {
        read_data: contents,
        write_data: Vec::new(),
    };

    handle_connection(&mut stream).await;

    let expected_contents = fs::read_to_str
    let expected_response = format!("HTTP/1
expected_contents);
    assert!(stream.write_data.starts_with(e
}
```

Appendix : Translations o

For resources in languages other than English.

- Русский
- Français
- فارسی